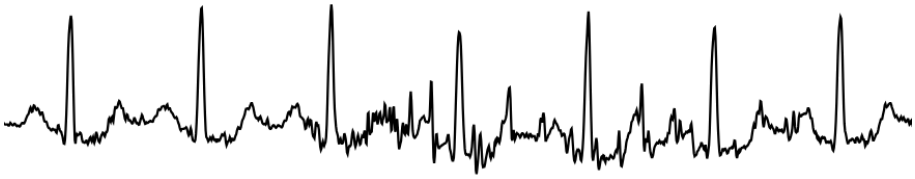
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA
Área Departamental de Engenharia Química



Electrocardiography Signal Generation System

DANIEL AFONSO DE JESUS RIBEIRO DOMINGOS
(Licenciado)

Dissertação para obtenção do grau de Mestre em Engenharia Biomédica

Orientadores:

Doutor André Ribeiro Lourenço

Doutor Hugo Plácido da Silva

Júri:

Presidente: Doutora Cecília Ribeiro da Cruz Calado

Vogais:

Doutor João Pedro Barrigana Ramos da Costa

Doutor André Ribeiro Lourenço

Novembro de 2025

Electrocardiography Signal Generation System

DANIEL AFONSO DE JESUS RIBEIRO DOMINGOS
(Licenciado)

Dissertação para obtenção do grau de Mestre em Engenharia Biomédica

Orientadores:

Doutor André Ribeiro Lourenço, ISEL/CardioID Technologies

Doutor Hugo Plácido da Silva, Instituto de Telecomunicações

Júri:

Presidente: Doutora Cecília Ribeiro da Cruz Calado, ISEL

Vogais:

Doutor João Pedro Barrigana Ramos da Costa, ISEL

Doutor André Ribeiro Lourenço, ISEL

Novembro de 2025

Acknowledgments

First and foremost, I would like to thank my supervisors Dr. André Lourenço and Dr. Hugo Silva, who guided throughout the project and whose expertise helped clarify new concepts and simplify ideas that arose with the research for this project. Without their guidance, I would have remained in a continuous loop of trying to understand concepts while adapting the system.

A note of gratitude to the remarkable team at CardioID Technologies and a particular thanks to Dr. André Lourenço for allowing the development of the project in the company space. It was impressive to see multiple areas of the room work together towards the same objective. A serious appreciation to David Velez for his extremely valuable help, particularly, the development of the hardware, expertise and suggestions for recurrent problems throughout development, and constant assistance. In addition, i extend my thanks to Dinis Vieira for insights regarding firmware programming and debugging tips. A statement of gratitude goes to Instituto de Telecomunicações for their support along the project.

A special thanks to my dad and my brother for all the support and encouragement throughout the year, and an appreciation for my aunt, uncle and my cousins for their optimism and always an ease environment whenever we visited.

This work received funding from a project developed in collaboration with IPL, ISEL, CardioID and National Public Health School, reference IPL/IDI&CA2024/M-IA-RCH_ISEL.

Machines take me by surprise with great frequency.

Alan Turing

Statement of integrity

I declare that this dissertation is the result of my personal and independent research. Its content is original, and all sources listed in the bibliographic references were consulted and are duly mentioned in the text. I further declare that all scientific and technical references relevant to the development of the work are duly cited and included in the bibliographic references.

The author

Lisbon, October 29, 2025

Resumo

Os sistemas de simulação desempenham um papel fundamental na manutenção, educação, pesquisa e treino de procedimentos clínicos, através da reprodução de condições específicas e sinais fisiológicos, como o ECG. No entanto, os simuladores de ECG convencionais evitam as complexidades presentes entre a pele e o elétrodo, essencial para a aquisição de sinais de alta qualidade que reflitam os presentes durante exames de electrocardiografia (ECG). A ausência desta estrutura em ambientes de simulação, muitas vezes, negligencia o impacto da impedância, um fator chave que influencia a qualidade e a veracidade do sinal.

Este estudo propõe um dispositivo de reprodução de ECG de 1 derivação, desenvolvido para simular a *interface* entre a pele e o elétrodo e avaliar os efeitos da impedância para com o sinal reproduzido. O sistema incorpora uma ligação por computador para controlar o dispositivo de reprodução de sinal; os sinais de ECG são, previamente, adquiridos de uma base de dados pública de sinais fisiológicos. O dispositivo desenvolvido demonstra a importância da área de contacto dos elétrodos na mitigação da distorção e na aquisição de sinais credíveis.

Este projeto destaca o potencial que o contínuo investimento em sistemas de simulação poderá ter por meio do aumento do realismo dos ambientes de treino e teste, culminando num diagnóstico mais preciso, uma melhor aprendizagem e uma manutenção mais minuciosa.

Palavras-chave

ECG; Interface Pele-Elétrodo; Impedância; Sistema Embebido; IoT

Abstract

Simulation systems play a critical role in maintenance, education, research, and procedural training, through modeling and reproduction of specific conditions and biosignals, such as ECG. However, conventional ECG simulators bypass the complexities of the skin-electrode interface, which is essential for high-quality and real-world resemblance signal acquisition during electrocardiogram (ECG) exams. The absence of this interface in simulation settings often neglects the impact of impedance, a key factor influencing signal quality and veracity.

This study introduces a novel 1-lead ECG simulation device designed to emulate the skin-electrode interface and assess the effects of impedance on ECG signal quality. The system integrates a computer interface for controlling and managing a signal generator device, with ECG signals sourced from a publicly available physiological signals database. The developed device demonstrates the significance of impedance in medical-grade ECG signal fidelity, emphasizing the importance of contact area in mitigating signal distortion.

This work highlights the potential of advanced simulation systems to enhance the realism of training and testing environments, ultimately improving diagnostic accuracy, learning and a rigorous maintenance.

Keywords

ECG; Skin-Electrode Interface; Impedance; Embedded system; IoT

Contents

List of Figures

List of Tables

List of Acronyms

ECG	Electrocardiography / Electrocardiogram
EEG	Electroencephalography
SA	Sino-atrial
AV	Atrioventricular
NCX	Sodium–calcium exchanger
GHK	Goldman–Hodgkin–Katz
RA	Right Arm
LA	Left Arm
LL	Left Leg
AHA	American Heart Association
CT	Central Terminal
EHRA	European Heart Rhythm Association
IEC	International Electrotechnical Commission
IoT	Internet of Things
MCU	Microcontroller Unit / Microcontroller
GUI	Graphic User Interface
PCB	Printed Circuit Board
DAC	Digital-to-Analog Converter
BLE	Bluetooth Low Energy
ADC	Analog-to-Digital Converter
SPI	Serial Peripheral Interface
WFDB	Waveform Database

I/O Input-Output

GPIO General Purpose Input-Output

RTOS Real-Time Operating System

UML Unified Modeling Language

ISR Interrupt Service Routine

MOSI Master-Out Slave-In

MISO Master-In Slave-Out

CS Chip Select

SCLK Serial Clock

DMA Direct Memory Access

JTAG Joint Test Action Group

SPST Single-Pole Single-Throw

IC Integrated Circuit

VFS Virtual File System

CW-FE Eq Model CardioWheel Front-End Equivalent Model

Chapter 1

Introduction

1.1 Context

Electrocardiography is one of the primary clinical tools in patient monitoring and the diagnosis of conditions associated with different physiological systems. In a hospital setting, medical-grade ECG monitoring is associated with the analysis of patient electrical cardiac activity using electrodes placed in the body, allowing the diagnosis of non-cardiovascular and cardiovascular related diseases, such as atrial fibrillation, arrhythmia or heart failure.

In the past years, the advancement of technology allowed the development of single-lead medical-grade acquisition devices (commonly known as "wearables"), such as the Apple Watch, ScanWatch 2 or Samsung Galaxy Watch 7 Pro, among others, allowing constant monitoring of biomedical parameters, such as oxygen and activity levels, heart rate, blood pressure, among others, by the users during daily activities. The development of new acquisition systems outside a clinical environment is reinforced by the growing demand of consumers for health monitoring capabilities. Although these devices present multiple properties, and might even present medical-grade features, they are uniquely an acquisition system and do not perform a diagnosis, being the responsibility of the physician.

1.2 Problem Statement

Worldwide, more than 300 million electrocardiograms are performed annually in a clinical setting [1] and can be associated with the diagnosis of cardiovascular diseases, such as arrhythmia, ventricular tachycardia or premature ventricular contraction. The reliability of the data presented in the acquisition systems is dependent on the quality of the maintenance performed on the equipment and the capacity of the test equipment to evaluate the acquisition system accordingly.

In concern to ECG acquisition systems, such as vital signs monitors, defibrillators or ECG exams systems, the maintenance is performed attaching electrode connectors to an equipment that generates specific signals for each lead, thus considering an 'ideal condition' where signals are acquired directly from the heart conductive system without considering any type of

interference. The development of higher-quality ECG systems requires the advancement of simulation devices that consider possible sources of interference associated with acquisition devices to test the validity of the data provided to medical professionals.

1.3 Objectives

In an ECG, the development of a diagnosis is intrinsically dependent on the data provided by the acquisition device, exposed to various sources of signal distortion, such as the skin-electrode interface or the power supply. This project aims to assess this point with the development of an embedded ECG simulation device that aims to simulate the effects on the contact area of electrodes in the skin through impedance.

The simulation system will consist of two main parts: software interface and control; and the firmware of the embedded system. The current advancements in the Internet of Things (IoT) opens the possibility for the development of the system in that direction.

1.4 Document Structure

The present document is organized as follows:

- **Chapter 1 - Introduction** presents the motivations and objectives of this project in the current scenario and the organization of the present document.
- **Chapter 2 - Background** reader presents the electrophysiological processes associated with the cardiac cycle responsible for the heart contraction/dilation mechanism, the origin of biopotentials and the development of the ECG signal, the concepts of heart anatomy and the history and current state of electrocardiography, the configurations of ECG leads and the skin-electrode interface.
- **Chapter 3 - State of the Art** describes the current advancements and technologies used by healthcare professionals, engineers and researchers, comprising of software and hardware simulation devices, and an insight of the proposed device.
- **Chapter 4 - ECG Signal Generator Design** provides information about the medical-grade database used in this project, signal sources and resources associated with this project. Afterwards, it describes the architecture of the system, including the organization of software and hardware, implementation, firmware development, communication protocols, microcontroller properties, and Printed Circuit Board (PCB) design.
- **Chapter 5 - System Evaluation** presents the performance tests executed in the ECG simulator and discusses the achieved results.
- **Chapter 6 - Conclusion** discuss the overall results of the project, presents the challenges throughout its development and improvements to be taken into account. The chapter also presents multiple promising branches for the continued development of this system.

Chapter 2

Background

2.1 Physiological Signals

Wearable physiological signal monitoring devices is a technology that as gained significant attraction in recent years due to its potential of revolutionizing the healthcare industry and quality, enabling a continuous and non-invasive monitoring, facilitating home care and monitoring of different patients in different locations and, in the end, improving the quality of the treatment and reducing the flow of patients in healthcare facilities [2, 3].

Physiological signals are the result of electrochemical activity of excitable cells, such as neurons or cardiac cells, associated with membrane movement of electrically charged particles (*ions*), such as sodium, calcium or potassium [4, 5]. The main property of wearable devices is its compact design, acquisition of multiple physiological parameters over extended periods of time and integration with applications that perform the recording, analysis and real-time health status for users [3].

Signals resulting by such electrochemical activity are composed of different frequency components reflecting different exams and anatomical areas, such as electrooculography, electroencephalography (EEG), electrocardiography (ECG), electromyography or axon action potentials. These biopotentials are characterized by their low amplitude, typically ranging up to 100 mV, and a frequency spectrum extending up to 10 kHz (Figure 2.1) [6].

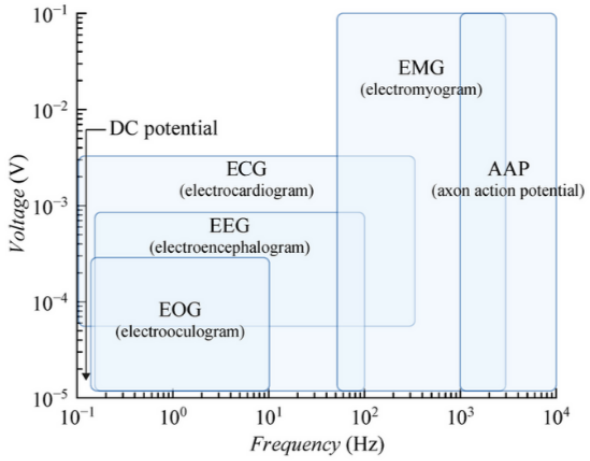


Figure 2.1 Voltage and frequency ranges of common biopotential signals. Extracted from [6].

Figure 2.1 presents different bioelectric potentials in the frequency spectrum. For the ECG simulator system presented in this study, we focus specifically on the ECG signal portion of this spectrum. Understanding these biopotential characteristics is fundamental for designing appropriate filters and signal conditioning circuits that can accurately reproduce ECG signals while maintaining signal integrity.

2.2 The Heart

The cardiovascular system consists of a closed network of arteries, veins and capillaries where the blood flows through the body pumped by the heart. The heart is associated with two circuits present in the body, namely pulmonary and systemic. The first is developed on the right side of the organ and is responsible for transporting blood to and from the lungs for carbon dioxide exhalation [4, 7]. The systemic circulation transports oxygenated blood from the lungs to all tissues in the body, where aerobic cellular respiration occurs, and carbon dioxide is released and sent back to the pulmonary circulation. The necessity of sending blood to a wide network of vessels in the systemic circulation explains the physiology of the heart, with a thicker wall on the left side compared to the right side, as well as its wrapped muscle pattern [4, 7].

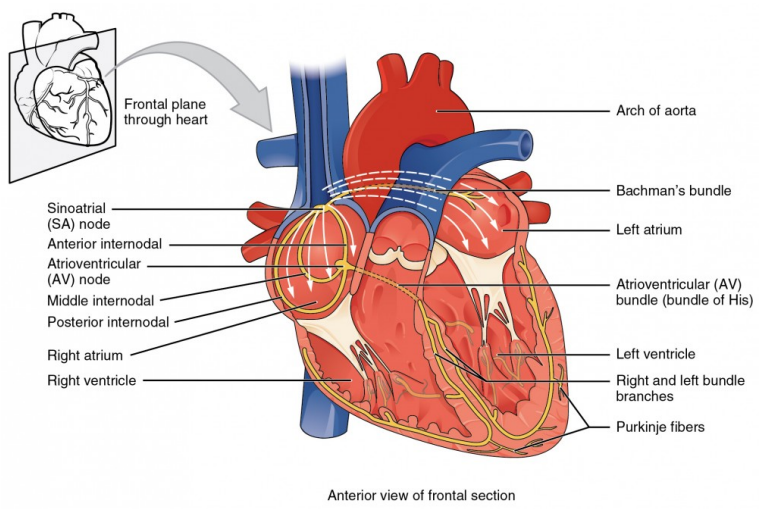


Figure 2.2 Heart electrical stimulation and propagation system. Extracted from [8].

2.2.1 Impulse Development in the Heart

Cardiac contraction is preceded by electrical excitation, which under normal physiological conditions is initiated by the sino-atrial (SA) node through the generation of an action potential. An action potential represents the rapid sequence of depolarization and repolarization phases that occur across the cellular membrane, generating an electrical impulse that propagates throughout the heart's conduction system, stimulating heart muscle [4, 7]. This process depends on specialized myocardial cells, named pacemaker cells, which have the capacity to depolarize rhythmically and initiate an action potential without external intervention (automaticity) [4, 9].

At the cellular level, the generation of an action potential requires the maintenance of ionic concentration gradients across the cell membrane, primarily involving sodium (Na^+), calcium (Ca^{2+}) and potassium (K^+) ions. These gradients are characterized by higher concentrations of sodium and calcium in the extracellular environment, while potassium maintains a higher concentration within the intracellular compartment [4, 9]. The electrochemical forces generated by both extracellular and intracellular concentration differences will promote the natural flow of ions towards an equilibrium and balance the membrane potential. In addition, the membrane includes voltage-gated ion channels to allow the passage of specific ions. The sodium-potassium pump is key in the regulation of transmembrane ion concentrations, transferring both ions against their concentration gradients at the cost of energy [4, 10]. In cardiac muscle, the creation and maintenance of potassium and sodium ions by the sodium-potassium pump is crucial for electrophysiological processes, such as generating the resting membrane potential and initiation and propagation of action potentials, as well as cell volume control, or Ca^{2+} extrusion via sodium-calcium exchanger (NCX) [10].

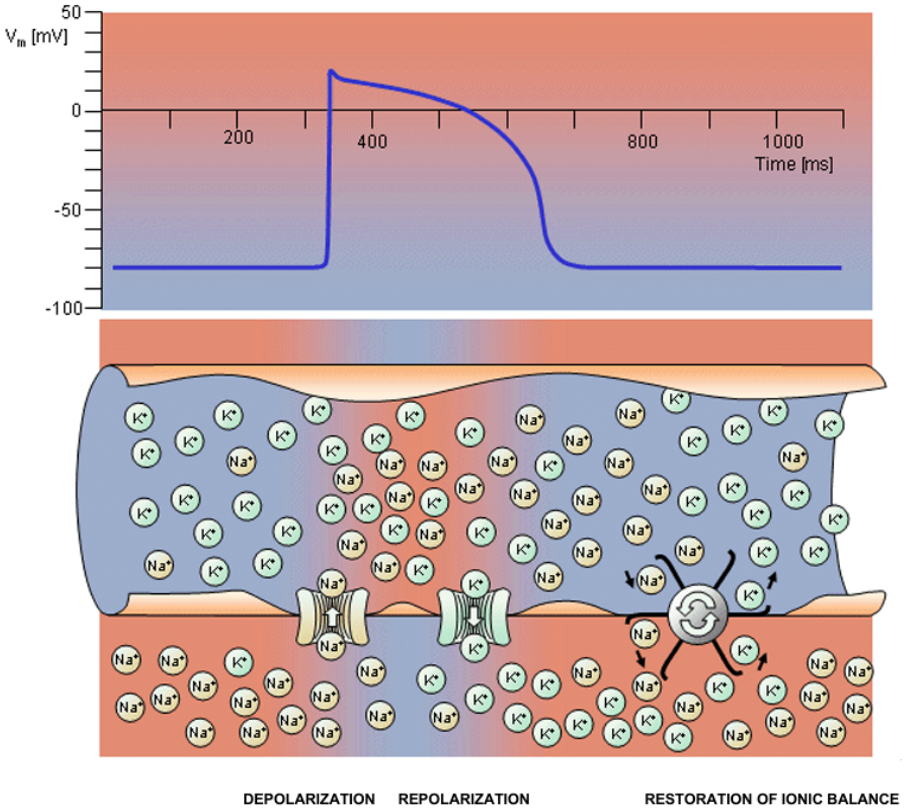


Figure 2.3 Electrophysiology of the cardiac muscle cell. In depolarization stage, Na^+ flow into the cells resulting in a negative potential outside the cells. In repolarizing stage, K^+ flows out from the cells resulting in a positive charged extracellular environment. The sodium-potassium pump is the last step to restore the initial ionic concentrations. Extracted from [5].

The membrane biopotential is described in the Goldman-Hodgkin-Katz (GHK) equation, which considers the potential and concentrations of multiple ions in intracellular and extracellular environments [4, 11]. The GHK equation is a globalization of the Nernst equation that describes the difference potential (Nernst potential) of a single-ion species opposing the force of a chemical concentration difference. Hence, the Nernst equation is an incomplete way of representing the extracellular and intracellular environment due to the existence of different ion species and potentials.

These cellular-level events coordinate the systematic sequence of depolarization and repolarization waves that generate the action potentials responsible for the electrical impulse that propagates throughout the cardiac electrical system via gap junction-mediated cell-to-cell activation, expanding a cellular event to an organ size event [4, 9].

Pacemaker cells are predominantly located in the SA and atrioventricular (AV) nodes, with some clusters in the His' bundle and Purkinje fibers. Although different structures of the electrical system of the heart contain cells with self-induction and pacemaking capabilities, the heart stimulation is developed according to the fastest pacemaker node which, in a healthy heart, is the SA node [4, 7].

Table 2.1 Rhythmicity in different structures of the heart [7].

Rhythmicity (beats/minute)	Structure
70-80	SA node
40-60	AV node
40-60	Atrial Muscle
35-40	Purkinje Fibers
20-40	Ventricular Muscle

2.2.2 Cardiac Cycle

The cardiac cycle is defined as the chain of events that occur between heartbeats [4]. Each cycle begins with a depolarization of pacemaker cells (P wave) in the SA node, generating an action potential that spreads from the atrium to the atrioventricular node through conductive fibers. The physiology of the conductive system provides a delay between the reception of the electrical impulse from the AV node and the SA node, allowing the atrium to contract before ventricular systole [4, 9]. Therefore, the atria act as primary pumps for the ventricles which then provide the main pumping power source to the cardiovascular system [4]. QRS waves appear as a result of the depolarization of the ventricles, initiating ventricular systole and increasing the pressure in the chamber. The T wave represents the stage of ventricular repolarization associated with ventricular diastole [4]. The normal rate of self-stimulation of pacemaker cells (also known as myocardial cells) is $\approx$ 70 beats/minute, thus being associated with 70 cardiac cycles [4, 7].

In normal functioning, each depolarization process is associated with a repolarization stage. Figure 2.4 shows the construction of the ECG pulse with multiple stimuli in the electrical system of the heart. Although not always present, or hard to detect in an ECG, U wave can be associated with electrolyte imbalances and/or repolarization of Purkinje fibers, and the remaining conduction system. Understanding these electrical phenomena is crucial because the biopotentials generated in the heart spread throughout the heart and can be measured reflecting each cardiac cycle behavior at specific locations of the heart.

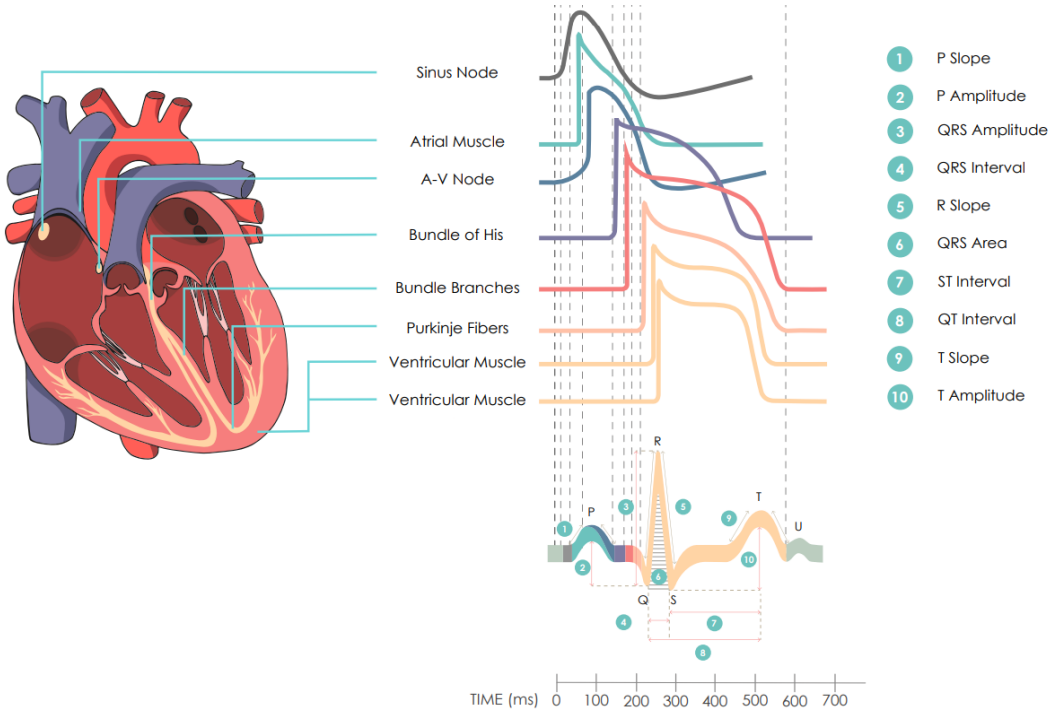


Figure 2.4 Heart electrical stimulation and propagation system. Extracted from [12].

2.3 Electrocardiography

2.3.1 Historical Development

Augustus Waller (1856-1922), a British physiologist, pioneered electrocardiography in 1887 by recording the first human ECG using Gabriel Lippmann's capillary electrometer (1873). Lippmann's electrometer operated through changes in mercury surface tension within a glass tube containing mercury and sulfuric acid in response to electrical fields, though the system suffered from poor response time [13, 14].

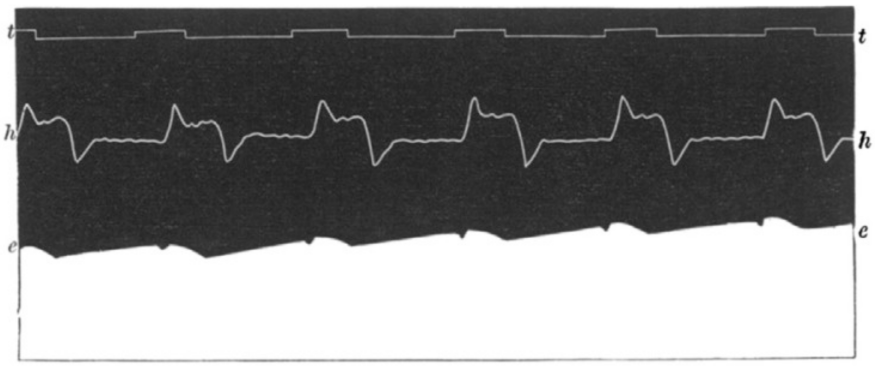


Figure 2.5 The first human electrocardiogram recorded by Augustus Waller in 1887 using a capillary electrometer and electrodes positioned on the chest and back of the subject. The waves *t.t.*, *h.h* and *e.e* represent, respectively, *time*, *cardiograph* and *electrocardiogram*. Extracted from [14].

Inspired by Waller's work, Dutch physiologist William Einthoven was able to improve the capillary electrometer and identified five deflections (ABCDE). A mathematical approach was introduced to account for the inertia of the capillary system resulting in the present waveform nomenclature PQRSTU [13, 14]. Although the capillary electrometer served as a foundation for his study of the heart's electrical activity, the limitations of the device prevented its use as a reliable diagnostic instrument. Therefore, in the beginning of the 20th century, the 3-lead string galvanometer was developed achieving higher quality signal readings and, due to its design, an adjustable sensitivity and response time by the operator [13]. The device weighted $\approx$ 272 kilograms, and was inconveniently large and immobile, however its work was carried by multiple physiologists and engineers and, in 1935, a company had reduced the weight to $\approx$ 11.34 kilograms [13, 14]. The resulting three standard, or bipolar, limb leads were used to construct Einthoven's triangle, a geometric representation associated with the positioning of each electrode that set the foundation for voltage measurements in electrocardiography. Each lead corresponds to specific electrode pairs: Lead I measures the potential difference between the left arm (LA) and right arm (RA), Lead II between the left leg (LL) and RA, and Lead III between LL and LA, creating an imaginary equilateral triangle around the heart (Einthoven's Triangle) [4].

2.3.2 Electrocardiogram

The electrical potentials generated during the cardiac cycle also spread to adjacent tissues, including the skin surface. The electrocardiogram is a record of these electrical potentials through the placement of electrodes on opposite sides of the heart [4, 7]. In a medical setting, this record is essential for diagnosing heart conditions such as coronary artery disease, ischemia, or arrhythmias, serving as a fundamental tool for assessing cardiac function and general heart health.

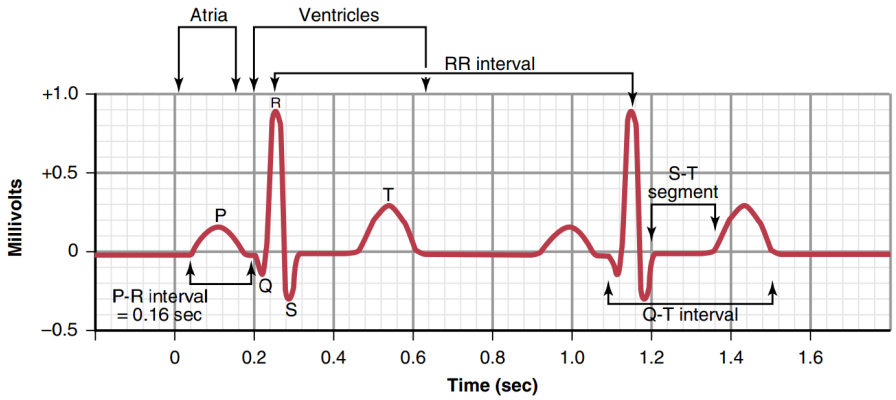


Figure 2.6 Example of a prototypical normal ECG. Extracted from [4].

The normal ECG (Figure 2.6) is composed of a P wave, a QRS complex, and a T wave. Normally, the QRS complex consists of three separate waves: Q wave, R wave, S wave; although, Q wave might be absent in some recordings. The different waves reflecting the cardiac cycle have different frequency components: P wave demonstrates frequencies between 5 to 30

Hz; the QRS complex usually contains frequencies of 8-50 Hz; the T wave exhibits components between 0 to 10 Hz [15]. The PQ or PR interval is the time between atrial depolarization and conduction of impulses through the AV node leading to excitation of the ventricles and may help diagnose diseases such as bradycardia, tachycardia, or heart block [4, 7]. The QT segment reflects the time associated with ventricle contraction and can be an indicator of myocarditis or hypocalcemia [7]. Heart rate can be calculated as the inverse of the RR interval measured between two consecutive R waves in an ECG.

2.3.3 Electrocardiogram Leads

The electrophysiology of the heart, established by sequential stages of depolarization and repolarization of myocardial tissue, allows the cardiac electrical system to be modeled as a common electrical vector field, where, predominantly, current flows from the base to the apex, and biopotentials can be constantly measured using ECG electrodes placed around the heart. Electrocardiographic leads represent specific electrode connections designed to record cardiac electrical activity in different points [4, 7].

During the first 30 years of the 20th century, Einthoven's standard leads were widely used and ECG machines were developed to become more portable. After the recognition of myocardial infarction as a clinical condition in 1910, electrocardiograms were used to differentiate cardiac from non-cardiac chest pain; however, it was noted that certain cardiac areas could not be detected by the 3-lead ECG system [5, 14].

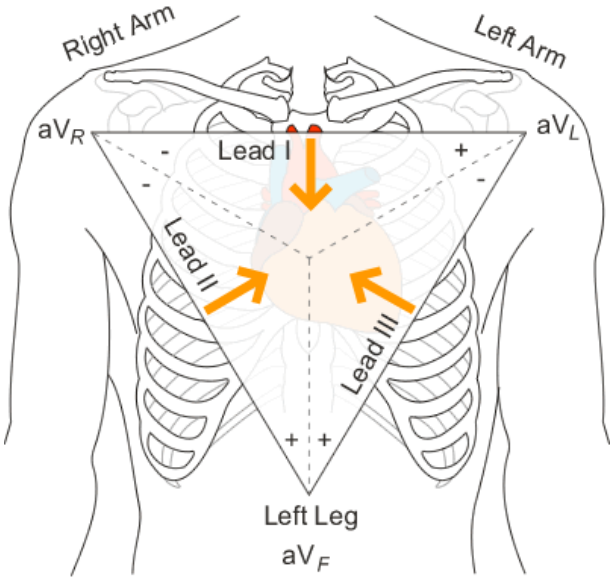


Figure 2.7 Representation of standard and augmented leads in Einthoven's Triangle. Extracted from [16].

In 1934, cardiologist Frank Wilson developed the concept of a *central terminal* (CT), an artificial ground reference created by connecting each lead to a $5k\Omega$ resistance and averaging the potentials measured from each standard limb lead. These *unipolar* leads measured electrical activity from only one electrode relative to CT, in contrast to bipolar limb leads that

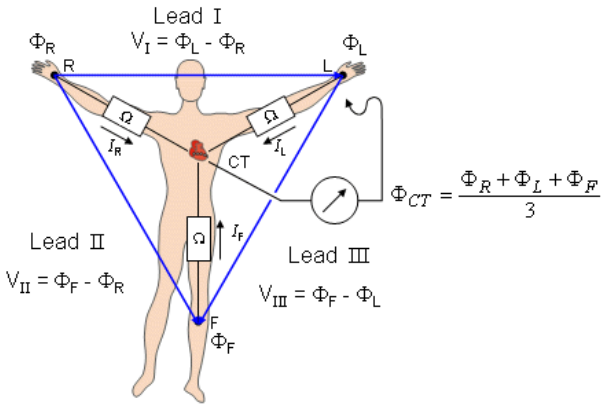


Figure 2.8 The Wilson CT is found at the center of Einthoven's triangle. Extracted from [5].

measure between two electrodes, and could theoretically be placed at any point on the body [4, 5]. The precordial leads ($V_1 - V_6$) were established following recommendations from the American Heart Association (AHA) and the Cardiac Society of Great Britain in 1938 to explore six specific areas of the chest [14].

In 1942, cardiologist Emanuel Goldberger, using a modification of Wilson's terminal, constructed three additional unipolar leads (aVL, aVR, and aVF) connected on each of the left and right arms and the left leg. This development provided a detailed coverage of the frontal plane with a hexaxial reference system with 30º increments (Figure 2.7) [4, 14]. Wilson's CT attempts to create a neutral reference point; Goldberger's modification excludes the central terminal and sets reference points in the middle of each side of Einthoven's Triangle where, for each augmented lead, only two of three bipolar limb leads contribute to the reference voltage. Removal of the standard lead from the reference voltage calculation results in a 50% increase in the amplitude of each augmented lead. The development of augmented leads was a major step towards the 12-lead ECG, which was recommended as the standard by the AHA in 1954. The resulting vector representation of the heart's biopotentials is shown in Figure 2.9 [5, 14].

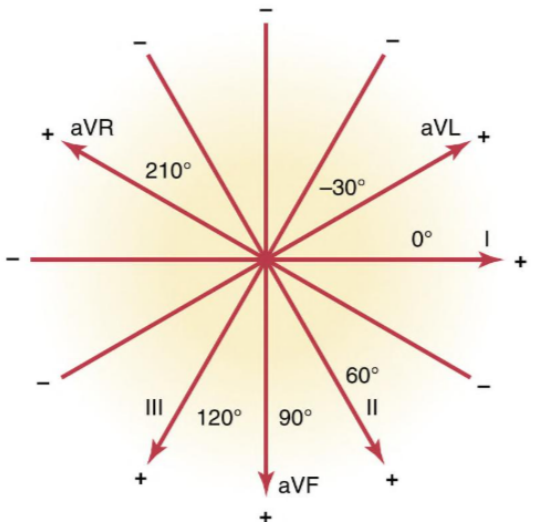


Figure 2.9 Axial representation of augmented and bipolar limb leads. Extracted from [4].

2.3.4 Electrocardiography Configurations

In a clinical environment, monitoring stations are capable of acquiring multiple biomedical signals and performing an evaluation, and they are one of the main instruments to assess the clinical condition of a patient. The reduced number of leads associated with these equipments preserves the quality of the ECG, reducing the interference associated with electrodes during medical treatment.

A 3-lead ECG system consists of two monitoring electrodes and a ground electrode where, depending on the configuration, it can be used to acquire lead I, II, or III waveforms, one at a time [17]. Two electrodes are placed below the right and left clavicles, and a third electrode is located on the lower left side of the rib cage [4]. A 5-lead system uses 4 monitoring electrodes, placed below both sides of the clavicles and rib cage. The fifth electrode is placed on one of the precordial leads, typically V_1, allowing better acquisition of heart rate and ST segment abnormalities, particularly relevant for heart conditions such as myocardial infarction [17]. In stress testing, the Mason-Likar configuration employs a 10-electrode system that modifies the standard 12-lead ECG by relocating the four limb electrodes from the extremities to the torso, significantly reducing movement artifacts and interference from equipment during exercise. The RA and LA electrodes are positioned in the depressions below the right and left clavicles, respectively, while the leg electrodes are relocated to the lower torso. The LL electrode is vertically placed in line with the LA electrode, midway between the lower border of the rib cage and the iliac crest. The right leg electrode is positioned at the intersection of a horizontal line from the LL electrode and a vertical line from the RA electrode, serving as the ground reference [5].

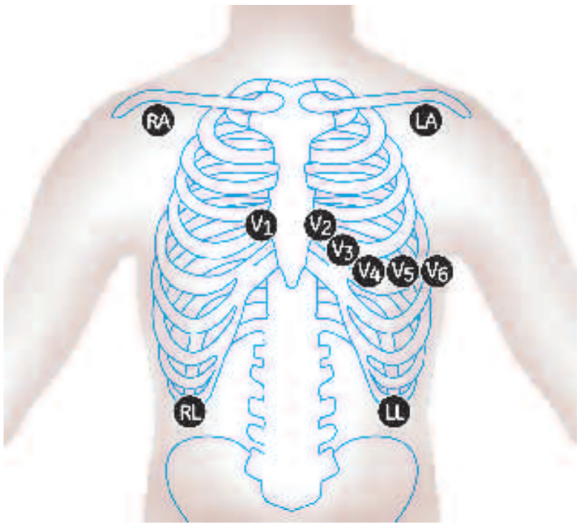


Figure 2.10 Mason-Likar 12-lead electrode placement. The precordial electrodes are positioned perpendicular to the heart, around the left side of the rib cage. The standard bipolar electrodes are located similar to 3-lead position, and a ground electrode is present in the lower right side of the rib cage following 5-lead configuration.

In 1957, cardiologist Norman Holter and his team developed the Holter monitoring, an ambulatory, long-term and non-invasive cardiac recording system. Holter devices can be selected between the common 2 to 3-lead or the 12-lead configuration depending on the purpose of the exam. A 12-lead Holter monitor is very accurate and can diagnose various diseases instantaneously, such as supraventricular tachycardia, ventricular tachycardia, atrial fibrillation or atrioventricular block. The initial device was significantly larger in comparison to the present days, consisting in an amplifier, tape recorder, electrodes and playback and analyzing units. It continuously records the cardiac electrical activity, usually, for 24 to 48 hours, during the normal daily routine of the patient. These devices allow physicians to detect and associate events that may not appear during a normal ECG exam, such as atrial fibrillation, low volume systolic dysfunction or arrhythmic events [18].

In contemporary healthcare, single-lead configurations have become the standard for wearable devices due to their compatibility with compact and lightweight designs. These devices employ high-performance filters and mathematical methods to perform a real-time acquisition, processing and display of biosignals to users, demonstrating its complex architecture. The investment in wearable devices reflects the increasing consumer interest in personal health monitoring and the constant miniaturization of the electronics and signal processing algorithms present in these systems. Wearable devices, such as smartwatches, can include a variety of sensors, such as GPS, accelerometer, photoplethysmography, Hall sensor or ECG, developing multiple features combined [19]. A single-lead ECG acquisition is represented in Figure 2.11.

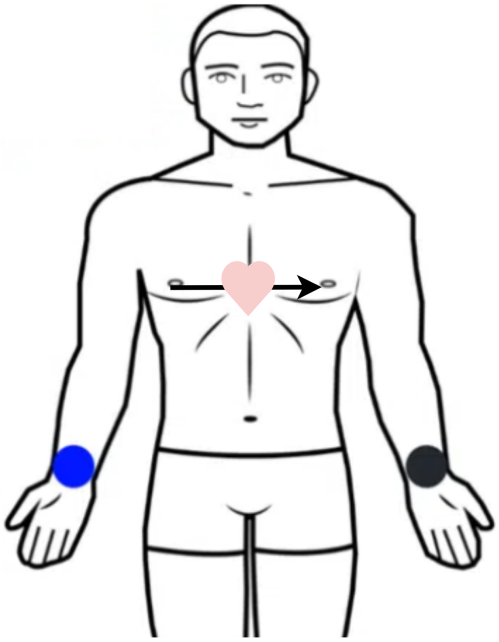


Figure 2.11 Representation of 1-lead ECG. The 2 electrodes are positioned in both wrists identified as blue and black dots.

2.3.5 Skin-Electrode Interface

The skin is the largest organ of the human body covering the entire external structure, and can be decomposed in 3 layers: the epidermis, the skin outermost layer, hosts several *"stratum"* type layers, including the *stratum corneum* as the uppermost layer, composed of keratin and dead skin cells, and part of the first line of defense of immune mechanisms; the dermis consists of 2 connective tissue membranes, housing several structures, such as sweat glands, hair follicles and hair, muscles, sensory neurons and blood vessels. The hypodermis, or subcutaneous fascia, is the deepest layer and contains adipose lobules, sensory neurons, blood vessels, hair follicles, and scanty skin appendages [5]. Between the multiple functions associated with this large and complex organ, its important to mention two particular properties for the present work: the sensory properties of the skin due to the presence of neuroreceptors to allow the interaction of an individual with the environment; the changes of properties of the skin, thus working as a diagnostic indicator [20].

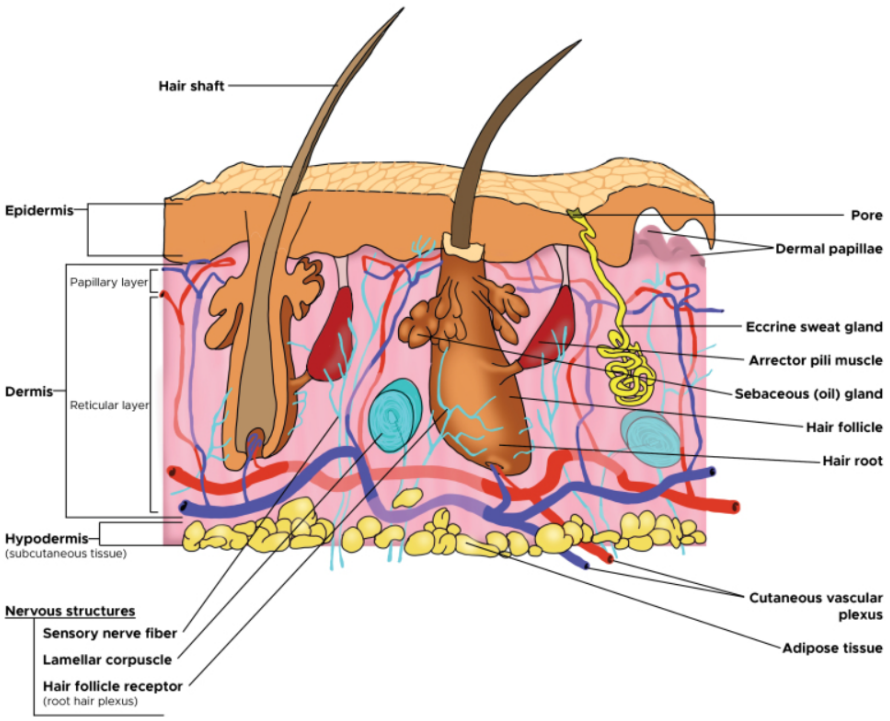


Figure 2.12 Cross section view of the skin, showing the different structures composing the layers, showing the different sweat glands, hair, hair follicles, muscles, sensory neurons and blood vessels [20].

Three fundamental components are essential for non-invasive biopotential measurement at the skin surface: the electrode; electrolyte medium; and skin. The electrode-electrolyte interface serves as the conductive pathway that facilitates optimal current flow from biological tissues to the measurement system. The electrolyte component might be from naturally body fluids, such as sweat, or specific conductive gel solutions applied between the skin and electrode surface to reduce impedance and improve the flow of current [2]. The electrode itself consists of electrically conductive materials, such as $Ag/AgCl$, usually associated with disposable materials, or Au, associated with reusable materials and providing a higher biocompatibility,

capturing biopotentials in tissues.

In general, electrodes can be categorized into wet, semi-dry or dry electrode types [21]: wet electrodes are the most commonly used and produce a high quality signal due to their conductive gel interface, which provides low initial impedance and good skin contact; semi-dry electrodes present a gelatinous contact surface and less humidity in comparison to wet electrodes, enhancing installation in patients while maintaining good signal acquisition; dry electrodes operate without electrolyte gel, which may result in higher initial impedance and potentially compromised signal quality during early acquisition phases [21, 22]. However, they provide a higher long-term performance compared to wet electrodes due to the absence of an impedance source associated with electrolyte dehydration or gel displacement, preventing signal degradation in comparison to wet electrodes [21, 22].

The electrode-skin interface can be modeled by a non-linear RC equivalent circuit. Different models have been proposed, such as Thomasset, Kirkup or Lapicque, and the interface behavior is, in the present, normally represented by Webster model (Figure 2.13) [21]. In this circuit, E_{hc} is the half-cell potential developed in the electrode-electrolyte medium with contact of the electrodes with the body, R_d and C_d model the impedance due to the electrode-electrolyte and the polarization effects, and R_s represents the interface effects associated with the electrolyte measured between the electrode and skin [22].

The equivalent circuit for a biopotential electrode is comprised of 3 sections in series: a voltage source E_{hc}; resistor R_d and capacitor C_d in parallel; a resistor R_s; enabling the simulation of multiple layers in the skin-electrode interface as modular cascaded RC circuits, where each stage reflects a physical layer [22].

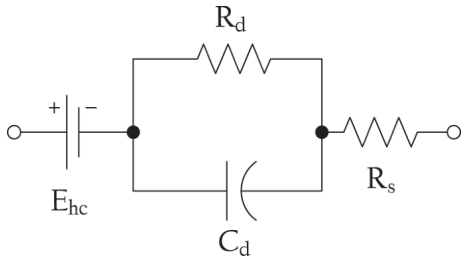


Figure 2.13 Webster equivalent circuit for a biopotential electrode-tissue interface. Extracted from [23].

The combination in a series of equivalent circuits allows the development of the skin-electrode interface, presented in Figure 2.14 for wet electrodes. The E_{se} voltage source results from a difference in the ionic concentration on the *stratum corneum*, which can be calculated using Nernst or GHK equations. The entire epidermis can be represented with a circuit, with a resistor R_e and capacitor C_e in parallel, and a resistor R_g representing the electrolyte resistance. The dermis can be treated as a parallel resistor R_d and capacitor C_d, and remaining subcutaneous tissues can be considered a simple resistance R_u [22].

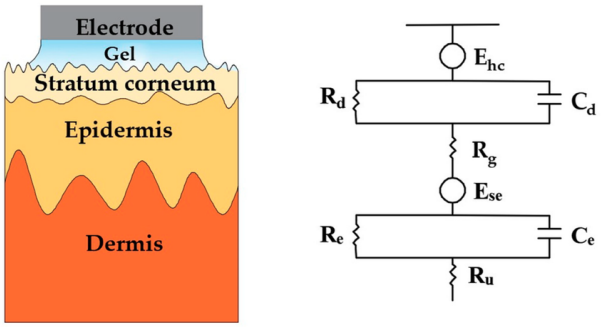


Figure 2.14 Simple schematic representation of wet skin-electrode interface (left side), and the associated equivalent electronic circuit (right side) [22].

Although the excellent properties provided by wet electrodes, their usage is generally associated with high signal quality requirements and short usage duration, such as some EEG or ECG exams, due to the required skin preparation, such as hair cutting, skin surface cleaning and gel placement, and time involved [21, 22]. Semi-dry or dry electrodes provide the best option for long-term signal acquisition due to the advantage of skipping these preparations steps, while preventing signal degradation. The main characteristic of semi-dry and dry electrodes is their ease of use and signal integrity over-time, and present a similar equivalent circuit to Figure 2.15. In wet electrodes, impedance associated with the gel is obtained by R_g, whereas in semi-dry and dry-electrodes equivalent circuit, the absence of the gel replaces R_g with an RC equivalent circuit, where R_i and C_i, represent the impedance between the electrode and skin [22]. Due to their ease of use, manufacturing and cost, semi-dry and dry electrodes are the most used in research and 'wearable' applications, being able to maintain a good skin contact in intense situations, such as motion or healthcare [22].

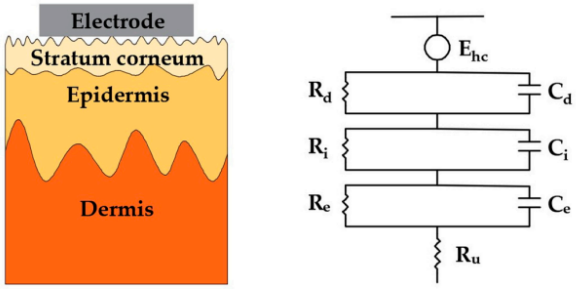


Figure 2.15 Simple schematic representation of dry/semi-dry skin-electrode interface (left side), and the associated equivalent electronic circuit (right side). The resistance R_i and capacitance C_i values reflect the electrode-*stratum corneum* contact and gaps, respectively [22].

Chapter 3

State-of-the-Art

3.1 Overview

Advances in the precision and complexity of ECG acquisition systems are accompanied by developments in technologies for generating accurate cardiac waveforms.

Simulation devices capable of accurately representing electrophysiological signals have been proposed as a teaching tool in healthcare related courses, such as biomedical engineering or medical school, as well as practice resources for medical professionals to perfect and train procedures. This approach is an advantage to both the students and the teachers by developing a bridge between the theoretical knowledge and providing a 'hands-on' experience under supervision. In the case of medical school, the current practical knowledge is time-consuming, can be exhausting to both the student and the supervisor, and could introduce a risk to the patient. Notwithstanding human-contact experience, simulators are intended to be an additional learning step in several medical procedures. The advantage of these devices was shown in a survey developed by the Scientific Initiatives Committee of the European Heart Rhythm Association (EHRA) associated with electrophysiology programs to improve skill and reduce risk. Among the 74 respondents of the survey, the large majority (81%) find simulators useful (47%) or very useful (34%). However, only 18% of the respondents have these devices available. The increase use of these devices would increase procedure efficiency, improve the management of risk and safety, and provide a better association between practical and theoretical knowledge [24].

In recent years, the evolution of artificial intelligence models has played a crucial role in developing systems capable of identifying and characterizing medical conditions, such as tumors and cardiovascular diseases, and the development of higher-quality simulation devices. The quality of the data generated by these devices provides the user with the knowledge that, if working as expected, the values presented by the acquisition devices are reliable.

3.2 ECG Testing Equipments

The current landscape of medical-grade simulation equipment with ECG generation offers a wide range of devices, such as defibrillator and electrical safety analyzers and patient simulators. Below is a list of common simulators with ECG generation capabilities:

- **Fluke Biomedical ProSim8/8P** patient simulator allows the testing of a wide range of biomedical parameters, such as 12-lead ECG, over 50 arrhythmia simulations, multiple heart conditions (tachycardia, hypotension, and hypertension, among others) and generation of pacemaker waveforms, and change simulation parameters such as heart rate, person type (adult or pediatric), oxygen levels and blood pressure (invasive and non-invasive). In addition, it has multiple performance tests with preset parameters, such as pulse generation and typical square, sine and triangle/sawtooth waveforms generated between 0.05 Hz and 150 Hz, and specific cardiac pulse waveform detection and artifact tests [25].

Figure 3.1 Top view of Fluke Biomedical ProSim8 patient simulator. On the left side, the device features 5 connections associated with invasive blood pressure, temperature, oxygen saturation and cardiac output. The top side presents 10 connectors for ECG waveforms simulation labeled, respectively, according with AHA and International Electrotechnical Commission (IEC) in the left and right sides. The center and right sides presents user interface capabilities [25].

- **PatSim 200** is a patient simulator that provides selectable stepped heart rate values between 30 and 300 BPM and amplitudes between 0.05 and 5.5 mV and associated menus of specific signals, such as arrythmias, performance waveforms, pacemaker waveform, among other selections in the menu tree.

Figure 3.2 The front and top view of the device are shown, respectively, in the left and right images. The user interaction with the device is mainly developed in the options presented in front-view and the 10 ECG connections shown in the top view comply with AHA and IEC labeling standards. Temperature, invasive blood pressure and USB connections are not shown [26].

- **MedTec & Science GMBH MS410** is a certified and more affordable ECG simulation device, capable of generating 6 beat series with 37 normal and pathological beat types, allowing the development of typical ECG acquisition and performance tests. In addition, according to IEC60601-2-25, it contains six series of beats with 114 beat types composed of typical ECG acquisition noise, such as high frequency signals, baseline wander and power supply [27].

Figure 3.3 The left and right images show the front and back views of the MS410 device, respectively. The missing pushbutton contacts, visible in the right image, are inserted before starting the device, connecting and arranging the electrodes, and testing the acquisition device [27].

3.2.1 Software-based ECG Simulators

Software-based ECG simulation devices work by continuously reproducing ECG waveforms, based on mathematical models or records in memory, without hardware changes. In 2010, van Dam *et al.* presented the ECGSIM (2.0), an interactive ECG simulation tool that can be used

for educational or research purposes. The simulation software enables users to interactively change parameters associated with the heart's conductive system, such as the voltage source value and the surface area, and observe the cause-effect relationship instantaneously with resulting simulated signals in an ECG and the surface of the heart [28, 29]. Figure 3.4 displays the main window of ECGSIM, consisting of 4 sections.

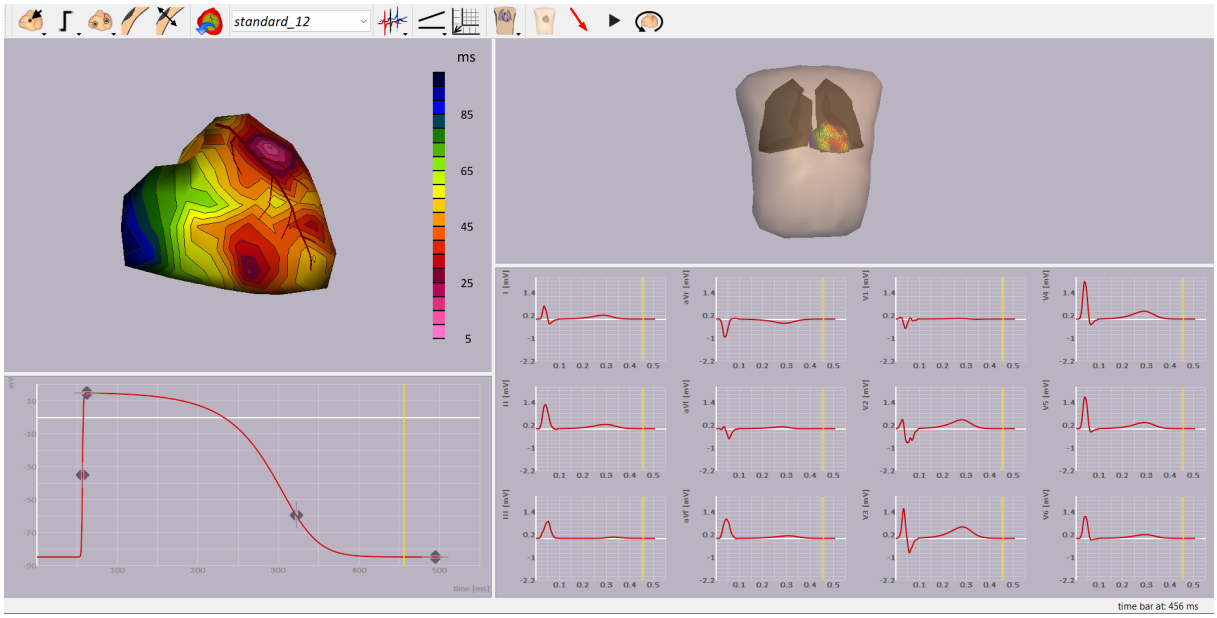


Figure 3.4 ECGSIM software simulator main window (v3.0.1.). The **top-left section** (Heart-pane) presents a geometrical representation of the atria and ventricles. The **top-right section** (thorax-pane) shows a anterior view of the body, compatible with the heart-pane view. The **bottom-left** section provides the user a pane from which the voltage source, positioned in the heart-pane, can be manipulated and the ECG waveforms can be displayed in the ECG-pane in the **bottom-right section** [29].

In addition to the features presented in Figure 3.4, the software presents other options associated with electrical stimuli propagation, anatomical visualization or signal filtering [29]:

- anatomical display of depolarization and repolarization wave propagation;
- heart and body surface potentials;
- multiple main simulation options (1-lead, 12-lead, vectorcardiogram, among others);
- baseline wander correction and AC or DC coupling;
- other options;

In 2016, Al-Rakhami, M. *et al.* proposed a new tool in the article *Cloud-based Graphical Simulation Tool of ECG for Educational Purpose*, a solely education simulator that physically separates the user interface (computer, smartphone, ...) from the main device. The overall architecture of the system was developed to enable remote operation of the device by the student, professor or health professional, and includes three modules: Graphic Viewer, responsible for the display of the menu of the system, ECG waveforms and stored ECG case files;

Case File Repository, includes and manages all case files in the system; Rhythm Generator handles all functions associated with ECG simulation. The user-friendly software generates simulated ECG signals and provides a repository with multiple readable case files that can be visualized [30]. In 2017, Coelho *et al.* presented a mobile tool designed to assist ECG interpretation. Unlike traditional simulators that generate ECG waveforms from heart models, such as ECGSIM, this system innovative approach animates a 3D heart model based on input ECG waveforms using skeletal subspace deformation algorithm. Users can manipulate the real-time visualization on smartphones or computers, viewing internal anatomy through clip planes and transparency options. The system presents two ECG rendering modes: dynamic, where a signal is continuously displayed and synchronized with heart animation; static, with a single ECG cycle display; making this software a valuable tool for research and educational purposes [31].

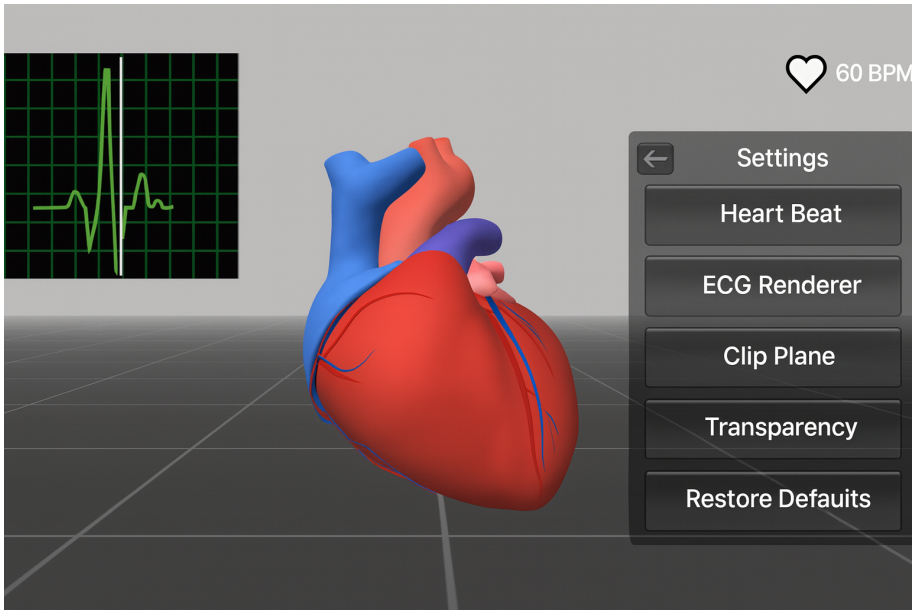


Figure 3.5 Configuration options in the simulation tool [31].

3.2.2 Hardware-based ECG Simulators

The bridge of ECG simulation systems is completed using hardware-based devices and cross from theoretical to practical visualization and knowledge. Hardware-based simulation devices provide users with a better hands-on experience with ECG signal acquisition, process and display. The development of prototype ECG simulators by researchers was made with the advantage of different microcontrollers. In 2021, Azizah, F. *et al.* proposed a 12-lead ECG simulator ("phantom ECG") based on Atmega2560 microcontroller unit (MCU) and comprised of a digital to analog converter type MCP4921 as an ECG waveform formation, a 2.4" LCD TFT display as user interface with the system, and a resistor network to provide the impedance difference of each lead. The simulation starts by setting the heartbeat (30, 60, 120 or 180 BPM) and wave amplitude (0.5, 1.0 or 2.0 mV) values. Afterwards, the generated values in the MCU will be forward to the digital-to-analog converter (DAC) and the resistor network, in sequence, to provide the impedance difference for each lead [32].

In *ECG simulator with configurable skin-electrode impedance and artifacts emulation*, Almeida *et al.* (2021) presented a single-lead ECG simulation and acquisition device that represents a significant advancement in ECG simulation with the account of skin-electrode impedance. The system is composed of 4 main modules. *Control* section hosts aN MCU and a memory card, and manages all electronic components, dataflow and configurations; in *Signal Generation*, the digital values received from the microcontroller were used to generate realistic ECG waveforms using 2 digital to analog converters with 16-bit resolution, 2 second-order reconstruction filters Sallen-Key topology, a differential attenuator circuit and 2 output buffers associated with positive and negative electrode terminals; the *Skin-electrode Model* section enables a configurable skin-electrode impedance by the user using an 8-bit resolution $1M\Omega$ digital potentiometer with I^2C interface and 2 8-bit I/O expanders, connected to a bank of capacitors, I^2C digitally controlled by the MCU. The *Signal Acquisition* module incorporates a 16-bit resolution analog to digital converter, communicating through SPI protocol with a microcontroller, and a second-order anti-aliasing filter to store test files [33].

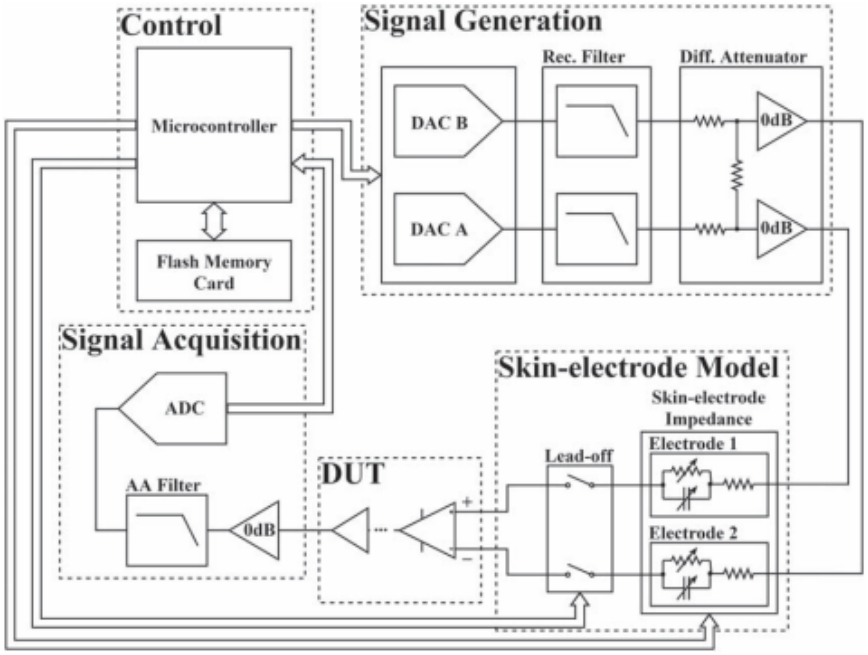


Figure 3.6 Overview of the system proposed by Almeida, D *et al.*. Extracted from [33, 34].

In 2022, Quiroz-Juárez *et al.* proposed a low-cost, versatile and tunable ECG patient simulator capable of mimic 12-lead ECG waveforms, reproducing $\sim$ 32 cardiac rhythms and simulating arrhythmia, using four different mathematical models [35, 36]:

- Modified FitzHugh-Nagumo model;
- Discretized reaction-difusion model;
- Three-coupled oscillators model;
- Quasiperiodic motions based model;

The interface with the system is performed with a LCD TFT touch screen managed by a first MCU responsible for the GUI (Graphic User Interface). The generation of ECG waveforms is executed by a second microcontroller and electronic circuitry. The 12-leads are generated using 9 reference DACs: the bipolar limb leads RA, LA and LL are generated using a Wye ("star") resistor network; unipolar augmented limb leads aVF, aVL and aVR are obtained through their relation between the physical RA, LA and LL electrodes and Goldberg's central terminal; the precordial leads $V_1 - V_6$ are unipolar potentials referenced to a common virtual electrode (Wilson central terminal).

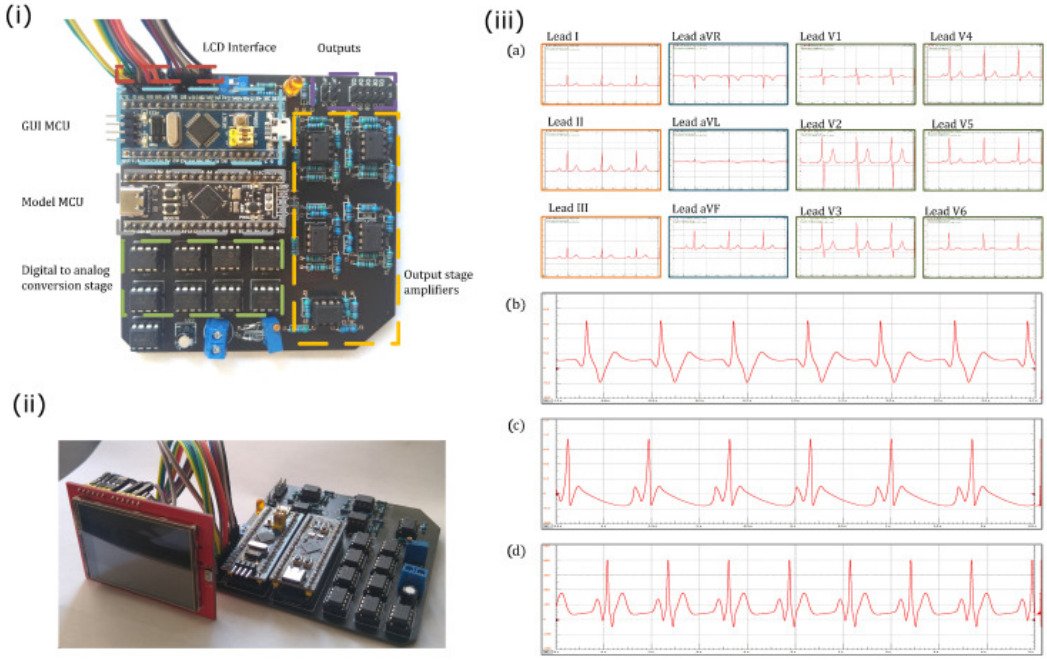


Figure 3.7 Top view of the proposed ECG simulator circuit and top-front view of the system with the user interface touch screen, respectively (i & ii). Resulting ECG waveforms generated with the simulator with each mathematical approach used (iii): a) modified FitzHugh-Nagumo; b) discretized reaction-difusion; c) three-coupled oscillators; and d) quasiperiodic motions [35].

3.3 Brief Description of the Proposed System

The device proposed in this thesis intends to be a novel system for ECG simulation signals with both advantages and disadvantages when compared to the existing state of the art devices and brands. Its core hardware follows the approach of Daniel Almeida, with the single difference of being solely a signal generation device for a front-end acquisition equipment.

3.3.1 Advantages

The main advantage of the developed system is its ability to simulate skin-electrode impedance over time using capacitor and resistor networks. The inversely proportional relationship between resistance and capacitance empowers users to model different electrode-skin contact areas, allowing for realistic simulation of various electrode types and different electrode contact

area values. The device is low-cost and presents a variety of tests and user control over all signal and system parameters. The proposed system provides control over important SD card operations, such as rename, deletion or extracting the content in the card.

3.3.2 Disadvantages and Possible Improvements

The implemented project is a 1-lead system, in contrast to the 12-lead commercially available simulators presented previously. Additionally, it does not support continuous signal generation, as it requires the ECG data to be retrieved, processed and stored in SD card. The software and firmware were continuously developed and tested using specific ECG databases from Physionet, mainly *MIT-BIH Arrhythmia Database*, thus the use of other ECG signal databases could pose an incompatibility issue. A major improvement to assess is the inclusion of Bluetooth Low Energy (BLE) or WiFi communication in the embedded system, both currently absent and important to control a single or numerous embedded systems.

Chapter 4

ECG Signal Generator Design

The following chapter will present the decisions and steps associated with the development of the simulator. At start, it presents the main data source and a brief introduction of its interaction with the system. Next, the chapter introduces the software interface between the user and the embedded system, the operations performed and organization. Afterwards, the reader will be presented with the development of the front-end of the simulator, including the back-end operation functions and front-end signal generation block.

4.1 Methodology

The signal generator prototype developed required work in three areas. The main hardware associated with the functional work of the device was developed by Daniel Almeida in his master thesis in Electronics and Telecommunications and Computers Engineering *Non-Intrusive ECG Acquisition Test-bed*, requiring adaptations for the latest developed system. The main work of the project was focused on the development of a firmware and Python interface for the embedded system.

4.1.1 Project Resources

The first steps with this project began with the development of the firmware using an ESP32-DEVKITC-32E (using an ESP32-WROOM-32E), a 2.4GHz dual-core development board with 4MB of memory and, among other features, BLE, WiFi and multiple data interfaces (SPI, DAC, ADC, among others). The constant development and improvement of the code lead to an increase of familiarity of Espressif libraries for programming and the their use to ease operations and have a better control over objects, as well as standard C/C++ library functions and POSIX library. The overall firmware was developed using Arduino IDE v2.3.5 running C/C++ v20.

The software was developed using JetBrains' Pycharm running Python v3.13. The circuit was simulated using LTSpice v17.0.37.0, an Analog Devices' free simulator, and the prototype was developed using Altium Designer v25.7.1. Later on, the physical prototype was developed using a printed circuit board (PCB) ordered from JLCPCB manufacturer.

Finally, a Tektronix TDS 2004B oscilloscope, with a sampling frequency up to 1 GS/s and

a bandwidth of 60 MHz, was employed throughout the project development phase to monitor and analyze the analog output signals and validate the system's performance, and a Vici VC99 multimeter was used to verify the integrity of connections during assembly stage of the project, modifications and validate the resistance values associated with the digital potentiometer.

4.1.2 Signal Sources

PhysioNet

This project aims to reproduce medical-grade acquisitions of ECG. These record are obtained from *PhysioBank*, one of three resources of *PhysioNet*, a public and international database of physiological signals from the *Research Resource for Complex Physiologic Signals*, a cooperative research organization established in 1999, and its managed my MIT's Laboratory for Computational Physiology. This repository hosts multiple biomedical signals, annotations and images, such as EEG, X-rays, PET-CT, and other biomedical signals.

The *MIT-BIH Arrhythmia Database* was the repository used throughout the development of the back-end of the system and is composed of 48 half-hour two-lead ECG records. These records were digitalized at a sampling rate of 360 Hz in each channel with 11-bit resolution over a 10 mV range. Furthermore, records were independently developed by two or more cardiologists in each record, resulting in approximately 110.000 annotations in the database. The *Noise Stress Test Database* is a set of 12 half-hour noise-free records, created using records 118 and 119 from the *MIT-BIH Arrhythmia Database*, and 3 half-hour records with noise, selecting ECG that contained predominantly baseline wonder (record 'bw'), muscule (EMG) artifact (record 'ma') and electrode motion artifact (record 'em'). The *MIT-BIH Normal Sinus Rhythm Database* includes 18 long-term acquisitions of individuals between 20-50 years of age. All record mentioned were sampled at frequencies between 128-360 Hz.

The database interacts with the interface system using the Waveform Database (WFDB) package, originally written in C, but extended for other popular languages, such as MATLAB, C++, FORTRAN or Python. The native WFDB package (v4.3.0) contains three subpackages (io, plot, and processing) that enable direct signal extraction from PhysioNet, signal analysis using annotations and processing tools, and signal operations such as resampling and normalization, among other functions [37, 38].

The interaction of the database with the remaining software is unidirectional. Only one waveform is extracted at a time; in multi-channel signals, the user is given the option to select the channel intended to extract the signal.

CardioID - CardioWheel

A secondary data source was the CardioWheel device, developed by CardioID Technologies, Lda. CardioWheel is a continuous 1-lead ECG signal acquisition device that uses special conductive leather on the steering wheel to capture heart biopotentials and help prevent drowsiness, evaluate cardiac health, and perform biometric recognition. The device is designed for continuous motorization in extreme conditions and can be extended to third-party devices, such

as telematics and anti-collision systems. CardioWheel interfaces with *ECG Biometrics*, an app developed by CardioID Technologies, through BLE and is capable of acquiring the location and ECG of the user. In addition, it has a biometric authentication feature. The ECG record values stored in a .TXT file include a header, composed of the date, sampling frequency and labels for three columns of data, including the sample index and the value.

4.2 System Interface Implementation

The software is responsible for interfacing and processing the data of acquired signals selected by the user. The software is responsible for acquisition, process and transmission of signals selected by the user and system configuration options. The software is composed of 7 sections, as shown in Figure **??**: the **User** can select different options presented by **Computer Processing & Interface** block; to generate a signal, the user selects the appropriate option and intended record, the same is retrieved from **PhysioNet** through **Signal Load**, and the skin-electrode impedance is defined overtime in the **Artificial Skin-Electrode Characterization** and the data is prepared for streaming in **Data Packing**. Finally, the packed data is sent in batches to **ECG Simulator**. The system will implement the top data pipeline when managing non-signal functions.

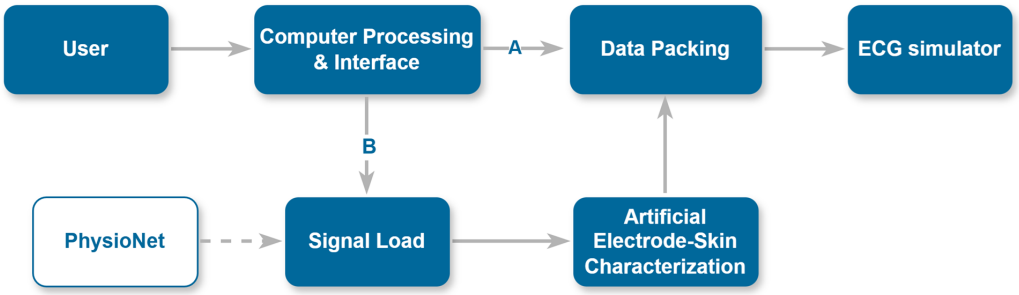


Figure 4.1 Overview of the software developed in Python, showing non-signal (A) and signal (B) data pipelines. A is used to change configurations of the embedded system or send single commands. B is used to send new signals, associated with a specific command.

The interface was developed using several established Python packages. NumPy, SciPy, Pandas, and WFDB libraries handle data management and signal processing operations. The Struct module organizes data into packets for serial transmission via the Serial library, while Matplotlib provides UI visualization capabilities. The interface system is composed of `ecgSystem` and `SignalDesigner` classes, where the second class is part of the first.

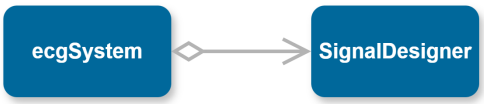


Figure 4.2 The `SignalDesigner` class is **aggregated** to the main class `ecgSystem`.

The interface is the access point to the software and the embedded system, providing three main options to start running the system: *ECG Manager*, *SD Card Manager* and *Test files*; and associated submenus. The menu tree presented in Figure 4.3 provide a better insight of the available options. In addition, some options the system architecture allows the user to return to the previous menu or, optionally and in some cases, jump between branches of the tree without the requirement of returning to the previous menu of the same branch. The first branch of the tree allows the user to start/stop resources in the simulator associated with signal generation and, in *Change ECG Signal*, select the intended ECG record to extract, process parameters and send to the simulation device. The middle branch empowers the user with file management processes by retrieving the present files in the microSD card and executing operations, such as renaming or deleting files, generating a signal, and retrieving information from a file (name, size, or last modification time). The right branch menu provides several options that allow the user to generate custom waveforms and, in addition, provides an option to read specific 'CardioWheel' signals stored in TXT files.

The interface configuration uses a Python dictionary to manage multiple parameters, including system settings, ECG acquisition parameters, and test function configurations. The usage of a dictionary, in early development, was due to its benefits, including easy maintenance, unlimited extension, and flexible data types.

```
current_config = {
    # System settings
    'file_name': '/sdcard/teste.bin',
    'sampling_frequency': 360,  # Auto-updated from ECG record
    'resampling_frequency': 360,

    # ECG signal settings
    'record': '100',
    'database': 'mitdb',
    'start_sec': 0,
    'duration_sec': 10,
    'channel': 0,

    # Test functions
    'width': 50,
    'duty_cycle': 50,
    'frequency': 1,
```

```
        'samples': 360
    }
```

Software Configuration Dictionary.

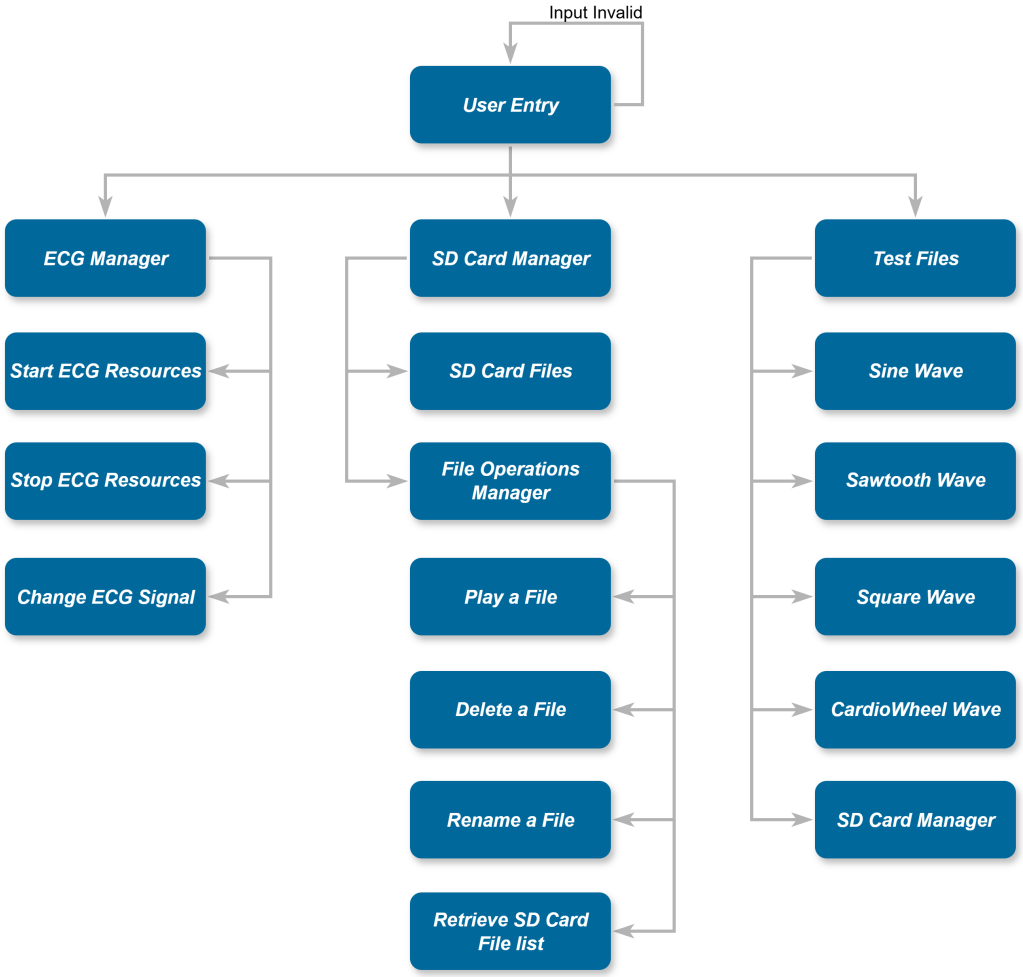


Figure 4.3 Vertical Tree view of the implemented menu and system.

4.2.1 Control Structure of Peripherals

The control of peripherals on the prototype board requires the transmission of data in a specific format. The general SPI command structures are prepared in Python using individual functions for digital potentiometer and I/O expanders, and packed before sending to the generation front-end.

Digital Potentiometer

The component uses a 16-bit command structure, composed of an 8-bit command and a data byte. The command byte is composed of two sequences of two dummy bits interlaced with 2-bit commands and 2-bit Wiper selection (Figure 4.4). The component operates in the slave mode and uses up to four interface pins ($\overline{CS}$, SCLK, MOSI, MISO). The SPI interface is specified to operate up to 10 MHz and supports modes 0 and 3.

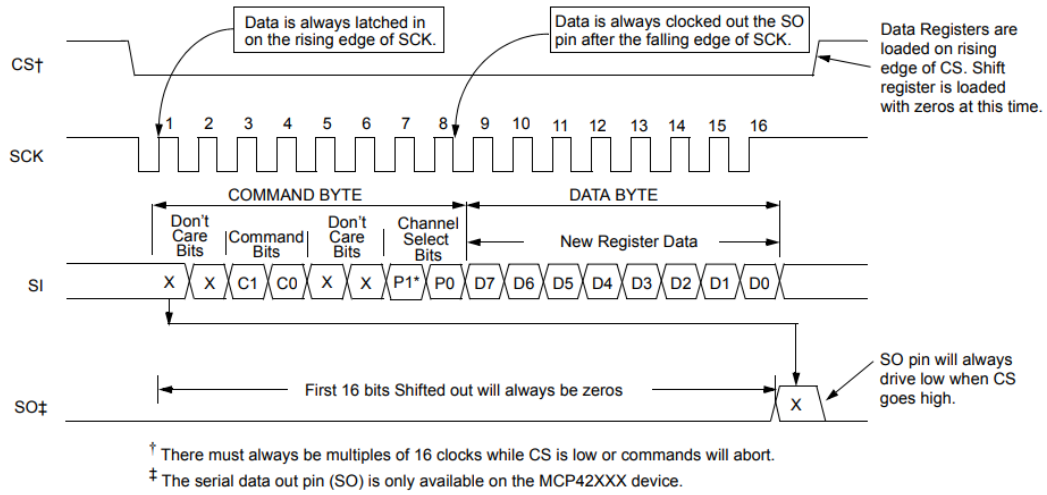


Figure 4.4 MCP42100 16-bit SPI command structure [39].

The function `mcp42100_command` is comprised of `command`, `pots` and `data` attributes to create the 16-bits sequence. The command and data bytes were built using bitwise operations, as shown below.

```
def mcp42100_command(self, command='write', pots='both', data=0x80):

    # Command codes (bits 4-5 of command byte)
    command_bits = {
        'nop': 0b00,  # no operation
        'write': 0b01,  # write data
        'shutdown': 0b10,  # shutdown mode
        'nop2': 0b11  # no operation
    }

    # Potentiometer selection (bits 0-1 of command byte)
    pot_select = {
        'none': 0b00,  # no operation
        'pot0': 0b01,  # wiper 0
        'pot1': 0b10,  # wiper 1
        'both': 0b11  # both wipers
    }

    command_byte = (command_bits[command] << 4) | pot_select[pots]

    # Data byte is the 8-bit wiper position
    data_byte = data & 0xFF

    return [command_byte, data_byte]
```

MCP42100 command sequence construction.

I/O Expanders

The MCP23S08 implements an SPI protocol with a 16-bit interface, including a *Control* byte and a *Register Address* byte. The client address contains five fixed bits, two user-defined hardware address bits (pins A1 and A0) and an additional read/write address bit (write mode = 0) to fill the control byte. The address pins allow connecting up to 4 components in the same bus, thus eliminating the necessity of two CS lines in the SPI bus by defining the address of

each component in the hardware development. The component includes 11 register addresses (Table 4.1), which defines the operations the component performs on the following I/O pins represented by each bit present in the byte [40].

Table 4.1 MCP23S08 I/O expander register addresses [40].

Address	Function
0x00	I/O Direction Register (IODIR)
0x01	Input Polarity Register (IPOL)
0x02	Interrupt-On-Change Control Register (GPINTEN)
0x03	Default Compare Register for Interrupt-On-Change (DEFVAL)
0x04	Interrupt Control Register (INTCON)
0x05	Configuration Register (IOCON)
0x06	Pull-Up Resistor Configuration Register (GPPU)
0x07	Interrupt Flag Register (INTF)
0x08	Interrupt Capture (INTCAP)
0x09	Port Register (GPIO)
0x0A	Output Latch Register (OLAT)

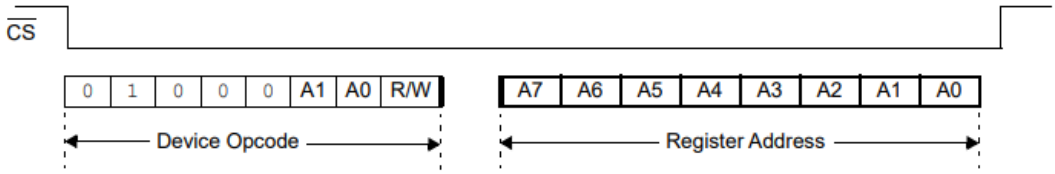


Figure 4.5 MCP23S08 16-bit SPI command format. *Device Opcode* is the *Control* byte [40].

The function `mcp23s08_command` consists of a `register_address`, `operation`, `device_addr` and `data` attributes to create the required 3 byte sequence: `device opcode`, `register address` and, at least one `data` byte. The first byte is developed by performing bitwise operations over a mask byte used to obtain the fixed 5-bit sequence. The second byte reflects the selected address from Table 4.1. For the third byte, the `pack_pins` function was developed to convert a list of pin numbers into active bit representations and return the associated byte value.

```
def mcp23s08_command(self, register_addr, operation='write', device_addr
    =0, data=None):

    # MCP23S08 register addresses
    REGISTERS = {
        'IODIR': 0x00,  # I/O Direction Register
        'IPOL': 0x01,  # Input Polarity Register
        (...)
    }

    # Build control byte: [01000][A1][A0][R/W]
    mask_byte = 0x40  # 01000000
    address_bits = (device_addr \& 0x03) << 1  # A1, A0 shifted to bits
       2-1
    operation_bit = 0 if operation == 'write' else 1
```

```
    control_byte = mask_byte | address_bits | operation_bit

    # Build register byte:
    if isinstance(register_addr, str):
        if register_addr.upper() in REGISTERS:
            reg_addr = REGISTERS[register_addr.upper()]

    # Build data byte:
    if data is None:
        data_byte = 0xFF
    else:
        if isinstance(data, list):
            data_byte = self.pack_pins(data)
        else:
            data_byte = data \& 0xFF  # Single byte

    return [control_byte, reg_addr, data_byte]
```

MCP23S08 command sequence construction.

The function `pack_pins(pin_list)` implements a loop `for` to cycle through each element in the list and set the associated bit.

```
def pack_pins(self, pin_list):

    result = 0
    for pin in pin_list:
        if 0 <= pin <= 7:
            result |= (1 << pin)
    return result
```

Contruction of the data byte reflecting the selected IO pins.

4.2.2 Signal Extraction from PhysioNet

Signal extraction from PhysioNet is implemented through the `load_ecg_data` function, which takes advantage of two WFDB package functions: `rdheader` and `rdrecord`. The first package function mentioned reads a WFDB header file and returns a 'Record' or 'MultiRecord' object with the record descriptors as attributes (record name, PhysioNet directory and multi-records flag). Since the software deals with single records, the function will be implemented solely with the name of the record and PhysioNet directory parameters. The objective of this implementation is to retrieve the sampling frequency of each file, update the software configurations, and consequently the required sampling rate in the simulator, and the signal length of the signal selected. The `rdrecord` function reads a WFDB record and returns the signal and record descriptors as attributes in a 'Record' or 'MultiRecord' object [37, 38]. The developed function takes `record_name`, `database`, `start_time`, `end_time` and `signal_number` attributes to create a simple and versatile acquisition function to select the main parameters associated with the signal: database, record, time range and channel, if more than one is present. The function pipeline begins updating the dictionary with the main configurations; afterwards, the signal acquisition is performed with the two mentioned WFDB package functions; finally, the signal is resampled and normalized to DAC values.

```
def load_ecg_data(self, record_name='100', database='mitdb/', start_time
    =0, end_time=10, signal_number=0):

     try:
         # Update system configurations
         record_name = self.current_config['record']
         database = self.current_config['database']
         start_time = self.current_config['start_sec']
         end_time = self.current_config['duration_sec']
         signal_number = self.current_config['channel']

         # Load header and update sampling frequency
         header = wfdb.rdheader(record_name, pn_dir=database)
         fs = int(header.fs)
         if fs != self.current_config['sampling_frequency']:
             self.current_config['sampling_frequency'] = fs

         # Obtain time range (sample range using saming rate)
         start_sample = int(start_time * self.current_config['
            sampling_frequency'])
         end_sample = int(end_time * self.current_config['
            sampling_frequency']) if end_time else header.sig_len

         # Retrieve Record object
         record = wfdb.rdrecord(
             record_name,
             pn_dir=database,
             sampfrom=start_sample,
             sampto=min(end_sample, header.sig_len),
             channels=[signal_number]
         )

         signal_data = record.p_signal.flatten()
         original_fs = self.current_config['sampling_frequency']

         # Resample the signal
         confirm = input("\nResample signal? (y/N): ").strip().lower()
         if confirm == 'y':
             if (self.current_config['resampling_frequency'] != 0 and
                     self.current_config['resampling_frequency'] != self.
                        current_config['sampling_frequency']):

                 original_time = np.linspace(0, len(signal_data) /
                    original_fs, len(signal_data))

                 duration = len(signal_data) / original_fs
                 target_samples = int(duration * self.current_config['
                    resampling_frequency'])
                 new_time = np.linspace(0, duration, target_samples)
                 signal_data = np.interp(new_time, original_time,
                    signal_data)

                 fs = self.current_config['resampling_frequency']
                 self.current_config['sampling_frequency'] = self.
                    current_config['resampling_frequency']

         # Normalize data to DAC
         normalized_signal = self.normalize_data(signal_data.tolist())
         if normalized_signal is None:
             return

         ecg_dac = int(normalized_signal * 65535)

         return ecg_dac
```

```
    except Exception as e:
        return None
```

Signal Extraction and Resampling function.

The resampling stage increases the sampling rate of the original signal and improves signal quality during generation using linear interpolation via NumPy's `interp` function. The extracted ECG signal values are then converted to DAC values using a custom `normalize_data` function, which normalizes the ECG values to a range of 0-1 and multiplies by 65535, representing the full scale of a 16-bit DAC.

```
def normalize_data(self, ecg_data):
    if not ecg_data:
        return None

    # Formula: (x-x_min) / (x_max - x_min)
    ecg_array = np.array(ecg_data)
    ecg_max = np.max(ecg_array)
    ecg_min = np.min(ecg_array)

    if ecg_max != ecg_min_value:
        normalized_data = (ecg_array - ecg_min) / (ecg_max - ecg_min)
    else:
        normalized_data = np.zeros_like(ecg_array)

    return normalized_data.tolist()
```

Signal normalization function.

4.2.3 Artificial Electrode-Skin Characterization

The contact interface between the skin and the electrode can be considered to act as a dielectric in the transduction mechanism and modeled by a contact capacitor. The electrode can be viewed as one of the capacitor plates and the *stratum corneum* the opposite plate [2]. The capacitance between the plates and the electrical resistance associated with the interface can be represented by

$$C = \frac{\varepsilon_0 \varepsilon_r A}{d} \tag{4.1}$$

where C is the capacitance, ε_0 is the permittivity of a vacuum, ε_r is the relative permittivity, A is the area of the plate and d is the separation between the plates, and

$$R = \rho \frac{L}{A} \tag{4.2}$$

where R is the resistance, ϱ is the resistivity of the material, L is the length and A is the cross-section area. The impact of pour electrode contact is expressed by Equations 4.1 and 4.2, resulting in a significant increase in resistance and a decrease in capacitance.

In the interface, the parameters described in Equations 4.1 and 4.2 are modeled using `SignalDesigner` class. Despite the fact that capacitance and resistance commonly depend on surface area, the computer interface enables users to individually define custom impedance

curves over time by selecting both parameters. The interface (Figure 4.6) is created using *MatplotLib* package: it presents a horizontal time axis and two vertical resistance and capacitance axis; three buttons to switch between the capacitance/resistance curve design, extract data and exit the interface; inherent with the package, it also provides enlarging, minimizing and dragging tools, and the time-scale value changes with the selected time frame.

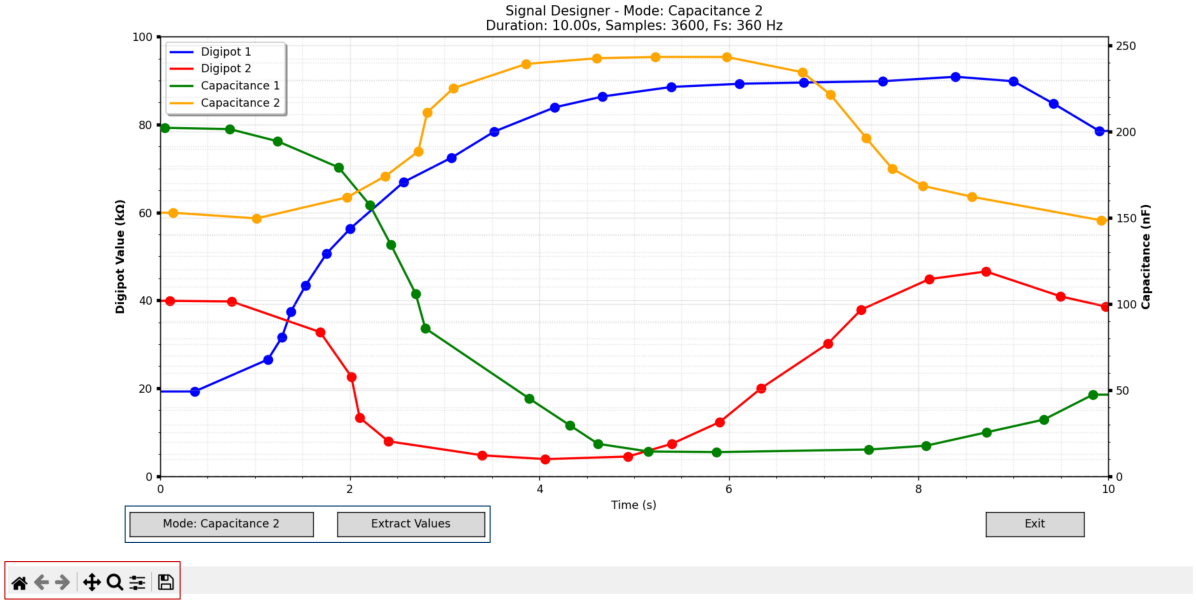


Figure 4.6 `SignalDesigner` class interface. The tools inside the red box change the view of the plot. The right and left axis represent, respectively, the capacitance and resistance. The buttons in the blue box allow the selection and extraction of values.

The artificial impedance design tool is presented as an option for signal generation to the user using `input` function. `design_signals_gui` receives the number of samples in the signal and the sampling frequency, both of which can be used to obtain the duration of the signal. The `SignalDesigner` launches using `get_user_signal_values`, admitting the sampling frequency and total samples to create the plot, and creating a SignalDesigner object.

```
design_now = input(f"Artificial Impedance design? (y/N): ").strip().
    lower()
        if design_now == 'y':
            self.design_signals_gui(
                self.current_config['sampling_frequency'],
                self.current_config['samples']
            )

(...)

def design_signals_gui(self, samplingFreq=None, total_samples=None):

    # Select default sampling frequency
    if samplingFreq is None:
        samplingFreq = self.current_config['sampling_frequency']

    # Select default number of samples
    if total_samples is None:
        if hasattr(self, 'current_config') and 'samples' in self.
```

```
            current_config:
                total_samples = self.current_config['samples']
            else:
                # Calculate from duration
                duration = self.current_config['duration_sec'] - self.
                    current_config['start_sec']
                total_samples = int(duration * samplingFreq)

        # Display artificial impedance GUI
        self.designed_signals = self.get_user_signal_values(
            total_samples=total_samples,
            sampling_rate=samplingFreq
        )

(...)

def get_user_signal_values(self, total_samples, sampling_rate):

    # Create figure
    fig = plt.figure("Signal Designer", figsize=(10, 6))
    designer = SignalDesigner(total_samples, sampling_rate, fig)
    plt.show(block=True)  # Show figure

    final_data = designer.extract_and_normalize()

    designer.cleanup()  # Destroy figure
    return final_data

(...)

def get_designed_signal_values(self):

    # Extract the normalized arrays (already 0-255)
    signals = self.designed_signals['signals']

    digipot1_values = signals['digipot1']['normalized']
    digipot2_values = signals['digipot2']['normalized']
    io_exp1_values = signals['capacitance1']['normalized']
    io_exp2_values = signals['capacitance2']['normalized']

    # Associate capacitance to I/O pins
    io_exp1_pins = self.capacitance_to_pins(io_exp1_values)
    io_exp2_pins = self.capacitance_to_pins(io_exp2_values)

    return (digipot1_values.tolist(),
            digipot2_values.tolist(),
            io_exp1_pins,
            io_exp2_pins)
```

Sequence of events associated with the extraction of resistance and capacitance values in GUI.

The digital potentiometer values must be normalized and scaled to the 0-255 range to fit the associated 8-bit data package. This process is performed before extracting data for use in the `ecgSystem` class. The capacitance values do not require normalization because the communication protocol with the component receives information about the I/O pins. The function `get_designed_signal_values` retrieves digital potentiometer values and capacitance values generated through specific pin combinations. The function `capacitance_to_pins` is used to associate the selected capacitance with I/O pins, connected with capacitors with specific values, over time. The function sorts and associates capacitance values with hardware pins. The

function iterates over every capacitance value present in the input list, resulting in two cases: if the capacitance is 0, the resulting pin list is empty; if the capacitance is different than 0, the pins representing the value are obtained by sequentially iterating and searching for the largest value that fits the capacitance, and retrieving the associated pin number. The result is multiple lists of pin numbers inside a main list.

```
def capacitance_to_pins(self, capacitance_values):

    # Pair each capacitor with a I/O pin -> (pin, cap_value)
    # Sort the tuple from high to low value 'cap_value'
    capacitor_values = [1, 2, 4, 8, 16, 33, 64, 128]
    sorted_caps = sorted(enumerate(capacitor_values), key=lambda x: x
       [1], reverse=True)
    pin_combinations = []

    # Loop through each target value from the input list.
    for cap_val in capacitance_values:
        selected_pins = []
        if cap_val == 0:
            pin_combinations.append([])
            continue

        # Loop to subtract each capacitor value and retrieve a pin
        for pin, value in sorted_caps:
            if value <= cap_val:
                selected_pins.append(pin)
                cap_val -= value
                if cap_val == 0:
                    break

        all_pin_combinations.append(sorted(selected_pins))

    return all_pin_combinations
```

Conversion of capacitance to IO pin numbers (Greedy Process).

The resulting designer lists are called in `send_stream` to develop multiple data packages containing the information of the sample and the artificial impedance parameters.

4.2.4 Data Packing

The simulation system communicates with the computer via a binary protocol to improve transmission efficiency, reducing data latency, and easing parsing by the simulation system. The data packing is processed through 3 main stages, presented in Figure 4.7.

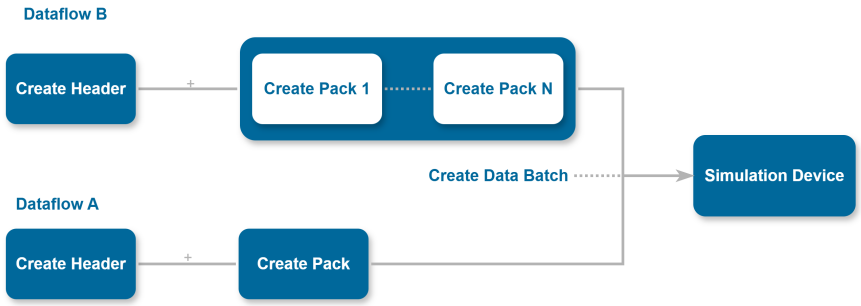


Figure 4.7 Data packing process reflecting Figure 4.1. In *Dataflow A*, the interface creates an individual pack to change configurations within the MCU or, alternatively, sends solely the header with specific commands. In *Dataflow B*, multiple data packs are created containing the ECG data associated with a specific signal.

The function `create_stream_header` creates the header for the data to be transmitted. It receives the 'COMMAND' to execute and the number of packets that follow in the transmission, if any. The header consists of 7 bytes: 2 start and end markers, 2 bytes for the number of packets to be sent, and 1 byte for the command recognized by the MCU.

```
def create_stream_header(self, command, n_packets):
    """| start(2) | n_packets(2) | command(1) | end(2) |"""
    header = bytearray()
    header.extend(struct.pack('<H', 0xBB66))  # Stream start marker
    header.extend(struct.pack('<H', n_packets))  # Number of packets
    header.append(command)  # Command
    header.extend(struct.pack('<H', 0x66BB))  # Stream end marker
    return bytes(header)
```

Contruction of the byte sequence of the header associated with each batch.

The function `create_data_packet` is used to organize each ECG sample, and both capacitance and resistance values. The ECG values and artificial skin-electrode parameters are processed individually: the ECG sample is converted bytes; afterwards, the functions `mcp42100_command` and `mcp23s08_command` are implemented to create the specific command sequence to control the components. Finally, all five parameters are packed using a byte array.

```
def create_data_packet(self, ecg_sample_mv, digipot1_value=0,
    digipot2_value=128,
                       io_exp1_pins=None, io_exp2_pins=None):

    # Convert ECG sample to DAC byte
    ecg_dac = int(ecg_sample_mv * 65535)
    ecg_bytes = struct.pack('<H', ecg_dac)

    # Generate digital potentiometer commands
    digipot1_value = self.mcp42100_command(command='write', pots='pot0',
        data=digipot1_value)
    digipot2_value = self.mcp42100_command(command='write', pots='pot1',
        data=digipot2_value)

    # Generate I/O expander commands
    io_exp1_value = self.mcp23s08_commmand('OLAT', device_addr=0, data=
       io_exp1_pins)
    io_exp2_value = self.mcp23s08_commmand('OLAT', device_addr=1, data=
       io_exp2_pins)
```

```
    packet = bytearray()
    packet.extend(ecg_bytes)
    packet.append(digipot1_value[1])
    packet.append(digipot2_value[1])
    packet.append(io_exp1_value[2])
    packet.append(io_exp2_value[2])

    return bytes(packet)
```

Organization of a single pack.

The resulting 6 bytes packet contains an ECG sample and commands associated with a digital potentiometer and I/O expanders. Subsequently, the function `create_stream_packet` builds a larger pack, comprising of a header and *N* smaller packs.

```
def create_stream_packet(self, command, ecg_data_mv, digipot1, digipot2,
    io_exp1, io_exp2):

    # Large packet with simple command
    if not ecg_data_mv:
        return self.create_stream_header(command, 0)

    # Large packet:   Header | Packet 1 | (...) | Packet N
    n_packets = len(ecg_data_mv)
    stream_data = bytearray()

    # Add stream header
    header = self.create_stream_header(command, n_packets)
    stream_data.extend(header)

    # Add ECG data packets with RAW mV values
    for i in range(n_packets):
        data_packet = self.create_data_packet(
            ecg_data_mv[i],
            digipot1_value=digipot1[i],
            digipot2_value=digipot2[i],
            io_exp1_pins=io_exp1[i],
            io_exp2_pins=io_exp2[i]
        )
        stream_data.extend(data_packet)

    return bytes(stream_data)
```

Structuring a transmission batch.

The `send_stream` function encompasses all mentioned functions, segmenting the retrieved data from the database, in a specified value, if any, and packing with the specified 'COMMAND'. The multiple bigger packets, organized by `create_stream_packet` are send in stream.

```
def send_stream(self, command, ecg_data, chunk_size=200):

    # Retrieve impedance parameters
    digipot1_values, digipot2_values, io_exp1_pins, io_exp2_pins = self.
       get_designed_signal_values()

    total_samples = len(ecg_data)

    # Segment data in chunks
    for i in range(0, total_samples, chunk_size):
        chunk_ecg = ecg_data[i:i + chunk_size]
        chunk_dig1 = digipot1_values[i:i + chunk_size]
```

```
chunk_dig2 = digipot2_values[i:i + chunk_size]
chunk_io1 = io_exp1_pins[i:i + chunk_size]
chunk_io2 = io_exp2_pins[i:i + chunk_size]

chunk_num = (i // chunk_size) + 1
total_chunks = (total_samples + chunk_size - 1) // chunk_size

# Create big packets with header + N smaller packs
stream_packet = self.create_stream_packet(command, chunk_ecg,
    chunk_dig1, chunk_dig2, chunk_io1, chunk_io2)

# Send the packet
if stream_packet:
    bytes_sent = self.ser.write(stream_packet)

    self.read_response()
    time.sleep(0.5)
```

Data segmentation for transmission.

Alternatively, the interface packing system, shown in Figure 4.7, presents *Dataflow A* to transmit single commands to the microcontroller, such as changing signal generation configurations or performing SD card operations.

4.3 Embedded System Implementation

The implemented system serves as a platform for on-demand signal generation by providing a full repository of ECG records and waveforms that can be accessed, integrating a signal processing and system control back-end, designed to generate multiple ECG waveforms to non-clinical ECG acquisition devices still in development.

The system includes some features usually absent in certified testing equipment, such as an SD card slot for simulation files or artificial impedance simulation, making it more versatile in performing tests and setting parameters, as well as in exploring the different electrode-skin interface impedances throughout the ECG waveform simulation.

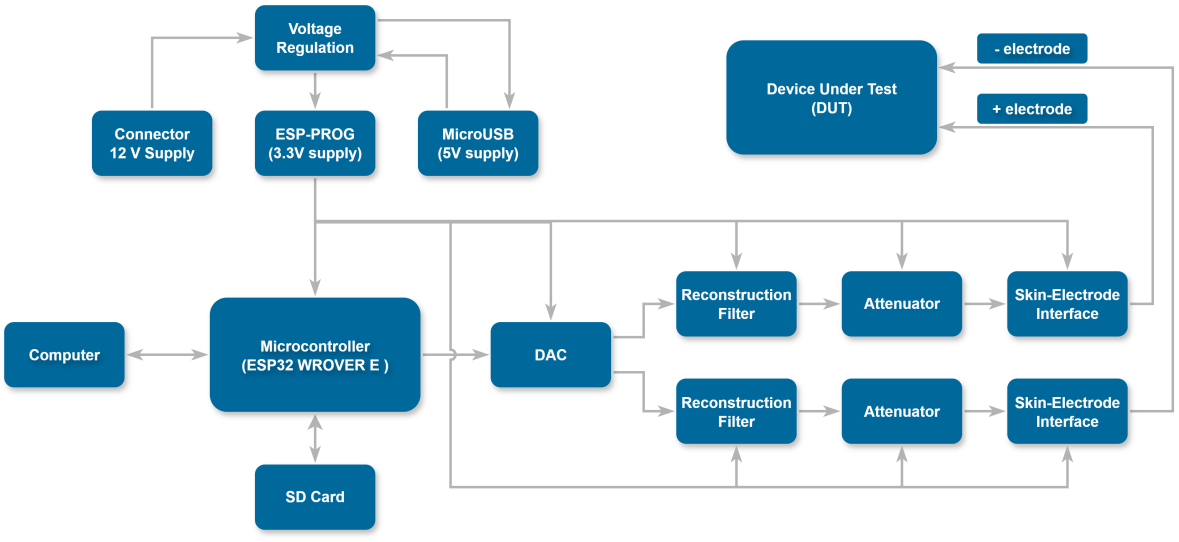


Figure 4.8 Block diagram of the hardware implementation of the system.

The hardware block diagram is presented in Figure 4.8. The **Computer** is the starting point of the system by selecting the intended options and signals, and expresses a two-way path with the MCU. The system can be supplied using a 12 V source using a battery, a 5 V supply from the microUSB connector or a 3.3 V connection from ESP-PROG, or JTAG (Joint Test Action Group), connector. The last option was used throughout the whole development process. The **SD Card** manifests a two-way connection with the **Microcontroller** due the the writing and reading operations performed. The **Microcontroller** transfers organized data to **DAC** block to create a stepped waveform which, in turn, is smoothed in the **Reconstruction Filter** block. The **Attenuator** block is necessary to provide low amplitude signals required at the input of the ECG front-end. The **Skin-Electrode Interface** circuit is digitally controlled by the **Microcontroller**, allowing the user to define individual impedance profiles of the electrodes for each signal sent and stored in the **SD Card**. The two resulting wires create a single-lead ECG signal generation with individually controlled circuits.

4.3.1 Power Supply

The proposed device features three different voltage sources. The 12 V segment implements a Molex LLC 43650-0200 connector to insert the power source and a protection circuit created by a 500 mA fast blow fuse and a standard recovery S2D-13-F diode to prevent reverse current flow. The 12 V output path is connected to a 5 V 500 mA BD450M5FP-CE2 LDO voltage regulator with a wide operational input range of 3 - 42 V [41]. The exit of the 5 V voltage source is connected to the 3.3 V regulation circuit and the micro USB port. The 3.3 V section has an (single-pole single-throw) SPST switch to control the current flow from the micro USB port and hosts an AP7343-33W5-7 LDO voltage regulator with an operational range of 1.7 - 5.25 V and an output voltage of 3.3 V 300 mA [42], while an LED indicates that the device is powered. Along the power pipeline, the three voltage sources contain decoupling capacitors and are separated with two solder bridges.

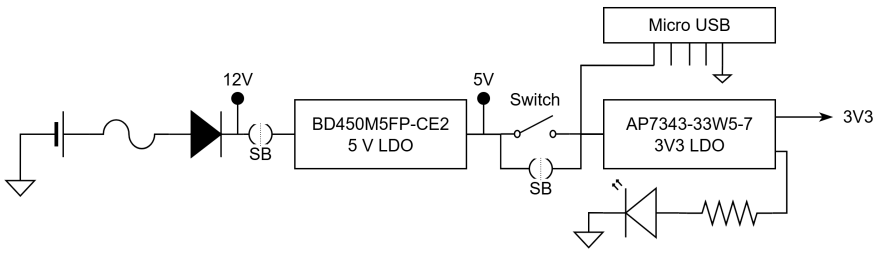


Figure 4.9 Power supply implementation. Solder bridges (SB) provide a physical separation between voltage sources.

The ESP32-WROVER-E has a peak current consumption of 350 mA in active mode when using radio frequency peripherals such as Wi-Fi, Bluetooth, or BLE. Since these features were not implemented in the system, the maximum current consumption of the MCU remains below 113 mA. The supply currents for the other main components are presented as follows [43].

- Digital potentiometer: 1 mA
- I/O expander: 1 mA $\times$ 2
- Digital-to-Analog converter: 480 μA
- SD card: up to 100 mA during high-speed writes [44]

4.3.2 Microcontroller Unit

The chosen microcontroller is an Espressif Systems Co., Ltd. ESP32-WROVER-E microcontroller, featuring an Xtensa dual-core 32-bit LX6 microprocessor capable of running up to 240 MHz and ideal for embedded systems. The MCU also features major peripherals associated with embedded systems, such as WiFi capabilities up to 150 Mbps and Bluetooth v4.2 (classic) or BLE. The memory includes Read-Only Memory (ROM) and Static Random Access Memory (SRAM) of 448 KB and 520 KB, respectively, optimal for large data operations and/or executing multiple operations at the same time or within a short window of time. It presents an operating voltage of $3.0 \sim 3.6$ V and up to 24 General Purpose Input-Output (GPIO) pins, including 5 strapping pins (GPIO0, GPIO2, MTDI, MTDO and GPIO5) that determine the chip boot flow and initial flow, that can be used to implement different communication protocols, access to integrated or external peripherals, change the system behavior dependent on the GPIO state or signal an external component connected to the GPIO [43].

Programming the Microcontroller

The MCU is programmed using a computer running any compatible Integrated Development Environment (IDE). The current work used Arduino IDE due to the ease of use, already installation, and being an open source software. In advantage, the IDE integrates multiple examples for different libraries, making it practical for an inexperienced programmer, as well as an online community of users that provided causes and solutions for code errors and labels occurring during the development of the code. Other online community websites were consulted, such as StackOverflow, ESP32 Forum, PlatformIO Community or Github Forum.

The prototype board offers two connectors and requires an external device (programmer) to flash and download the new firmware - *ESP-PROG* board. ESP-PROG (Figure 4.10) is an Espressif Systems development and debugging tool, with functions that include automatic firmware downloading, serial communication, and JTAG online debugging [45]. The user can connect the programmer to the computer via USB and execute operations after connecting either JTAG or PROG connector to the board. The PROG connection was used throughout the development of the code in the prototype, the required debugging steps were performed using flags, and the presented errors were emended. The main advantage of the JTAG connection is its advanced debugging and testing capabilities [45].

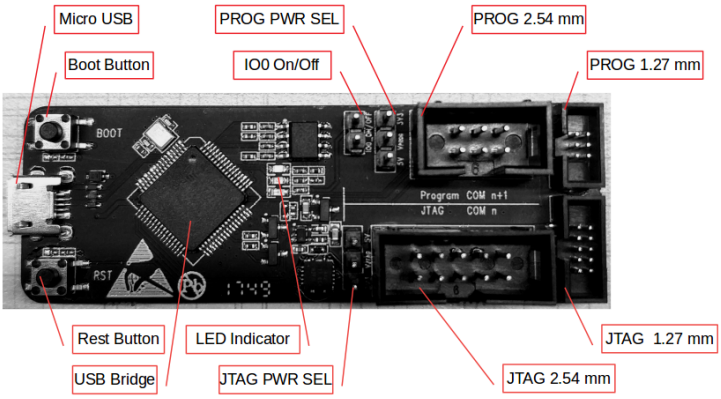


Figure 4.10 Front view of ESP-PROG. From [45].

When uploading a new firmware, the IDE compiles the code into structured binary data and sends it to the MCU via ESP-PROG, where the same is stored in flash memory. After a successful firmware upload, each time a normal reset is triggered and the strapping pins present default values, the MCU will initiate, perform a copy of the firmware stored in flash memory to RAM and execute the program at the entry point in RAM. Therefore, the device can be turned off or reset without repeating programming the MCU [43].

Real-Time Operating Systems & Multi-Threading

In embedded systems, the development of an application running in continuous loop is not ideal. A Real-Time Operating System (RTOS) is a particular form of developing an application using tasks/threads. The main features associated with a task can be considered its priority (reflecting its importance), the deterministic behavior (knowing when, how and why the task begins and terminates) and the resources allocated to the task. Real-time properties require the system to respond to a certain event within a *deadline*. A part of the operating system called the scheduler is responsible for deciding which program to run providing the illusion of simultaneous execution by rapidly switching tasks [46].

FreeRTOS is an open-source class of RTOS designed to be small enough to run on a microcontroller. The FreeRTOS scheduler achieves deterministic behavior by allowing the user to assign a priority to each thread of execution. Each should task execute within its own context without dependency on other tasks in the application or the scheduler itself, and should be simple, without use restrictions, prioritized and support full preemption and scheduler re-entrance. Therefore, FreeRTOS provides the core real-time scheduling functionality, inter-task communication, timing and synchronization primitives. Two mechanisms implemented for inter-task communication are queues and ring buffers. Synchronization primitives are key objects for task synchronization and protection of shared resources, and include binary semaphores, mutual exclusion (MUTEX) semaphores, and spinlocks [46].

The programming of RTOS tasks should follow the structure below, consisting of an endless `for` loop and an exit/deletion section. The task is created using `xTaskCreate()`, or related functions, by defining its properties: name, function, memory to allocate, variables, priority or

handle. In the developed code, `xTaskCreatePinnedToCore()` was used, which is an ESP-IDF extension of the FreeRTOS function that enables task creation with specified core affinity (0 or 1) or operation on both cores (`tskNO_AFFINITY`) [46, 47].

```
TaskHandle_t taskHandle; // reference to the task (handle)

void taskFunction( void *pvParameters ) {

    for(;;) {
        -- Task application code here. --
    }
}

/* Task Creation w/ affinity */
BaseType_t xTaskCreatePinnedToCore(
         TaskFunction_t pvTaskCode, // function
         const char * const pcName, // name
         const configSTACK_DEPTH_TYPE uxStackDepth, // memory
         void *pvParameters, // passed parameters
         UBaseType_t uxPriority, // priority
         TaskHandle_t *pvCreatedTask, // handle
         const BaseType_t xCoreID // core affinity
       );
```

Typical Task contruction with core affinity (ESP-IDF).

Device Operation & Task Management

To start operating the device, the SD card should be inserted. Alternatively, if the SD card is inserted after the device is already running, the device should be reset to recognize the card and check file operations. Afterwards, the device should be connected to the computer interface via ESP-PROG, or a similar controller with a USB-UART bridge and, if possible, debugging capabilities, where the user can perform operations and manage the overall system. The MCU is responsible for all tasks on the device, receiving and processing information from the computer interface, and managing SPI bus components and remaining peripherals.

The design and architecture of the system can be presented using Unified Modeling Language (UML) diagrams, a general-purpose, object-oriented, and visual approach of representing the structure, behavior, and interactions in the system [48]. The structure of the device was organized using a UML Component diagram in Figure 4.11. The microcontroller firmware consists of four tasks, namely, `PacketParserTask`, `SDWriteTask`, `outputBufferAddTask` and `outputBufferGetTask`, and synchronization primitives.

`PacketParserTask` receives and interprets binary serial data, assembles data packets, and executes corresponding actions based on the commands incoming. The header of the stream package is decoded with `processStreamHeader`, a function that acts as the first-line validation area and routes commands without data sections in the stream package to the appropriate functions. The function parses incoming serial binary data using switch...case statements, where each case represents a specific header section, advancing upon successful completion and terminating in the 'PARSE_STREAM_END' case. The final case associated with header parsing will either reset the parser for additional incoming data or advance to parsing the data sections identified by a specific `command` (Figure 4.12). The data section of

the stream is retrieved and processed in an additional case statement within the parser using `processStreamData`, where each data packet is processed and sent to a queue for `SDWriteTask` (Figure 4.13).

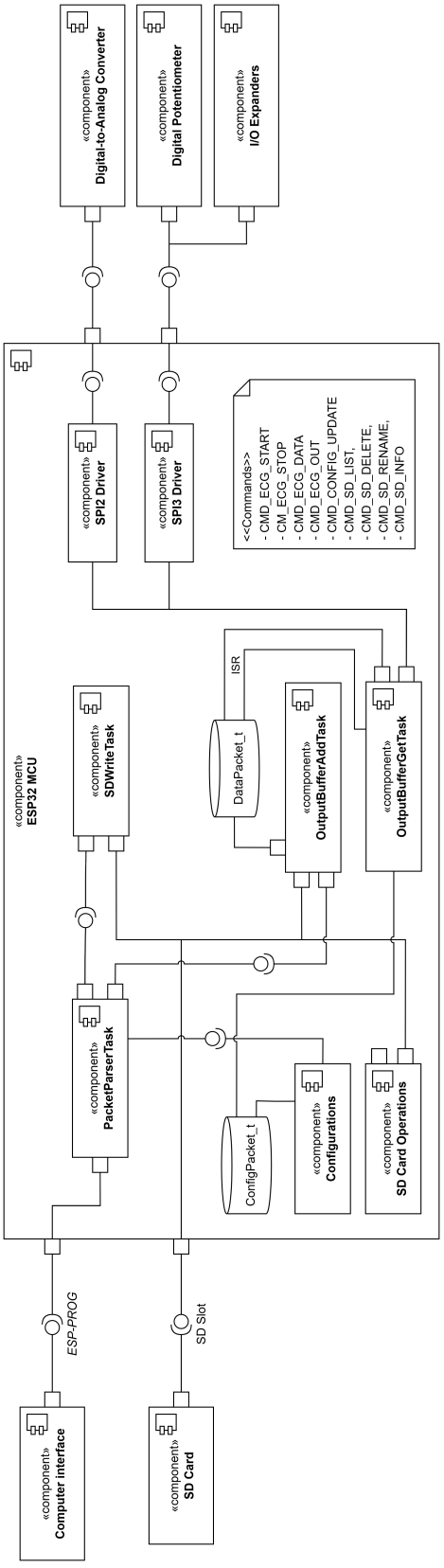


Figure 4.11 UML Component diagram of the system.

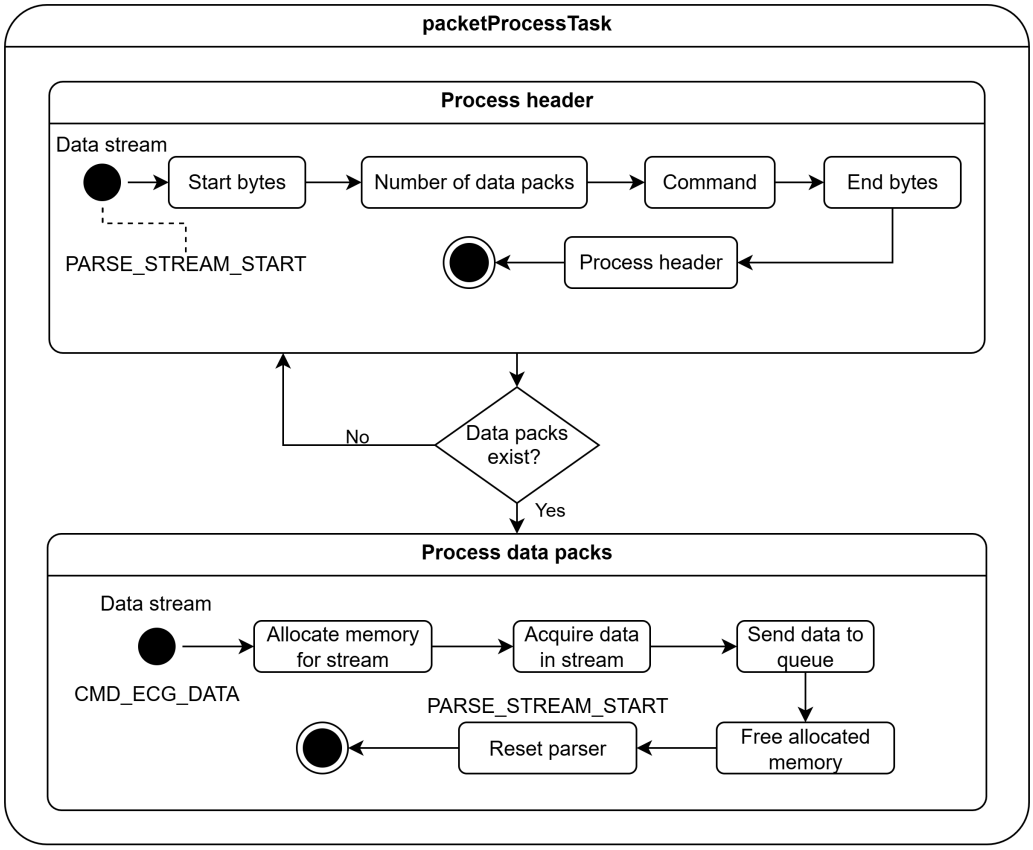


Figure 4.12 UML Behavior diagram of `packetParserTask`.

The signals are stored on the SD card via `SDWriteTask`. The task is structured in three sections: creation of a .BIN file with the name previously appointed in the software interface; main `for` loop to receive and write data in the memory card; task deletion section. The file is created with the specified `filename` and write/update mode for binary files `wb+` allowing input and output operations. A 16 KB buffer in *full buffering* mode is established to facilitate data storage on the micro SD card. Data packets retrieved from the queue are written to the SD card after acquiring the associated control semaphore. The buffered values are saved on the SD card by flushing the data, and subsequently the associated semaphore is restored. Once all signal data has been received, the task proceeds to the cleanup process: it performs a final buffer flush, closes the file, and eliminates both the task and its handle.

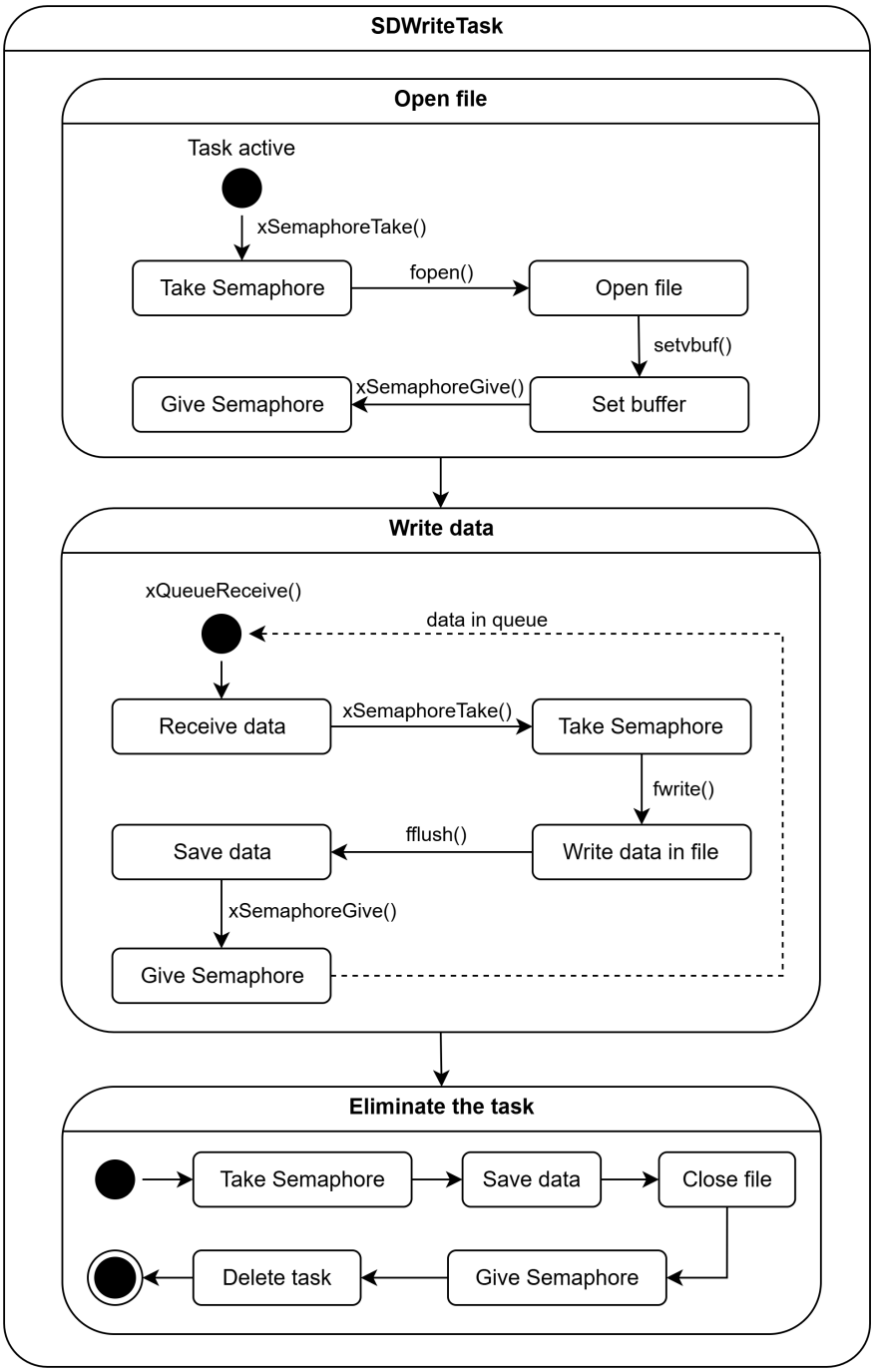


Figure 4.13 UML Behavior diagram of `SDWriteTask`.

The generation of signals stored on the card is performed with two tasks `outputBufferAddTask` and `outputBufferGetTask` that create a producer-consumer system in a shared ring buffer. The first task opens and the file, retrieves data in large blocks and pre-fills the buffer, after which the timer starts. The second task is mainly performed in Interrupt Service Routines (ISR) and retrieves a value every sampling period. The sampling frequency is guaranteed by the *General Purpose Timer* event callbacks and can be changed using the computer interface. With every ISR event, both DAC channels, the Wiper terminals and I/O expanders are updated.

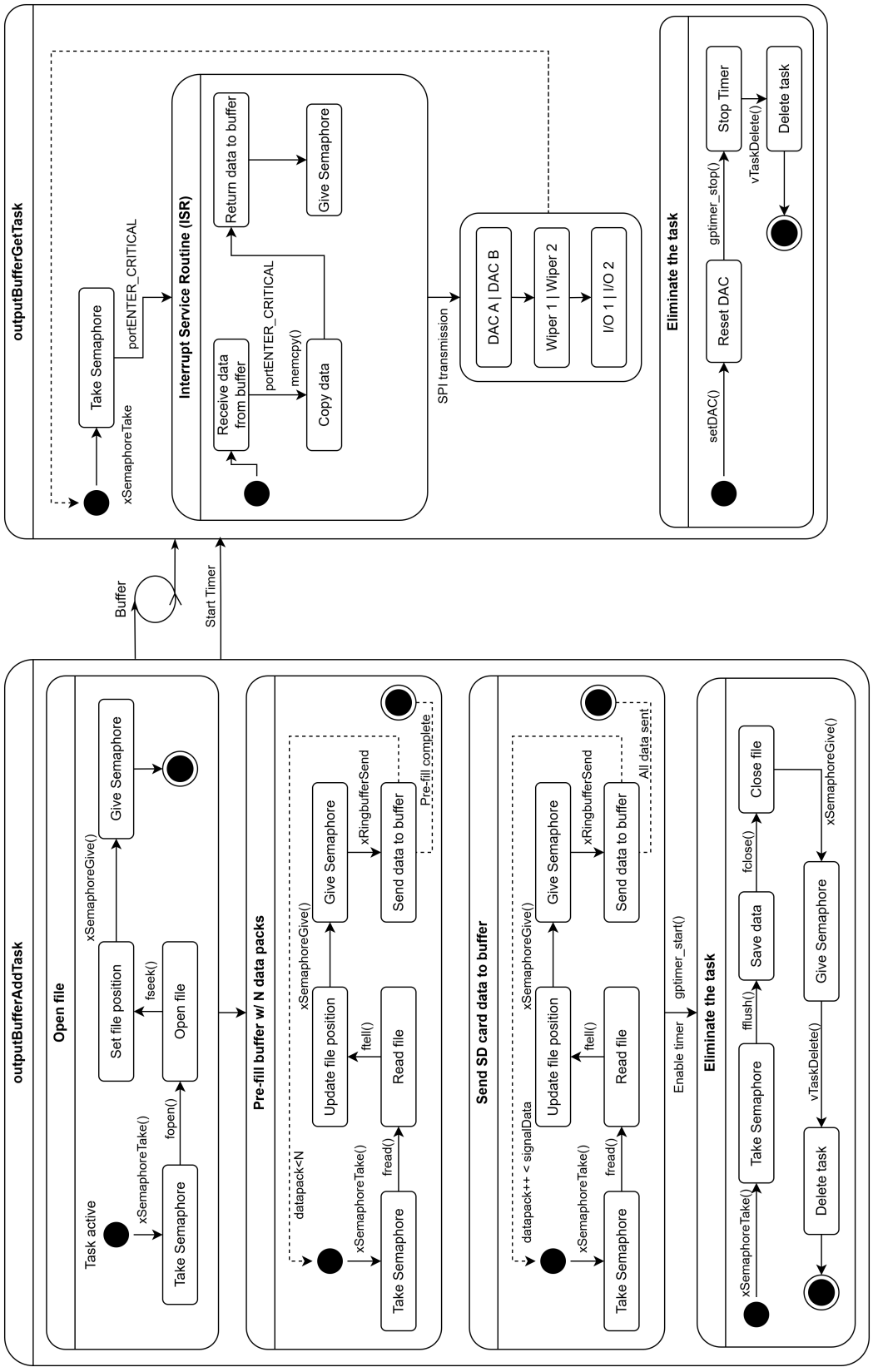


Figure 4.14 UML Behavior diagram of `outputBufferAddTask` and `outputBufferGetTask`.

4.3.3 SPI Communication Bus & Peripherals

The SPI is a synchronous communication protocol used for short-range communication, particularly in embedded systems, and mainly characterized by its master-slave architecture, where a master device controls one or more slave devices. This communication protocol is advantageous for its high-speed, simultaneous data transmission and reception across the connected slaves, flexibility and can consist of four signal lines [49]:

- **Serial Clock (SCLK)** is generated by the master to synchronize data transfer;
- **Master-Out Slave-In (MOSI)** line is used to transfer data to the slave(s);
- **Master In Slave Out (MISO)** line is used to transfer data to the master device;
- **Chip Select (CS)** is managed by the SPI controller to select the device to communicate.

The data transfer that occurs with master and slave is associated with different protocol implementations. A full-duplex design incorporates all mentioned lines, thus creating a transmission and reception bus. An half-duplex is comprised of CS and SCLK lines, and a data transmission line (MOSI or MISO). A three-line half-duplex joins MOSI and MISO into a single transmission/reception wire [49, 50].

The MCU integrates four SPI protocol controllers: controller SPI0 is used as a buffer for accessing external memory; controller SPI1 can be used as a master; controllers SPI2 and SPI3 can be configured as either a master or a slave [43]. The native SPI protocol provides multiple useful properties that can be individually configurable between buses [50]:

- Programmable data transfer length, in multiples of 1 byte;
- Full-duplex, half-duplex and three-line half-duplex communication support;
- Master and slave modes
- Programmable polarity (CPOL), phase (CPHA) and clock;

The last mentioned property allow the selection of four different SPI modes due to the clock polarity and phase configurations. Each SPI controller is associated with a single mode. Figure 4.15 shows the differences between all four SPI modes. The developed simulator implements two different modes: SPI0 and SPI1.

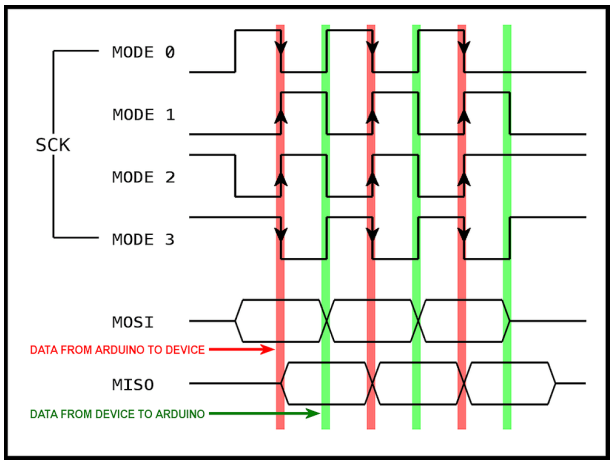


Figure 4.15 Phase and Polarity changes in SPI modes. SPI0 and SPI1 have equal polarity and different phase. SPI0 and SPI3 have different polarity and equal phase. From [51].

The diagram in Figure 4.16 provides a bigger picture of the communication system within the simulator. SPI2 (HSPI) controller manages the DAC, and the SPI3 (VSPI) controller manages the remaining peripherals: digital potentiometer; I/O expanders; SD card.

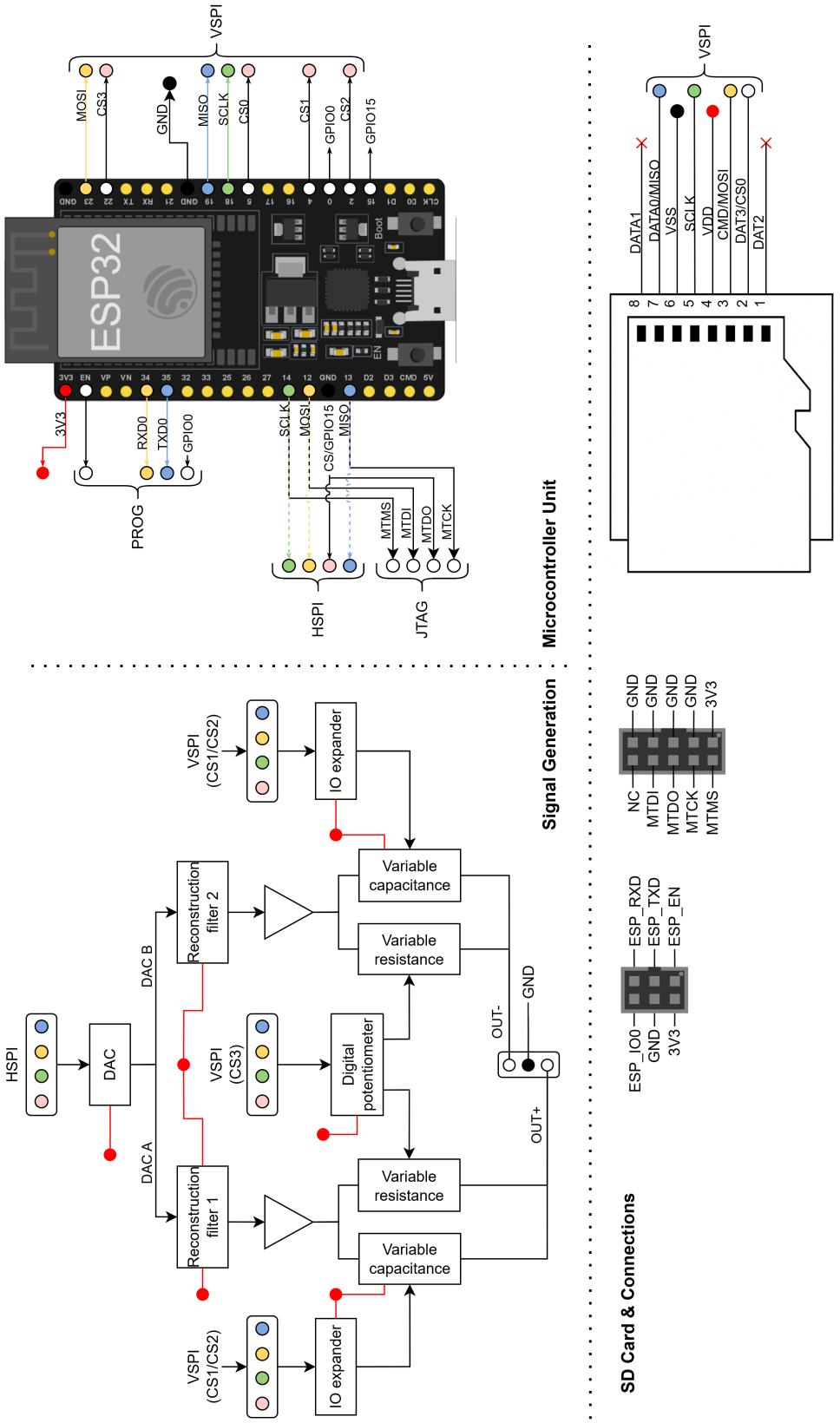


Figure 4.16 Hardware architecture diagram. The signal generation circuit is on the top-left side. The MCU and connections are on the right-top side. The ESP-PROG and JTAG connectors, and SD card are in the bottom-middle position.

The interface was developed implementing two SPI master drivers. The data transfer can occur using interrupt or polling transactions and is defined at the moment of initiation of the driver; however, the driver can send 'mixed' transactions by changing the transaction type alternatively and, recommended, within a single task [50].

Interrupt transactions block the transaction routine until it completes, allowing the MCU to run other tasks. A task can queue multiple transactions from different sources, and the driver automatically handles them one by one in the ISR, thus preventing the waste of MCU cycles [50]. Alternatively, polling transactions avoid interrupts by having the routine maintain data transfer until finished, saving time on queue handling to achieve a smaller transaction duration, but at the cost of keeping the MCU continuously busy [50].

To improve data transfer for one device, the SPI driver provides a function to acquire the bus, suspending transactions from other devices sharing the same controller until the bus is released. In addition, the devices or the driver can be removed from the bus.

The SPI driver usage follows the pipeline in Figure 4.17. `spi_bus_config_t` structure sets the parameters for each SPI bus in `spi_bus_initialize`. Afterwards, devices can be registered to the SPI bus with `spi_bus_add_device`; multiple transaction structures can be defined using `spi_transaction_t` to attend the required parameters to send. Finally, both SPI drivers call `spi_device_polling_transmit` to execute data transfers using polling transactions.

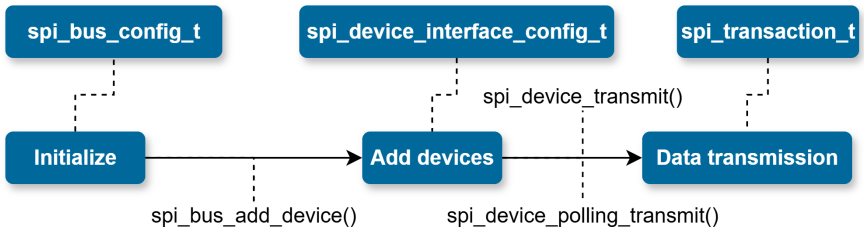


Figure 4.17 SPI driver usage diagram.

Direct Memory Access (DMA) on ESP32 allows high-speed data transmissions between peripherals and can be used in multiple communication protocols, such as SPI, I2C or UART without intervention of the MCU, thus contributing to multitasking. In both drivers, the DMA channels are automatically selected by the driver ensuring its use and reducing MCU load [50]. In the proposed device, both SPI drivers are initialized with `spiStart` function. The devices are added in the following chapters.

```
esp_err_t spiStart() {
    esp_err_t ret;
    // VSPI Configuration
    spi_bus_config_t vspi_bus_config = {
        .mosi_io_num = GPIO_NUM_23,
        .miso_io_num = GPIO_NUM_19,
        .sclk_io_num = GPIO_NUM_18,
        .quadwp_io_num = -1,
        .quadhd_io_num = -1,
        .max_transfer_sz = 4096,
    };
    // HSPI Configuration half duplex (DAC)
```

```
    spi_bus_config_t hspi_bus_config = {
        .mosi_io_num = GPIO_NUM_13,
        .miso_io_num = -1,
        .sclk_io_num = GPIO_NUM_14,
        .quadwp_io_num = -1,
        .quadhd_io_num = -1,
        .max_transfer_sz = 4096,
    };

    // initialize VSPI bus (SPI3)
    ret = spi_bus_initialize(VSPI_HOST, &vspi_bus_config,
       SPI_DMA_CH_AUTO);
    if (ret != ESP_OK) {
        Serial.printf("Failed to initialize VSPI bus: %s\n",
           esp_err_to_name(ret));
        return ret;
    }
    // initialize VSPI bus (SPI2)
    ret = spi_bus_initialize(HSPI_HOST, &hspi_bus_config,
       SPI_DMA_CH_AUTO);
    if (ret != ESP_OK) {
        Serial.printf("Failed to initialize HSPI bus: %s\n",
           esp_err_to_name(ret));
        return ret;
    }
    return ret;
}
```

Initialization of SPI drivers and bus configurations.

Digital to Analog Conversion

The DAC8552IDGKT, manufactured by Texas Instruments Incorporated, was used to generate stepped signal waveforms through two channels with 16-bit precision, where data is sent to the component using a dedicated SPI bus. The properties that make this component appropriate for the simulator include low-voltage operation, excellent resolution, an internal reference voltage, two output channels (ideal for single-lead ECG), and SPI communication interface [52].

The DAC uses a 3-wire serial interface ($\overline{\text{SYNC}}$, SCLK and D_{IN}) compatible with SPI protocol ($\overline{\text{CS}}$, SCLK and MOSI). The component accepts a 24-bit sequence composed of 8 control bits and 16 data bits, read from the most to the least significant bit. The control sequence manages the DAC channels, buffer usage and impedance inherent to each DAC channel, allowing to simultaneously load both channels and generate a signal. The data sequence accepts up to 65536 different values (between 0 and 65535) used to generate stepped waveforms; the ideal output voltage is given by Equation 4.3 [52]. Similarly to supply, the implemented reference voltage V_{REF} was 3.3 V.

$$V_{OUT}A, B = V_{REF} \times \frac{D}{65535} \quad [V], \quad V_{REF} = 3.3 \quad [V] \tag{4.3}$$

The DAC8552 SPI interface is capable of operating at a baudrate up to 30 MHz for 5 V [52]. Figure 4.18 shows the synchronization of the three signal lines. Data is periodically sent to the DAC registers with data bits synchronized with the clock rising edge. The control bits are sent, followed by the data bits, and both sequences are read from the most significant to the

least significant bit.

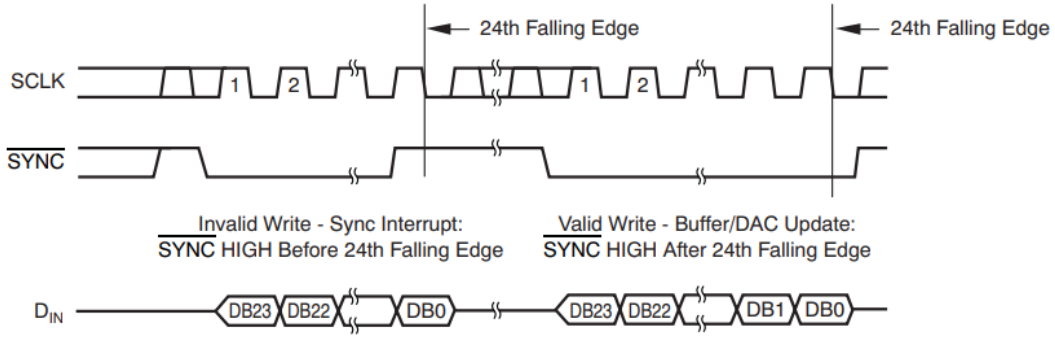


Figure 4.18 DAC8552 SPI interface timing diagram [52].

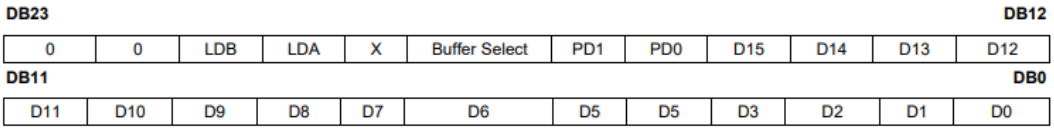


Figure 4.19 DAC8552 Data Input Register Format [52].

The 24-bit sequence should follow the format of Figure 4.19 for proper functioning. The first two bits are reserved and must at low state. The LDB and LDA bits control the updating of each analog output with the specified 16-bit data value or Power-Down command. Bit 19 is irrelevant for the operation of the component. Bit 18 (Buffer Select) defines the destination for the 16-bit data sequence or Power-Down command, defined in the 17 and 16 bits. In normal operation, both bits are set to a low state, while the remaining bit combinations switch between impedance modes [52].

The operations of the DAC start with its inclusion in HSPI bus, after the same has been initiated previously, through `dacStart` function.

```
#define DAC_CS    GPIO_NUM_15
spi_device_handle_t dacHandle;

esp_err_t dacStart() {
    esp_err_t ret;
    // dac Configuration
    spi_device_interface_config_t hspi_dac_config = {
        .command_bits = 0,
        .address_bits = 0,
        .mode = 0,
        .duty_cycle_pos = 128,
        .clock_speed_hz = 1000000,  // 1MHz
        .spics_io_num = DAC_CS,
        .queue_size = 5,
    };

    // add dac to HSPI
    ret = spi_bus_add_device(HSPI_HOST, &hspi_dac_config, &dacHandle);
    if (ret != ESP_OK) {
        Serial.printf("Failed to add DAC to HSPI: %s\n", esp_err_to_name
            (ret));
    } else {
        Serial.println("DAC added to HSPI");
```

```
    }
    return ret;
}
```

Inclusion of DAC in SPI interface and configurations.

After the device is initiated, `setDAC` function is responsible for generating a staircase waveform of the signal selected by the user from the computer interface. Since the DAC8552 only owns one output buffer capable of being used by DAC A or DAC B channels, a decision was made to send values to both channels as separate functions so that both can make use of the buffer. Therefore, the component recognizes two registers.

```
// Write buffer A; load channel A
#define DAC_A_WRITE_AND_LOAD   0x10  // 00010000
// Write buffer B; load channel B
#define DAC_B_WRITE_AND_LOAD   0x24  // 00100000

void setDAC(uint8_t command, uint16_t dataBytes) {

    uint8_t tx_data[3]; // 3 bytes

    tx_data[0] = command;   // control register
    tx_data[1] = (dataBytes >> 8) & 0xFF; // high half 16-bit sequence
    tx_data[2] = (dataBytes & 0xFF);     // low half 16-bit sequence

    spi_transaction_t transaction;
    memset(&transaction, 0, sizeof(transaction));

    // Set up transaction
    transaction.length = 24;
    transaction.tx_buffer = tx_data;
    transaction.flags = 0;

    esp_err_t ret = spi_device_polling_transmit(dacHandle, &transaction)
        ;
    if (ret != ESP_OK) {
        Serial.printf("ERROR: Failed to set DAC 0x%02X  0x%02X  0x%02X:
            %s\n", tx_data[0], tx_data[1], tx_data[2], esp_err_to_name(
            ret));
    } else {
        Serial.printf("DAC command 0x%02X | highData 0x%02X | lowData 0x
            %02X\n", tx_data[0], tx_data[1], tx_data[2]);
    }
}
```

DAC transmission function.

Figure 4.20 presents the data transmission between the MCU and DAC8552 device. The acquisitions were performed using static data byte values, specifically, full-scale (0xFFFF) and middle-scale (0x8000) written in channel A and channel B, respectively.

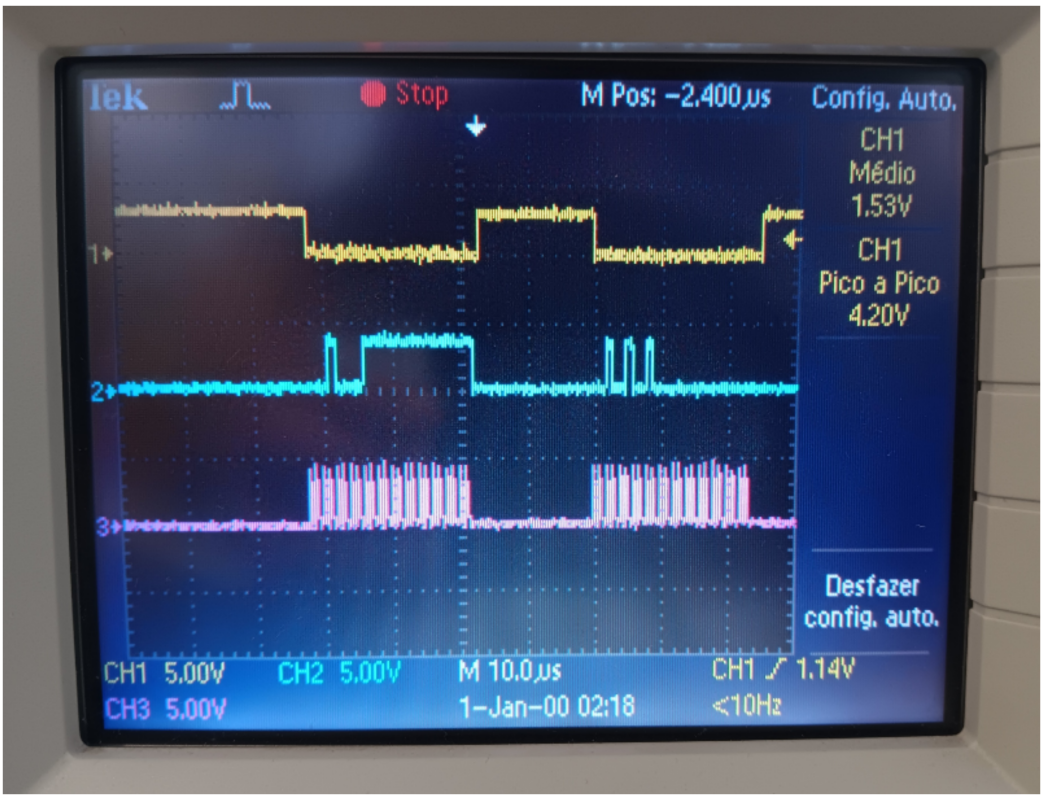


Figure 4.20 Data transmission between MCU and DAC8552. Timescale $10\mu s$. The yellow signal represents the $\overline{CS}$ of the device. The blue signal corresponds to MOSI line. The purple signal corresponds to SCLK line.

Digital Potentiometer

The MCP42100 digital potentiometer is a dual 8-bit precision resistor network of 100 $k\Omega$, supporting an SPI interface with a baudrate up to 10 MHz and modes 0 or 3, and an operating voltage between 2.7 V and 5.5 V. The component contains two digitally controlled resistances that allow zero-scale to full-scale connections and three terminals. The total resistance is given between terminals A and B, and the terminal Wiper sets a measuring point for the programmable resistance R_{AW} or R_{BW}.

The data is transmitted and received by the component using Serial Out (SO) and Serial In (SI) pins, associated with the MISO and MOSI pins of the master controller (MCU). The SPI controller begins data transmission to the component by setting the clock frequency and driving $\overline{CS}$ to an active level (Figure 4.4). In the implemented device, the SPI3 controller manages data transfer using the CS3 pin (GPIO22).

The digital potentiometer is hosted in VSPI controller and is added with `digipotStart` function. Afterwards, both resistor networks are set to the middle scale. Figure 4.21 presents the communication between the microcontroller and the digital potentiometer in the VSPI bus. The acquisitions were performed using static resistance values, specifically, 32 (0x20) and 16 (0x10) steps in potentiometers 1 and 2, respectively.

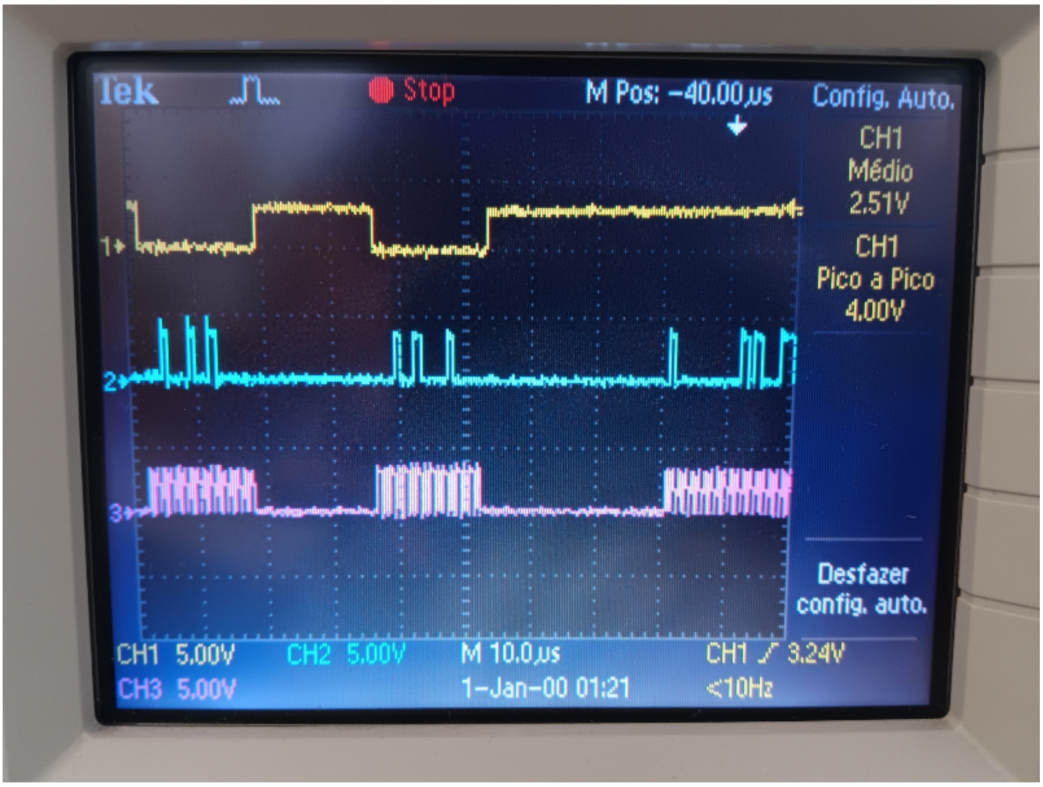


Figure 4.21 Data transmission between MCU and MCP42100 digital potentiometer. Timescale $10\mu s$. The yellow signal represents the $\overline{CS}$ of the device. The blue signal corresponds to MOSI line. The purple signal corresponds to SCLK line.

```
#define DIGIPOT_CS    GPIO_NUM_22
spi_device_handle_t digipotHandle;

esp_err_t digipotStart() {
    esp_err_t ret;

    // Digipot Configuration
    spi_device_interface_config_t vspi_digipot_config = {
        .command_bits = 0,
        .address_bits = 0,
        .mode = 0,
        .duty_cycle_pos = 128,
        .clock_speed_hz = 1000000,  // 1MHz
        .spics_io_num = DIGIPOT_CS,
        .queue_size = 5,
    };

    // Add digipot to VSPI
    ret = spi_bus_add_device(VSPI_HOST, &vspi_digipot_config, &
       digipotHandle);
    if (ret != ESP_OK) {
        Serial.printf("Failed to add Digital Potentiometer to VSPI: %s\n
           ", esp_err_to_name(ret));
    } else {
        Serial.println("Digital Potentiometer added to VSPI");
    }
    return ret;
}
```

Inclusion of MCP42100 in SPI interface and configurations.

The positions of the wiper 1 and wiper 0 terminals in the digital potentiometer can be individually changed using the `setWiper` function. The first 8-bit sequence contains the command selection bits, potentiometer selection bits and dummy bits, while the second byte sets the impedance value of the wiper. Therefore, generating a signal requires calling the function once for each wiper, to achieve individual impedance control, or a single-function with both wipers selected, both setting the same resistance value [39].

```
void setWiper(uint8_t commandByte, uint8_t valueByte) {

    // Prepare transaction
    uint8_t = tx_data[2];
    tx_data[0] = commandByte;
    tx_data[1] = valueByte;

    spi_transaction_t = transaction;
    memset(&transaction, 0, sizeof(transaction));

    // Set up transaction
    transaction.length = 16;  // 16 bits (2 bytes)
    transaction.tx_data = tx_data;

    // Execute transaction
    esp_err_t result = spi_device_polling_transmit(digipotHandle, &
        transaction);
    if (result != ESP_OK) {
        Serial.printf("ERROR: Failed to set wiper: %s\n",
            esp_err_to_name(result));
    }
}
```

MCP42100 transmission function.

I/O expanders & Capacitor Bank

The MCP23S08 component is an 8-bit remote bidirectional I/O expander with an operational voltage range of 1.8 to 5.5 V. The device communicates using an SPI interface with a baud rate up to 10 MHz. The component includes two externally biased hardware address pins, allowing up to four devices to share the $\overline{CS}$ line. The device inline with *DAC A* dataflow has both address bits in low state, while the second I/O expander has bit A1 at a low state and bit A0 at an active state [40].

SPI write and read operations begin by pulling $\overline{CS}$ to a low state. A specific 16-bit start sequence for the operation is then sent to the device, followed by at least one data byte. After the operation completes, $\overline{CS}$ returns to its idle state [40]. The activation and disablement of the I/O pins in the MCP23S08 affect the state of multiple connected switches in the 74HC4066,653 SPST analog switch [53]. The 74HC4066,653 integrated circuit (IC) hosts 4 analog switches and has a recommended operational voltage of 2 V to 10 V, ideal for 3.3 V or 5 V operation [53]. In addition, the SPST design allows individual control of the capacitors connected to the switches. Each switch consists of two input terminals (Y and Z) and an active terminal, controlled by MCP23S08 I/O pins, that closes both terminals Y and Z. This setup increases the number of I/O pins required to control the capacitors. The simulator electronic circuit employs 2 analog switch components and 8 capacitors with different values (1 nF, 2 nF, $\approx$ 4 nF, 8 nF,

16 nF, 33 nF, 64 nF and 128 nF), with a global capacitance of ≈ 255 nF. The capacitance bank circuit is presented in Figure 4.22.

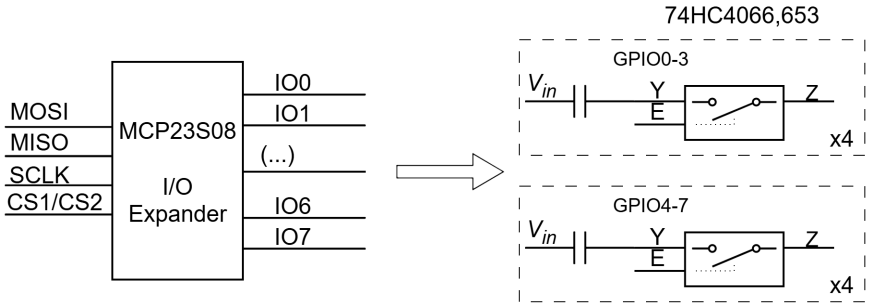


Figure 4.22 Block diagram of the implemented capacitance bank in the simulator. Each DAC channel pipeline contains an I/O expander and, therefore, 2 SPST analog switch ICs. The capacitors are connected to terminal Y.

The IOCON register contains several bits to change the configuration of the device. The data byte 0x08 corresponds to Hardware Address Enable (HAEN) control bit and enables/disables the hardware address pins (A1, A0) on the MCP23S08. Since both I/O expanders are configured as a single device sharing a common CS line, this distinction is fundamental to the device's operation [40]. Before operation, both devices are added to VSPI bus with `IoSpiStart` and configured through `startMCP23S08` functions.

```
#define IO_EXP_CS_1    GPIO_NUM_2
spi_device_handle_t ioExpanderHandle;

esp_err_t IoSpiStart() {
    esp_err_t ret;

    // I/O Exp Configuration
    spi_device_interface_config_t vspi_io_config = {
        .command_bits = 0,
        .address_bits = 0,
        .mode = 0,
        .duty_cycle_pos = 128,
        .clock_speed_hz = 1000000,  // 1MHz
        .spics_io_num = IO_EXP_CS,
        .queue_size = 5,
    };

    // I/O Expander to VSPI
    ret = spi_bus_add_device(VSPI_HOST, &vspi_io_config, &
       ioExpanderHandle);
    if (ret != ESP_OK) {
        Serial.printf("Failed to add IO Expander to VSPI: %s\n",
           esp_err_to_name(ret));
    }
    return ret;
}

(...)

// MCP23S08 Register Addresses
#define MCP23S08_IODIR   0x00  // Direction register (0=output, 1=input)
#define MCP23S08_IOCON   0x05  // Configuration register
```

```
#define MCP23S08_GPPU    0x06  // Pull-up resistor register
#define MCP23S08_OLAT    0x0A  // Output latch register

void startMCP23S08(uint8_t deviceOpCode) {

    // Enable hardware addressing
    setIOExpander(deviceOpCode, MCP23S08_IOCON, 0x08);

    // Configure all pins as outputs
    setIOExpander(deviceOpCode, MCP23S08_IODIR, 0x00);

    // Disable pull-ups
    setIOExpander(deviceOpCode, MCP23S08_GPPU, 0x00);

    // Set all pins low initially
    setIOExpander(deviceOpCode, MCP23S08_OLAT, 0x00);
}
```

Inclusion of MCP23S08 in SPI interface and configurations.

The eight pins of each I/O expander are configured using `setIOExpander`. The 24-bit sequence is created using an opcode byte, a register address byte, and a data byte, and the interface between both master and slave components is presented in Figure 4.23. The acquisitions were performed using static pin combinations of 4/5 (0x30) and 1/2/3/4 (0x0F) associated, respectively, with devices 1 and 2.

```
#define MCP23S08_OLAT     0x0A  // Output latch register
void setIOExpander(uint8_t control, uint8_t registerAddress, uint8_t
    dataByte) {

    // Prepare transaction
    uint8_t tx_data[3];

    tx_data[0] = control;
    tx_data[1] = registerAddress;
    tx_data[2] = dataByte;

    spi_transaction_t transaction;
    memset(&transaction, 0, sizeof(transaction));

    // Set up transaction
    transaction.length = 24;
    transaction.tx_buffer = tx_data;
    transaction.flags = 0;

    esp_err_t ret = spi_device_polling_transmit(ioExpanderHandle, &
        transaction);
    f (ret != ESP_OK) {
    Serial.printf("ERROR: Failed to set IO expander register 0x%02X: %s\
        n", registerAddress, esp_err_to_name(ret));
    }
}
```

MCP23S08 IO pin interface function.

Figure 4.23 Data transmission between MCU and both MCP23S08 I/O expanders. Timescale $10\mu s$. The yellow signal represents the $\overline{CS}$ of both components. The blue signal corresponds to MOSI line. The purple signal corresponds to SCLK line.

Memory Card

The micro SD card slot features 8 connections, similar to the memory card used throughout the development, with a push-push mechanism, and is surface mount. Communication between the system and the SD card requires a common interface with a Virtual File System (VFS). A VFS is an abstraction communication layer that provides a single interface for different file systems [54]. The initialization of the memory card is executed in `sdCardInit`, which does the following:

1. initializes the SPI slave device in the SPI master driver with the specified parameters;
2. starts the SD card with default configurations, specifying the master bus and CS pin;
3. mounts FAT partition on SD card;
4. registers FATFS with VFS, with '/sdcard' prefix;

```
#define SD_CARD_CS GPIO_NUM_5
static const char* SD_MOUNT_POINT = "/sdcard";
sdmmc_card_t* card = NULL;

esp_err_t sdCardInit() {
    esp_err_t ret;
```

```
    // mount the SD card
    esp_vfs_fat_sdmmc_mount_config_t mount_config = {
        .format_if_mount_failed = false,
        .max_files = 5,
        .allocation_unit_size = 16 * 1024
    };

    // SD card configuration
    sdspi_device_config_t slot_config = SDSPI_DEVICE_CONFIG_DEFAULT();
    slot_config.host_id = VSPI_HOST;
    slot_config.gpio_cs = SD_CARD_CS;

    // initialize SD SPI host
    sdmmc_host_t host = SDSPI_HOST_DEFAULT();
    host.slot = VSPI_HOST;
    host.max_freq_khz = 4000;

    // Mount SD FAT in VFS
    ret = esp_vfs_fat_sdspi_mount(SD_MOUNT_POINT, &host, &slot_config, &
        mount_config, &card);

    if (ret != ESP_OK) {
        Serial.printf("Failed to initialize SD card: %s\n",
            esp_err_to_name(ret));
        return ret;
    }
    return ret;
}
```

Inclusion of SD card device in SPI interface and configurations.

The device interacts with the micro SD card depending on the command received from the software interface, as detailed in Figure 4.24, using the VSPI driver and pulling the pin $\overline{CS0}$ to a low state. The microSD card handles signal storage and reproduction and allows multiple operations to be performed.

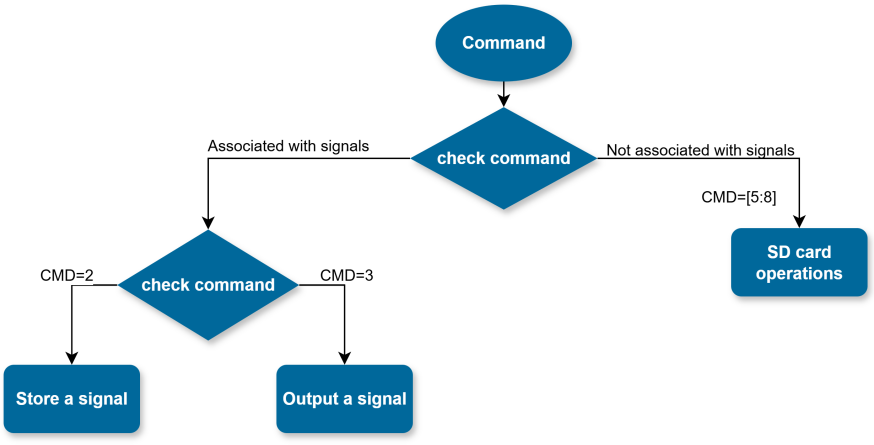


Figure 4.24 Memory card operations flow.

4.3.4 Analog Front-end Implementation

The analog section of *Signal Generation* section is responsible for the reproduction of a signal. In this sub-module, both 16-bit resolution channels of the DAC reproduce stepped waveforms provided by the microcontroller via SPI. A low-pass reconstruction filter is required to limit the bandwidth of the DAC output 4.25. These reconstruction filters, also known as anti-aliasing filters, eliminate high-frequency components and inherent mirror frequencies associated with discrete-time reconstruction that arise from the sampling process, explained by the typical sampling of continuous-time signals to discrete-time signals and the fast voltage transitions of the DAC. The operational amplifier (Op-Amp) used in the filter circuit is *OPA350*, manufactured by Texas Instruments, due its rail-to-rail input and output, unity gain stability and optimization for low voltage single supply operations [33, 34, 55]. The voltage follower circuit is developed using *OPA388 (Texas Instruments)* Op-Amp, featuring a low offset voltage ($\pm 0.25 \mu V$), and introduced noise, specially from 0.1 to 10 Hz (140 nVPP), important in ECG applications, and a true rail-to-rail input and output [33, 34, 56].

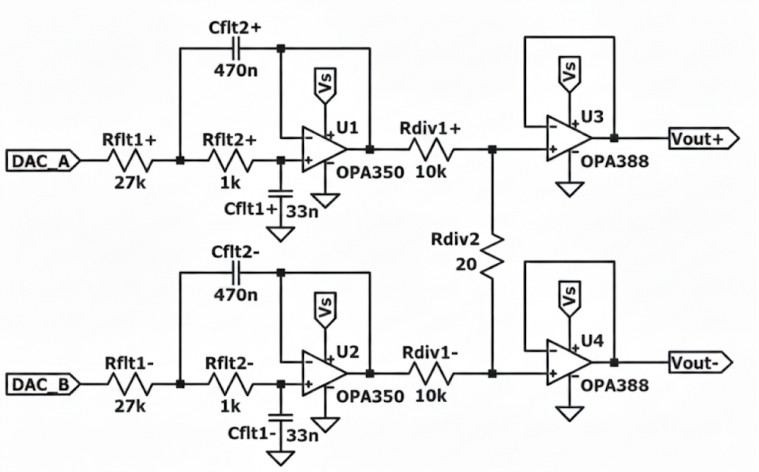


Figure 4.25 Reconstruction filter and attenuation circuit schematic [33].

In Figure 4.25, the reconstruction filter, attenuation and buffering circuit comprise low-pass filters, two voltage dividers sharing resistance R_{div2} and two output buffers. The attenuation section is created by R_{div1+}, R_{div2} and R_{div1-} to lower the resulting voltage to milliVolt levels (Equation 4.4), while both voltage follower buffers U_3 and U_4 reduce the output impedance associated with both voltage dividers.

$$V_{out} = \frac{R_{div2}}{R_{div2} + R_{div1}} \times V_{in} = \frac{20k}{10k + 20k} \times 3.3V \approx 6.5 \quad [mV] \tag{4.4}$$

The signal reconstruction is performed with the implementation of a second-order Butterworth filter with Sallen-Key topology, whose performance can be measured using the frequency response. This property is a quantitative measure of the magnitude and phase of the output as a function of input frequency, with an attenuation (negative magnitude) value of 20 dB per decade per order of the filter at the cutoff frequency. The Butterworth filter is characterized by its maximally flat frequency response in the passband, making it ideal for ECG signal applica-

tions. In comparison with other classes of filters, such as Chebyshev type I or Elliptic filters, it does not introduce ripple in the pass-band maintaining the integrity of the signal. The main disadvantage associated with this class of filters is the relatively wide transition band between the passband and stopband regions. However, filters can be cascaded to increase the overall order of the system and its frequency response, reducing the transition band to the ideal 'wall' at the cost of phase-shift increase. The phase shift introduced by this class is another disadvantage, with a value of 45 degrees per order of the filter at the cutoff frequency. The relation between phase shift (ϕ) and time delay (τ_p) is expressed in Equation 4.5 for single frequencies, and the group delay (τ_g) can be represented across all frequencies with Equation 4.6 [57].

$$\tau_p(\omega) = -\frac{\phi(\omega)}{\omega} \tag{4.5}$$

$$\tau_g(\omega) = -\frac{d\phi(\omega)}{d\omega} \tag{4.6}$$

For the implemented second-order filter, these characteristics result in a phase shift of 90 degrees at the cutoff frequency and a roll-off rate of -40 dB per decade. The cutoff frequency obtained through an LTspice simulation was 243.2 Hz [33, 34], closely similar to the theoretical value of 245.94 Hz calculated using Equation 4.7. The group delay represents the time delay that the filter imposes on each frequency component (Equations 4.5 and 4.6). In Butterworth filters, the group delay is relatively flat in the middle of the passband region and increases close to the cutoff frequency. Therefore, frequency components close to the cutoff frequency are exposed to a greater delay in comparison to frequencies in the pass-band which affects the integrity of the signal. The implemented filter exhibits a group delay of 0.9 ms within the passband region, rising to 1.1 ms as frequency components approach the cutoff (Figure 4.26).

$$f_c = \frac{1}{2\pi\sqrt{R_1 R_2 C_1 C_2}} = \frac{1}{2\pi\sqrt{(27 \cdot 1 \cdot 470 \cdot 33) \cdot 10^{-12}}} \approx 245.94 \quad [Hz] \tag{4.7}$$

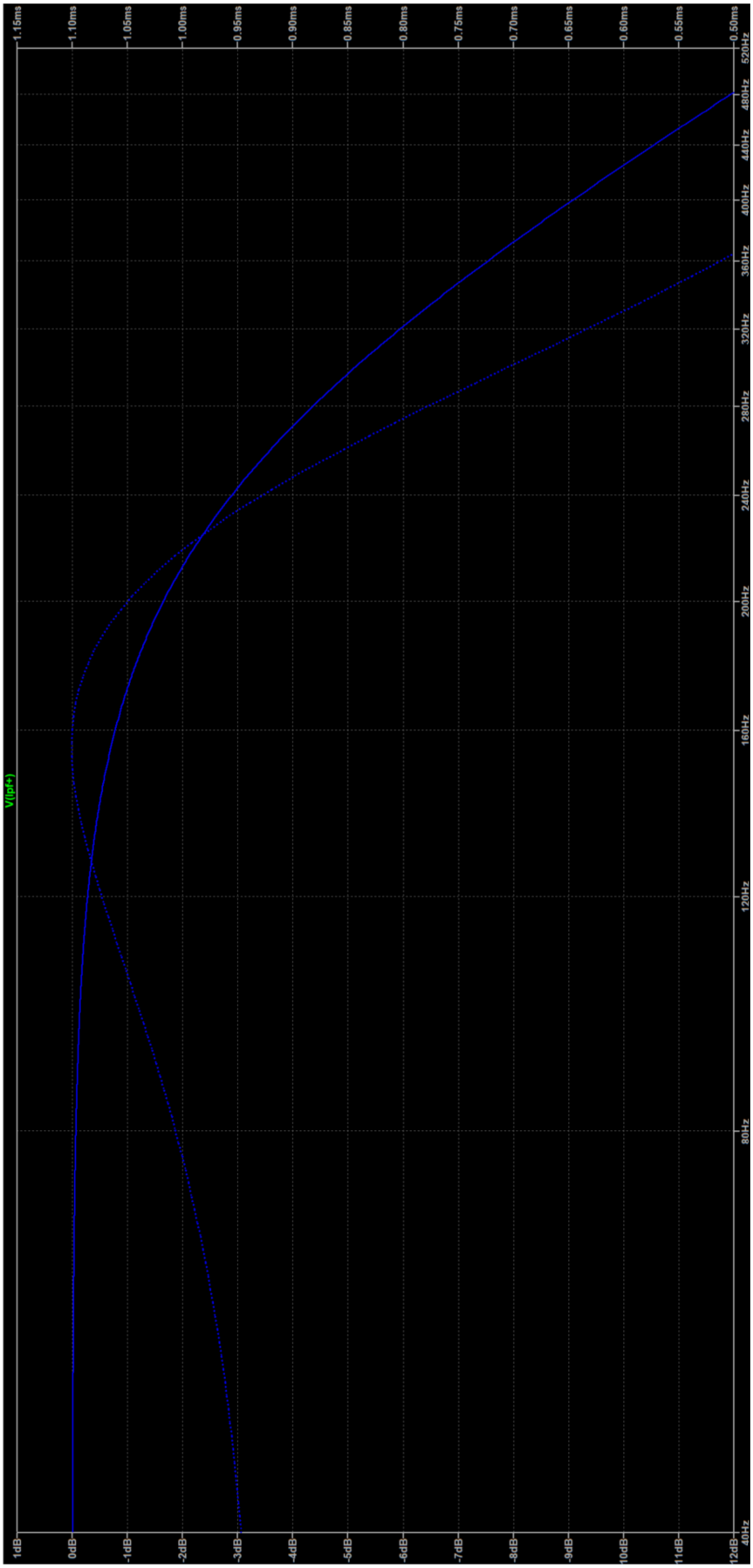


Figure 4.26 SPICE simulation of the frequency response of the Butterworth filter showing the attenuation and group delay.

4.3.5 Skin-Electrode Interface Simulation

The skin-electrode interface allows the user to adjust the resistive and reactance properties of the impedance, opening a door for the simulation of different electrodes and the individual contact area with the surface of the skin. The interface consists of a digital approach of the model proposed by J. Webster (Figure 2.13): R_d is replaced by the digital potentiometer and C_d is replaced by the capacitance bank, forming the reactive capacitance component of impedance, expressed in Equations 4.8 and 4.9.

$$Z_R = R, \quad R = R_{AW} \tag{4.8}$$

$$Z_C = \frac{1}{j\omega C}, \quad \omega = 2\pi f \tag{4.9}$$

The total impedance of the circuit is described by Equation 4.10, demonstrating two possible behaviors associated with both components. In low frequency components, the behavior of a capacitor approaches an open circuit and the total impedance approaches the resistance (Equation 4.8). In high frequency components, the capacitive reactance will reduce the overall impedance of the interface to Equation 4.10.

$$Z_{Total} = \frac{Z_R \cdot Z_C}{Z_R + Z_C} = \frac{R \cdot \frac{1}{j\omega C}}{R + \frac{1}{j\omega C}} = \frac{R}{1 + j\omega RC}, \quad \omega = 2\pi f \tag{4.10}$$

The impedance of the skin-electrode interface can be constant or change over time during signal generation. In IEC 60601 standards (2011), the defined values are $R_d = 51k\Omega$ and $C_d = 47nF$ [33, 58].

4.3.6 Printed Circuit Board Design

Figure 4.27 presents the first prototype of the system developed using Altium Designer PCB Design software, consisting of a 100x100mm two-layer board. In the first implementation, the system is actively working without *JTAG connector* as well as the BOOT and RESET buttons.

This design features two power supply connections (Section 4.3.1) with their associated supply chains. The programming of the system and the interface operations are performed through the *ESP-PROG connector* (JTAG connection is not implemented) and can also serve as an alternative power source, despite the limited capacity. In addition, the prototype features several headers, test points, and several through-hole mounting points to simplify debugging, implementation of additional circuitry, and rework. The headers are located on both sides of the MCU and in the middle of the board to allow the debugging of MCU pin states and to check the VSPI bus and peripherals. The test points are located along the signal generation circuit and in the bottom middle of the board. The through hole array in the middle of the board provides additional space to expand the circuit.

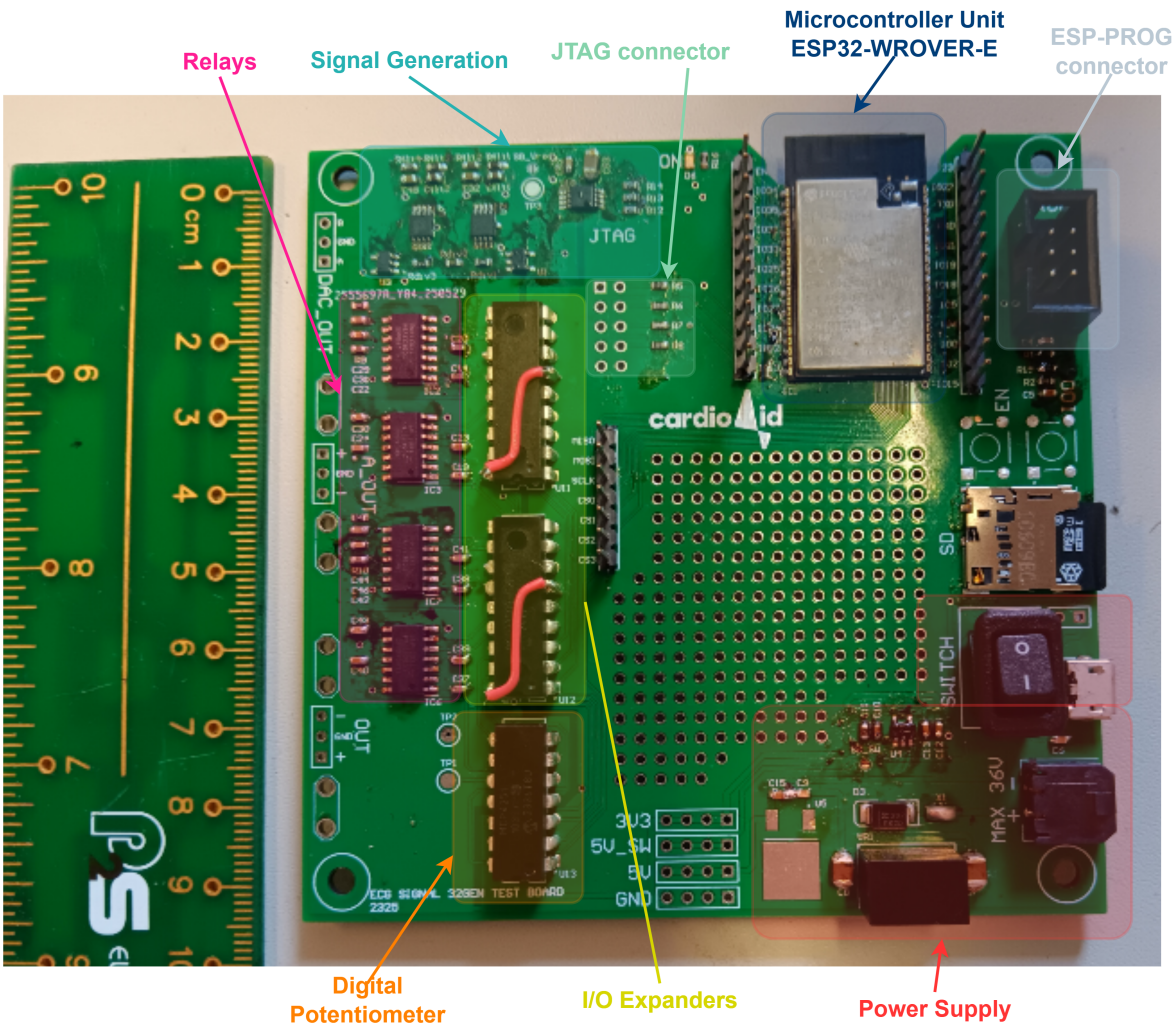


Figure 4.27 First PCB Design - Detailed Layout.

4.3.7 Final System Design

The final design of the signal generation board is presented in Figure 4.28. The board enclosure consists of two acrylic plates separated by threaded standoffs and secured with four screws. The CS 1 and CS 2 connections, associated with both I/O expanders and previously connected using a jump wire, are connected using a simple jumper connector.

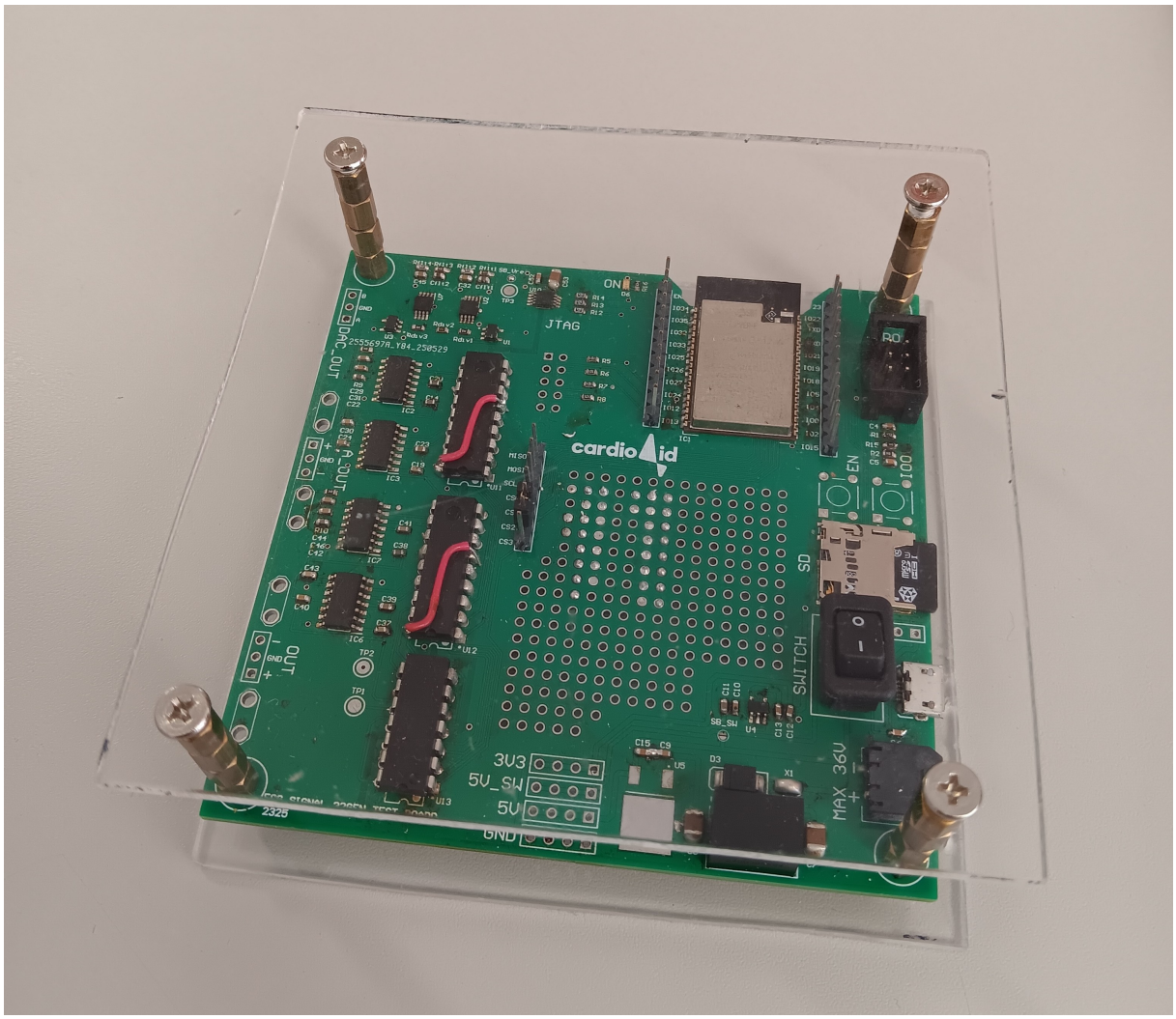


Figure 4.28 Final ECG system design.

Chapter 5

System Evaluation

The performance of the system was evaluated regarding its ability to maintain signal reproduction within an intended duration, set by its duration, as well as step transitions of the digital potentiometer and the expected resistance. Since the capacitive reactance is a minimal component of impedance, the I/O expander tests were not performed.

5.1 Analog Circuit

The signal generation section was evaluated by sweeping the circuit with a chirp signal. The frequency response was captured via oscilloscope FFT analysis, with the probe connected to the *A_OUT* header pins (Appendix E). The test was performed with a specific script (Appendix D), and the measured response is shown in Figure 5.1. The analysis aligns with the simulation circuit result presented in Figure 4.26, in 4.3.4.

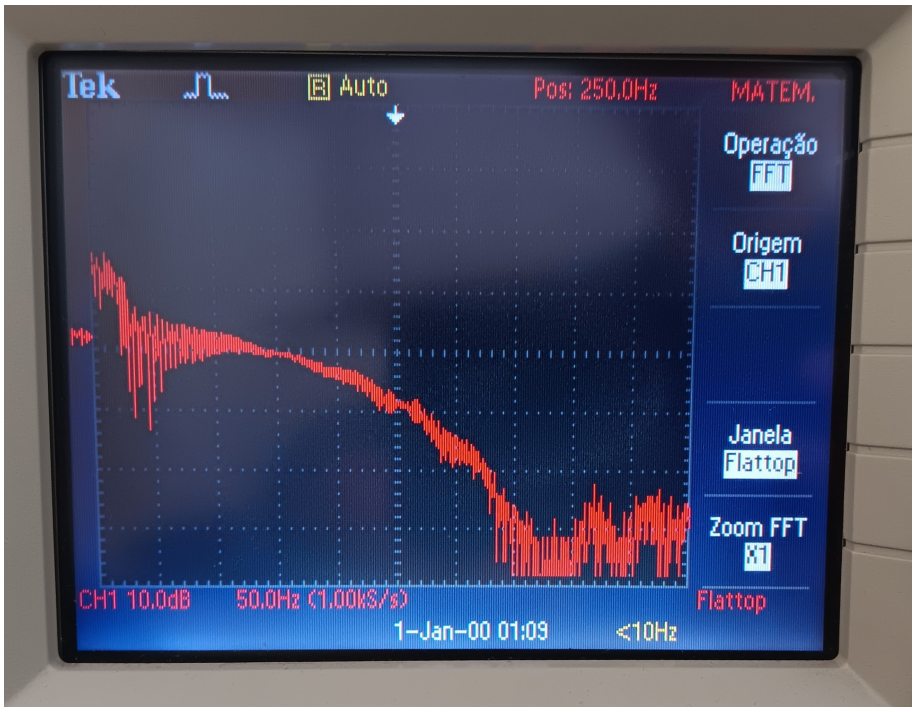


Figure 5.1 FFT performed during the generation of a chirp signal. Frequency scale: 50 Hz.

5.2 Digital Potentiometer

The validation of the digital potentiometer was conducted with a sweep of the wiper position *N*. The resulting voltage changes were captured using a 12-bit ADS7822 ADC, manufactured by Texas Instruments, capable of reaching sampling frequencies up to 200 kHz and with a power supply range between 2.7 and 5.25 V, within the ideal voltage for testing to avoid additional wiring associated with different power sources. The equations and different test settings can be consulted in [39]. The test was performed for both wipers and the acquired values were compared to the expected (theoretical) values.

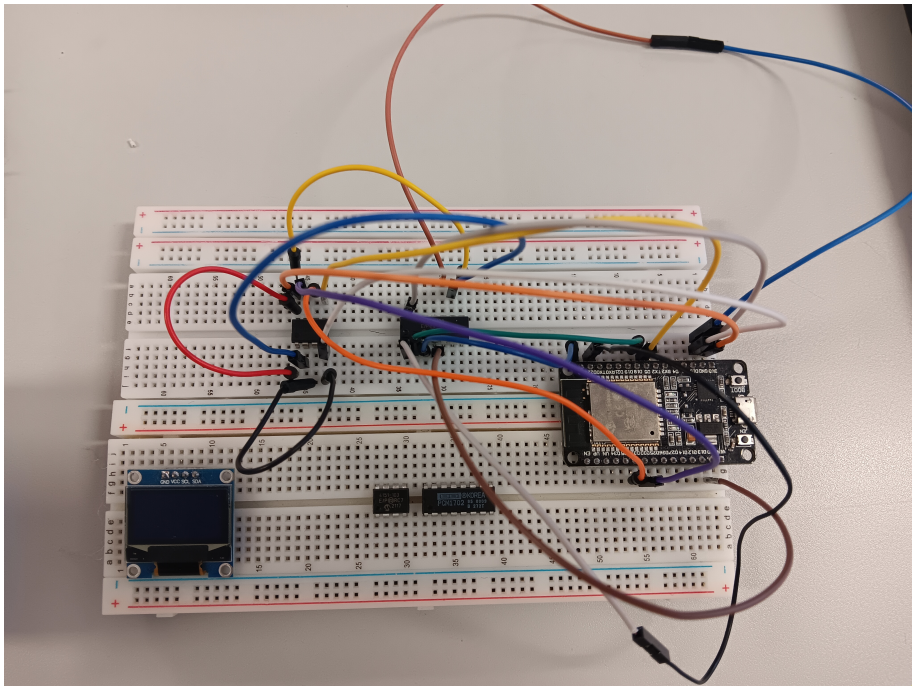


Figure 5.2 Circuit implemented during test of the digital potentiometer. In the image, the jump wires are arranged to test wiper 0 (top side). Note: The LCD display and both integrated circuits on the bottom of the image can be discarded in the interpretation of the circuit.

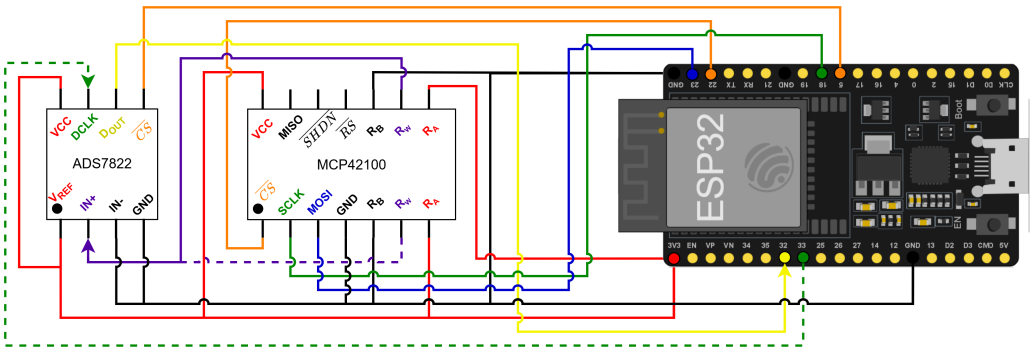


Figure 5.3 Schematic of the implemented test circuit.

The component was tested outside the signal generation board, using a breadboard, an ESP32-WROOM-32E, part of the ESP32-DEVKITC-32E, and a simple circuit created using

jump wires. Figures 5.2 and 5.3, respectively, show the connections performed to test the component and a representation of the connections. The supply source is the ESP32-DEVKITC-32E, which provides a stable voltage of 3.3 V with the on-board Low-Dropout voltage regulator, connected to the computer. The number of bits that compose the data value, provided by the ADC, allows the assessment of the resulting values from the digital potentiometer with a higher precision. In addition, each value is the result of the average of 20 acquired samples. The C/C++ script implemented can be consulted in C.

$$R_A(N) = \frac{R_{AB}(256 - N)}{256} = R_{AB}\alpha; \quad \alpha = \frac{D}{256}; \quad D \in [1 : 256] \tag{5.1a}$$

$$R_B = \frac{R_{AB}N}{256} = R_{AB}\alpha; \quad \alpha = \frac{D}{256}; \quad D \in [1 : 256] \tag{5.1b}$$

$$V_{out} = \frac{R_B}{R_A + R_B} \cdot V_{in} = \frac{\frac{R_{AB}D}{256}}{\frac{R_{AB}(256 - D)}{256}} \cdot V_{in} = \frac{D}{256 - D} \cdot V_{in}; \quad \alpha = \frac{D}{256}; \quad D \in [1 : 256] \tag{5.1c}$$

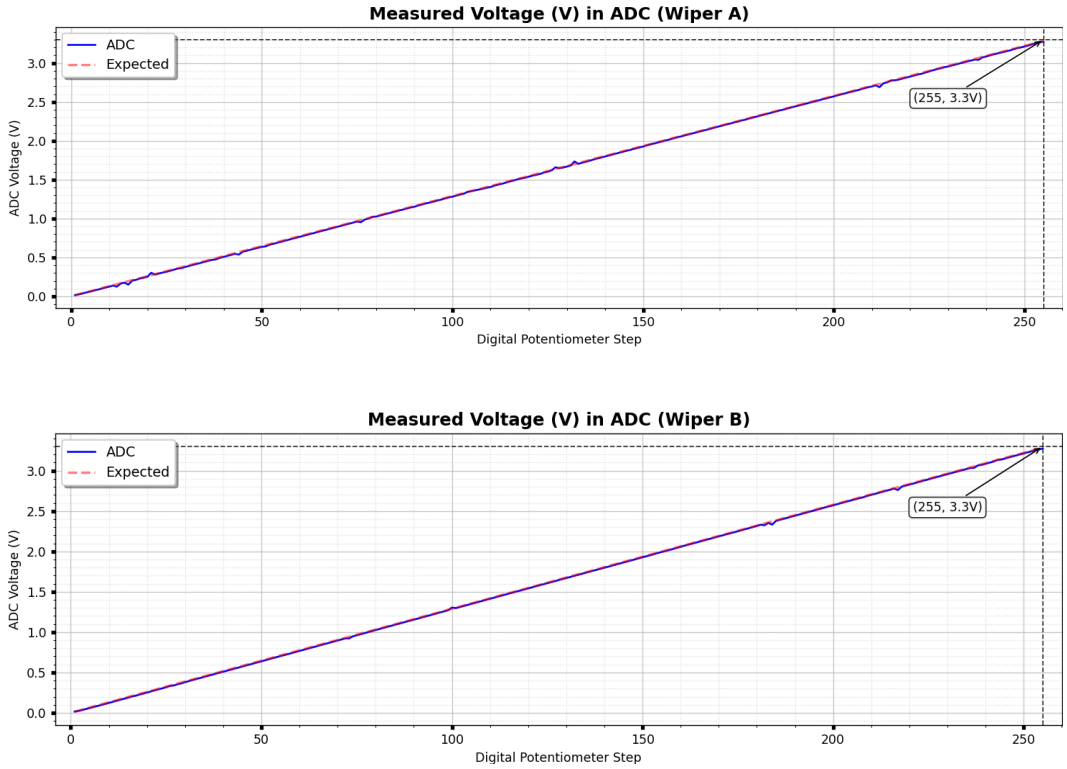


Figure 5.4 Voltage measured by the 12-bit ADS7822 for MCP42100 wipers. A (0) and B (1). The dashed red line represents the expected behavior in optimal conditions and the blue line is the voltage representation of ADC measurements.

The measurements of the ADC acquisitions were extracted and compared to the expected values using equations 5.1a, 5.1b and 5.1c, presented in the component datasheet [39]. In

Figure 5.4, it is noticed that both digital potentiometers exhibit a near linear relationship with the wiper setting, with a correlation value of 99%. Although voltage acquisitions may appear precise in Figure 5.4, small variations can introduce significant changes in signals. Figure 5.5 illustrates the voltage discrepancies between the theoretical and collected values.

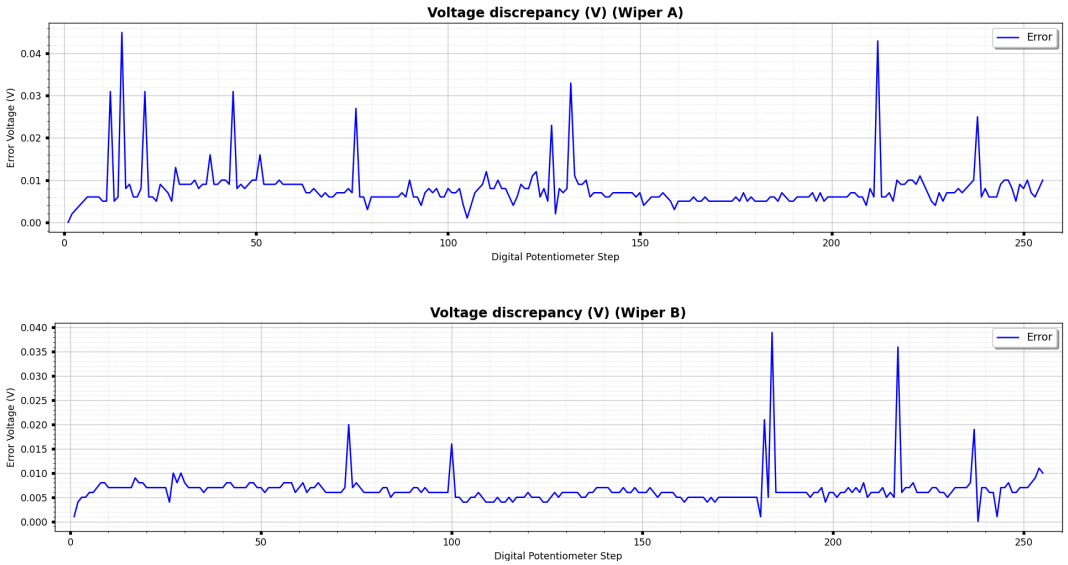


Figure 5.5 Voltage discrepancies between the theoretical and measured values.

5.3 Task Execution Time

5.3.1 Output Task - Producer

The *outputbufferAddTask*, i.e. the producer task, is responsible for filling the shared buffer with the consumer task, retrieving data from the SD card. The global execution time of the task depends on the theoretical production, T_{prod}, and the task execution time, T_{Exe}. Equation 5.2 expresses the approach taken to obtain the production time, ensuring that the resulting value is always bigger than $\frac{BatchSize}{f_s}$, thus preventing buffer emptiness.

$$T_{prod} = \frac{\frac{1}{f_s} \cdot BatchSize + 1000}{999} - 1 \tag{5.2}$$

To ensure the producer task meets the target execution time, the end-of-task delay T_{delay}, is calculated based on two conditions:

$$1.\ T_{Exe} \geq T_{prod} \ \rightarrow \ T_{delay} = 1ms$$

$$2.\ T_{Exe} < T_{prod} \ \rightarrow \ T_{delay} = T_{prod} - T_{Exe}$$

The proposed condition eliminates fixed delays, enhancing system versatility while preventing program blocking, watchdog timeouts, and task starvation. Figure 5.6 shows task execution time measured by toggling a GPIO pin at the start and end of each loop iteration. The first condition enables preemption by other pending tasks on the CPU core running *outputbufferAddTask*, allowing higher-priority tasks to execute. The second condition maintains

constant production rate to ensure data consumption per batch matches the production rate while preventing timeouts.

To prevent buffer underrun, the production time T_{prod} (Equation 5.2) is set slightly below the theoretical execution time, ensuring continuous data availability. The buffer implements a pre-fill phase before enabling the hardware timer and initiating consumption. The producer task is expected to complete before the consumer task due to data presence in the shared buffer.

The execution time is calculated as the difference between tick counts captured at task start and end using *xTaskGetTickCount()*. Production timing was verified via oscilloscope measurement, as shown in Figure 5.6. Under these conditions, the code snippet below yields a stable T_{prod} and oscillating T_{Exe} values ranging from approximately 414 to 497 milliseconds.

Figure 5.6 Producer task execution time, acquired using GPIO33, with an execution time of 624 ms. Sampling frequency: 800 Hz. *BatchSize*: 500 values. Note: Due to the duration of the task, the time window could not be expanded, possibly preventing a more precise measurement. Time scale: 100 *ms*.

```
void outputBufferAddTask(void* parameter) {
    /* Middle start declarations and execution */
    TickType_t startTicks;
    TickType_t endTicks;
    int32_t elapsedTime;

    for (;;) {
        if (ecgOutConfig->active) {
            digitalWrite(33, HIGH);
            startTicks = xTaskGetTickCount();

            /*Execution*/

        }

        endTicks = xTaskGetTickCount();
        digitalWrite(33, LOW);
        elapsedTime = endTicks - startTicks;
```

```
        if (elapsedTime >= currentConfig.productionCountMS(CHUNK_SIZE))
            {
             vTaskDelay(elapsedTime - currentConfig.productionCountMS(
                CHUNK_SIZE));
        } else {
             vTaskDelay(currentConfig.productionCountMS(CHUNK_SIZE) -
                elapsedTime);
        }
    }
}
```

Tick acquisitions associated with the measurements in Figure 5.6.

5.3.2 Output Task - Consumer

The main requirement of the consumer task is its time precision execution, guaranteed by the ISR, and the reproduction of the signal with the expected number of samples.

Interrupt Service Routine

An ISR is a code block triggered by hardware or software events. The ISR for signal reproduction is triggered by the hardware timer every $\frac{1000000}{f_s}$ μs and is configurable via the computer interface. Figure 5.7 shows ISR timing performance, consisting of data packet extraction from the ring buffer and copying to a critical section shared with the consumer task. ISR duration was measured using *xTaskGetTickCount()* positioned in the consumer task with SPI transactions commented out. The *startTicks* and *endTicks* variables within the loop calculate the *elapsedTime* value.

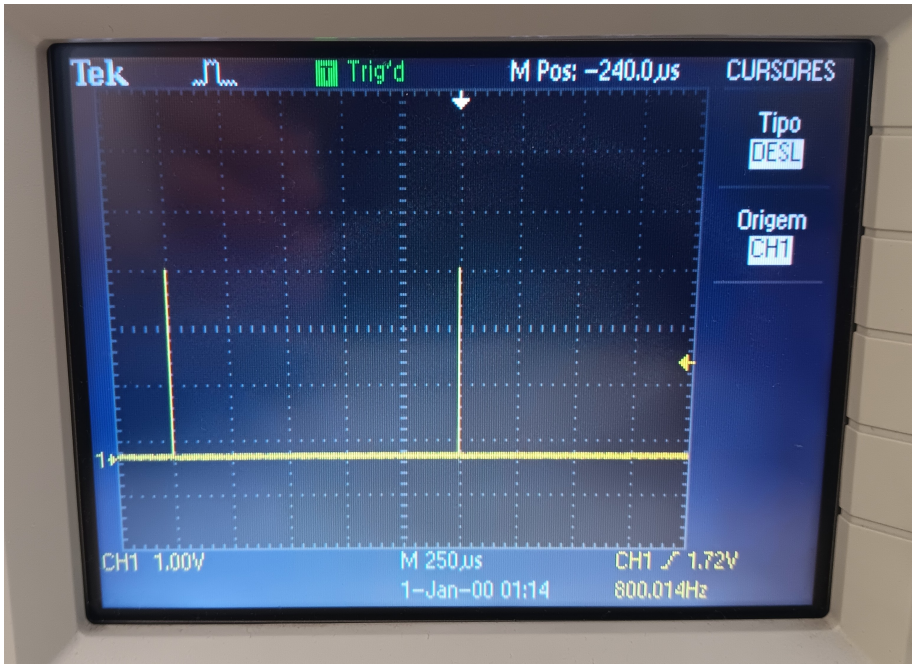


Figure 5.7 Interrupt routine execution time, acquired using GPIO32, with an execution time of 1.25 ms, correspondent to a sampling frequency of 800 Hz. Time scale: 250 μs.

```
bool IRAM_ATTR onTimer(gptimer_handle_t timer, const
    gptimer_alarm_event_data_t* edata, void* user_ctx) {
  /* Middle start declarations and execution */
  digitalWrite(32, HIGH);

  /*Execution*/
  }
  digitalWrite(32, LOW);

  return true;
}
```

Tick acquisitions associated with the measurements in Figure 5.7.

SPI Transmissions & Thread Duration

In the consumer task, data transmission occurs outside the ISR to prevent SPI transaction overhead and ensure signal transmission stability. From the previous chapter, the total SPI transmission time for a single data packet, excluding the CS pin settlement time for each device, is approximately 190 μs, as shown in Figures 4.20, 4.21, and 4.23; the overall time associated with SPI transactions could not be measured using the FreeRTOS utility, thus ESP-IDF timer was used, providing microsecond resolution and a result of $\approx 250 - 270 \mu s$ (Figure 5.8).

Figure 5.8 Sequential periodic transmission of 2 data packets to DAC, digital potentiometer and I/O expanders. The CS pins of the mentioned peripherals are, respectively, the yellow, purple and green lines. The blue line represents GPIO32 toggle controlled with ISR and consumer task.

The duration of task execution was measured by accumulating the elapsed time between consecutive iterations, calculated as the difference between *endTicks* and *startTicks*. The number os reproduced samples in the consumer task was measured using a counter *outputCounter* implemented within the consumer task allowed monitoring of the number of reproduced samples. Both tests were conducted for different signal durations: 1, 10, 15, 20, 25 and 30 minutes.

All acquisitions were performed at a sampling frequency of 800 Hz, with the number of test iterations decreasing as the signal length increased. The results are presented in Figures 5.9 & 5.10.

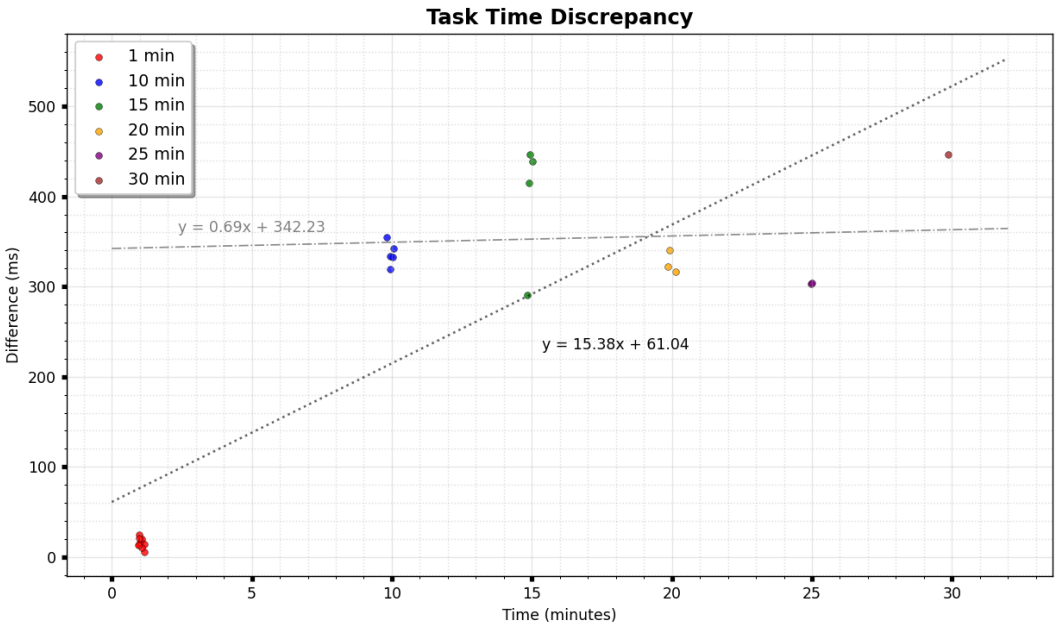


Figure 5.9 Consumer task execution time discrepancies for varying signal durations at 800 Hz sampling rate. The time disparity was calculated as the difference between the obtained and expected execution times. The gray line presents a linear regression discarding the 1-minute time signals.

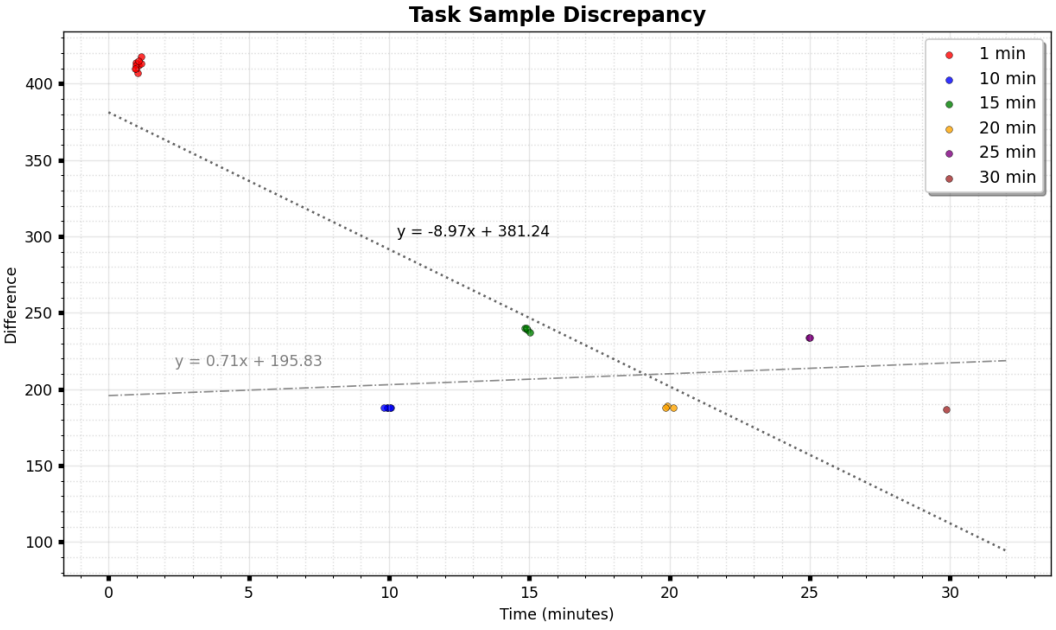


Figure 5.10 Consumer task execution sample discrepancies for varying signal durations at 800 Hz sampling rate. The sample disparity was calculated as the difference between the expected and reproduced samples. The gray line presents a linear regression discarding the 1-minute time signals.

The multiple acquisitions and subsequent data analysis presented in the above plot indicate an increasing disparity between theoretical and actual T_{Exe}, and a decreasing sample

discrepancy over time. The results exhibit a linear regression trend primarily influenced by data collected during 1-minute test intervals. However, extended signal reproductions demonstrate stable sample and time disparity, suggesting this condition may be independent of task loop execution. For each time signal sequence, the relative errors are presented in Table 5.1.

Table 5.1 Relative errors associated with the expected and reproduced samples in each timed signal.

Duration (minutes)	Acquisitions	Relative Error (%)
1	10	0.8590
10	5	0.0392
15	4	0.0332
20	3	0.0196
25	2	0.0195
30	1	0.0130

5.4 Signal Performance

5.4.1 Acquisition Results

The test system was composed of *Cardiowheel* device that was used to acquire signals with different impedance profiles. Acquisitions were performed using semi-dry ECG electrodes, manufactured by Ambu® BlueSensor P, in positions 10 and 2 of the wheel (Figure 5.11). The tests performed required 3 sets of electrodes; the electrodes required to be changed over the testing time due to interference developed with long term use. The left arm (LA) and right arm (RA) electrodes correspond to the positive and negative outputs of the generation board. The tests were performed using record 103 from MIT-BIH Arrhythmia database.

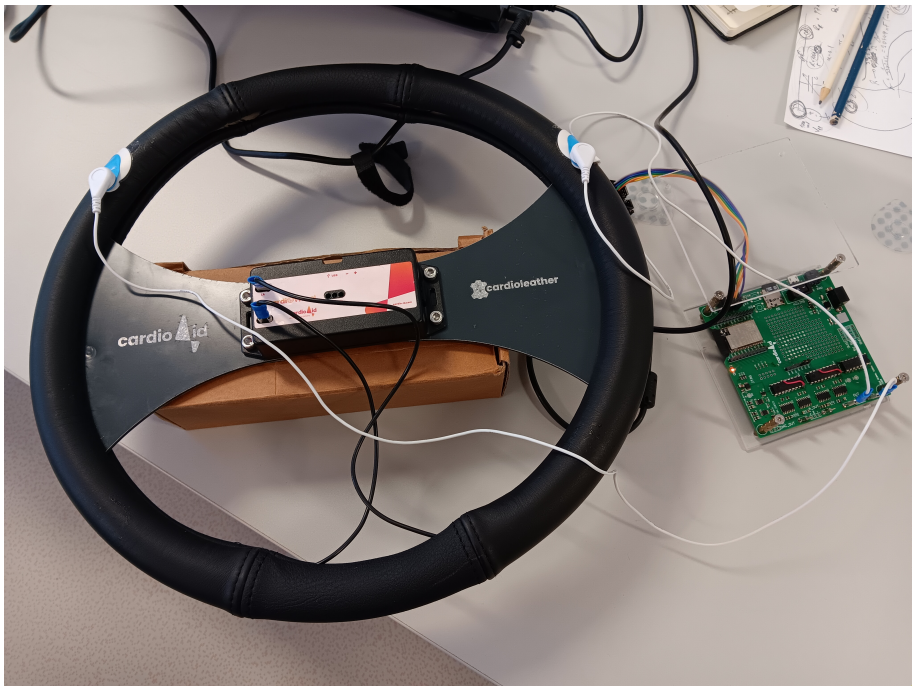


Figure 5.11 Experimental setup comprised of the ECG simulation system, Cardiowheel acquisition device and two semi-dry electrodes. To prevent possible noise sources, the acquisition device was positioned on top of a box.

In order to establish a comparison term, a reference test was performed without the artificial skin-electrode interface, presented in Figure 5.12. The impact of impedance on the ECG waveform was evaluated in positive and negative electrodes, shown in Figures 5.13 and 5.14. In the first 18 seconds, both electrodes have equal impedance ($C_d = 205nF$, $R_d = 20k\Omega$).

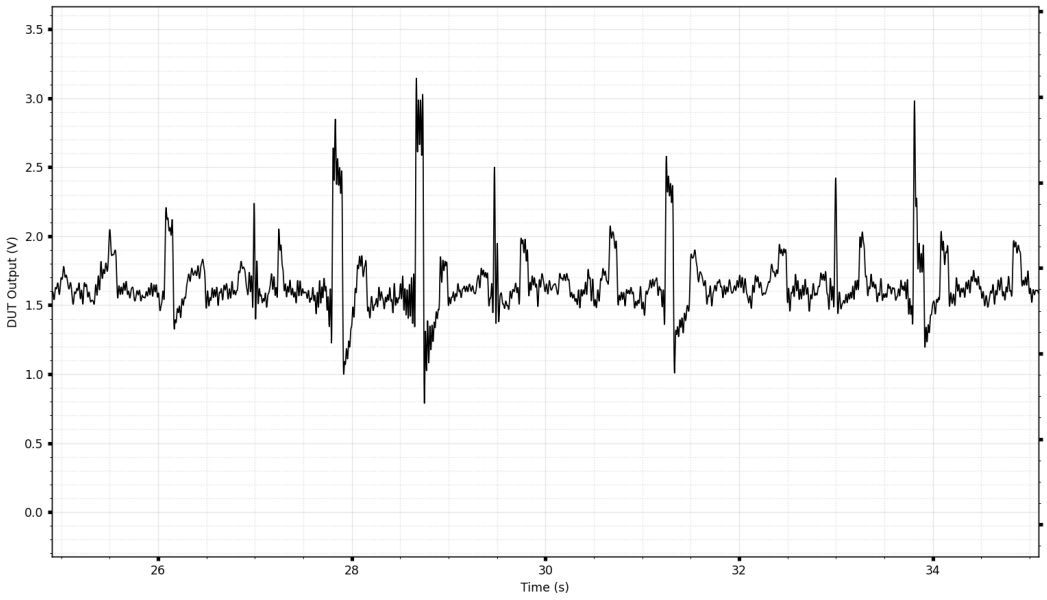


Figure 5.12 Reference ECG signal without skin-electrode impedance.

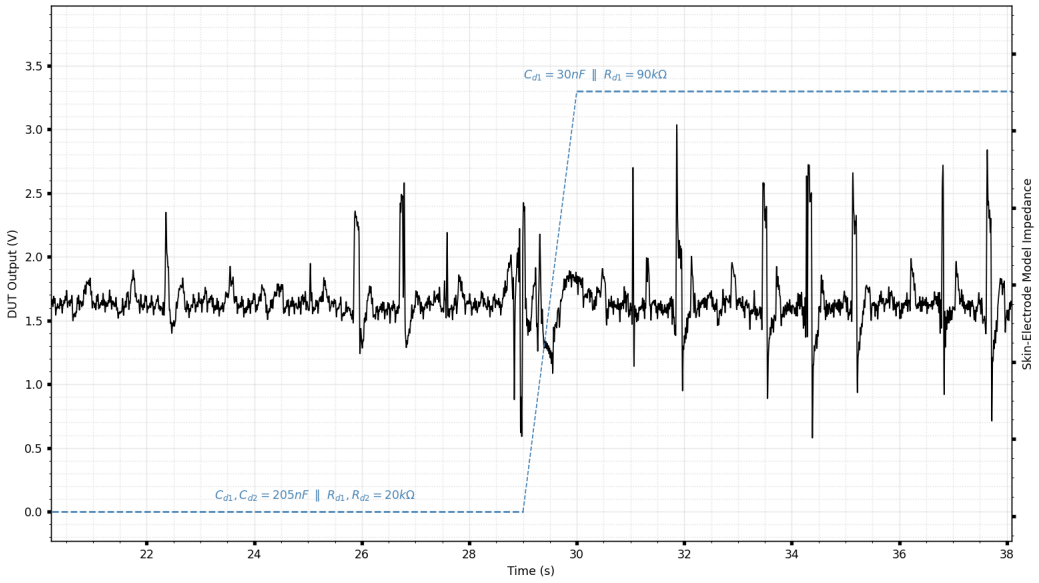


Figure 5.13 Simulation of a swift increase of impedance in positive electrode. At instant 18 seconds, the capacitance decreases and the resistance increases rapidly ($C_d = 30nF, R_d = 90k\Omega$).

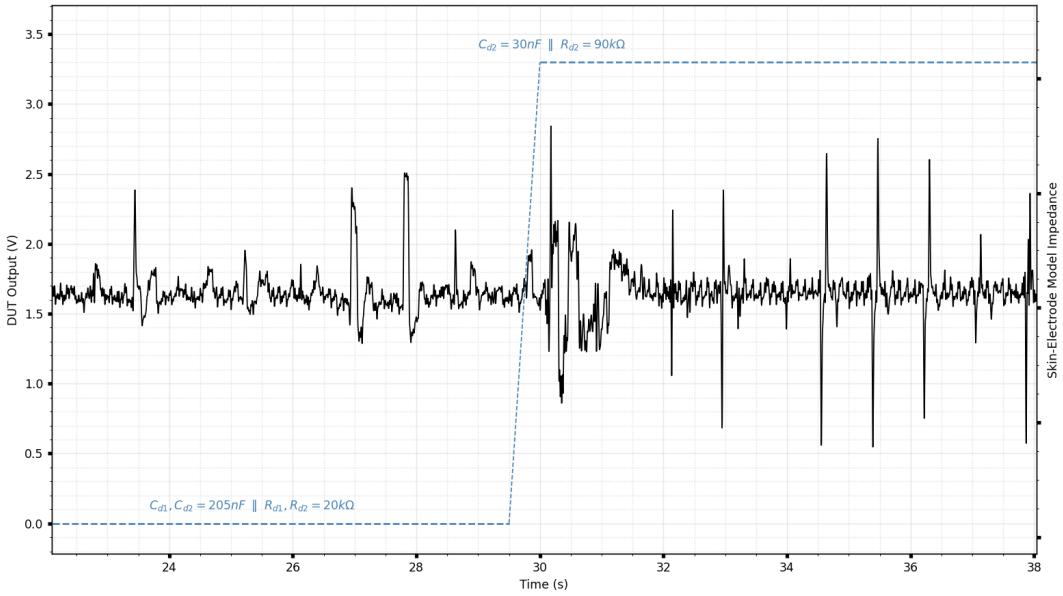


Figure 5.14 Simulation of a swift increase of impedance in negative electrode. At instant 18 seconds, the capacitance decreases and the resistance increases rapidly ($C_d = 30nF, R_d = 90k\Omega$).

The abrupt impedance changes at both electrodes individually correlate with the transient artifact observed at approximately 18 seconds, in addition to the physiological QRS complex transients characteristic of the ECG waveform, and the increase thickening of the signal associated with the noise present in the signal. It can be observed that the impedance has a more pronounced effect on the negative electrode compared to the positive electrode: The clarity of the P wave is reduced, the QRS complex shows abrupt voltage transitions, and the T wave exhibits noise artifacts.

The following two figures present the resulting waveforms from individual resistance changes

at both electrodes. In Figures 5.15 and 5.16, resistance increases in a short time frame, while the remaining components of the model maintain their values. The positive electrode presents a transient, approximately, at instant 30.5 seconds and restored baseline 2.5 seconds after the increase of resistance. The negative electrode presents presents an higher amount of noise content in comparison to the previous signal, although restored the signal baseline 0.5 seconds faster. In the positive electrode, the components of the ECG waveform can be distinguished, whereas the multiple artifacts in the negative electrode do not allow recognition of P and T waves. The simulation, on the positive electrode, appears to have increased the amplitude of the biological QRS complex transients while attenuating the noise present in the signal and, on the negative electrode, the QRS complex amplitudes are inferior in comparison to its twin, and the noise content is amplified.

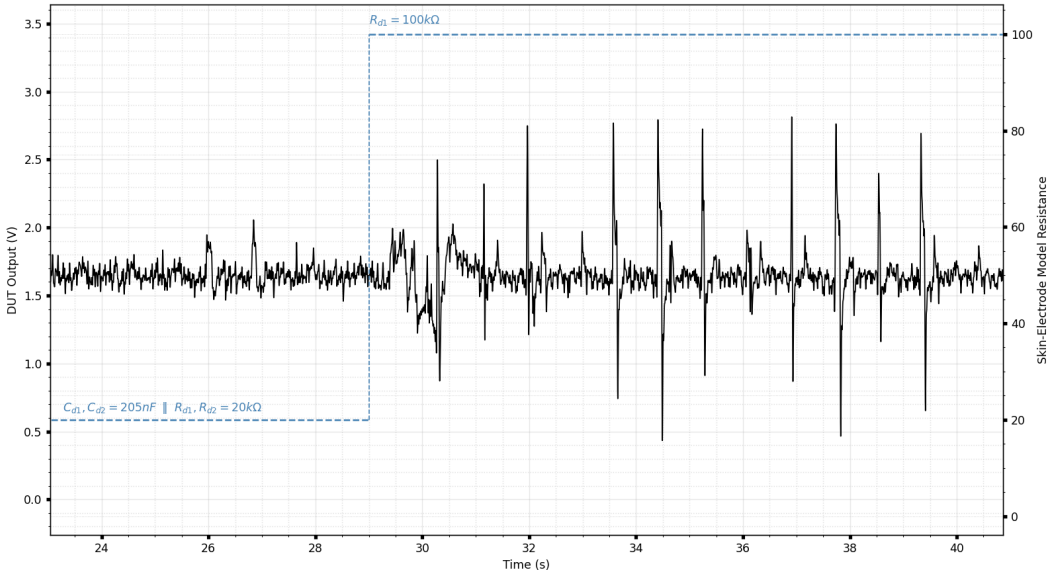


Figure 5.15 Simulation of an abrupt increase in resistance in the positive electrode. At instant 30 seconds, the resistance increases 5 times the base level. Start condition: $C_{d1}, C_{d2} = 205nF||R_{d1}, R_{d2} = 20k\Omega$.

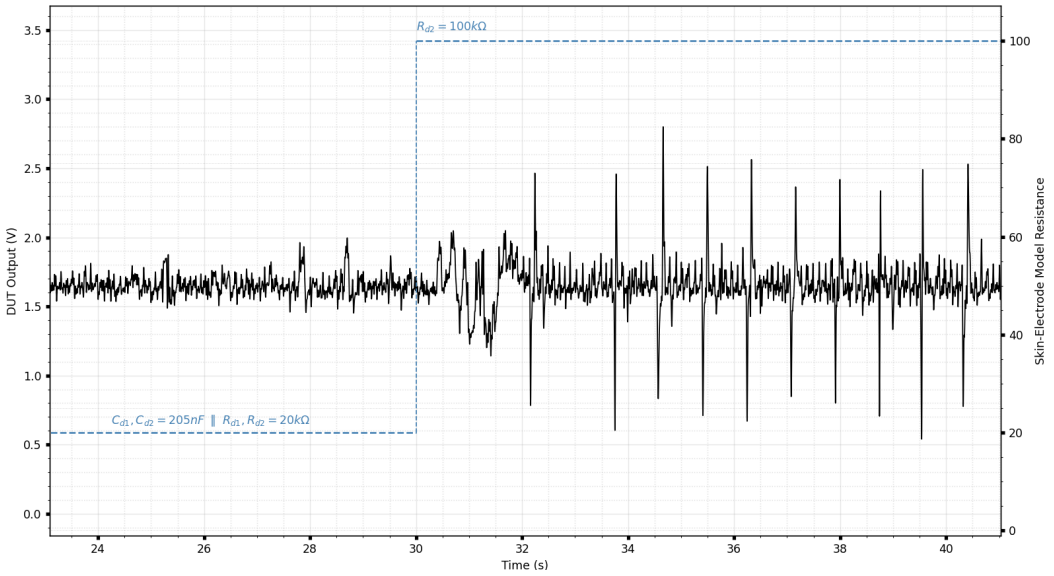


Figure 5.16 Simulation of a rapid increase of resistance in the negative electrode. At instant 30 seconds, the resistance increases 5 times the base level. Start condition: $C_{d1}, C_{d2} = 205nF||R_{d1}, R_{d2} = 20k\Omega$.

The next signals illustrate the individual impact of a rapid capacitance change in the ECG waveform for both electrodes. In both signals, capacitance decreases in a short time frame, while the remaining components of the model maintain their values. In Figure 5.17, the positive electrode presents multiple transients and requires nearly 6 seconds to restore the signal and the baseline, after which the components of the ECG wave are clearly visible. The negative electrode simulation, presented in Figure 5.18, shows a faster signal recovery time (about 2 seconds). The noise content is significant; the QRS waves are visible; however, T wave appears obscured by the multiple artifacts, similar to P wave, although this component can be identified in some pulses.

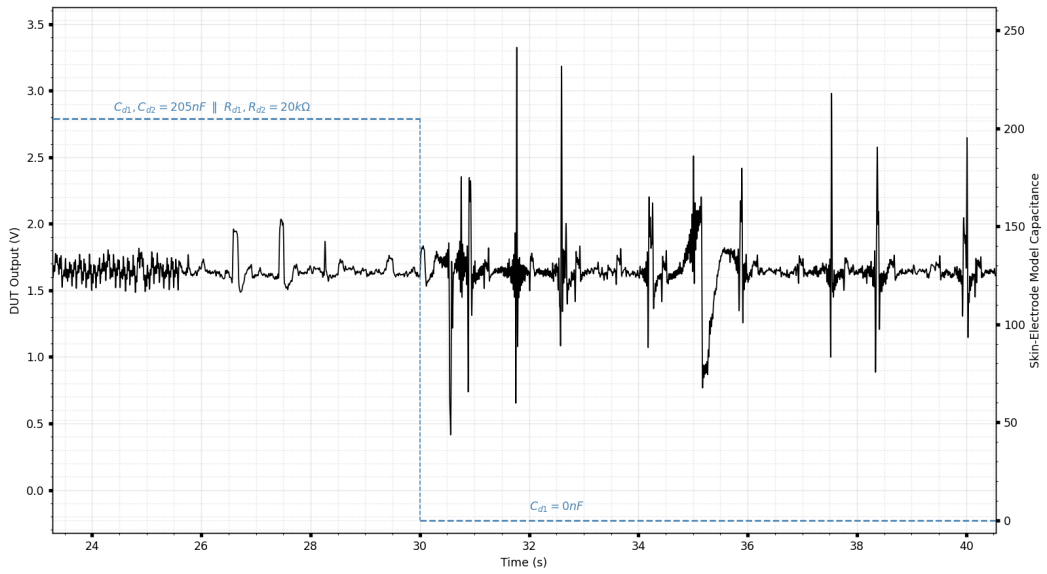


Figure 5.17 Simulation of a abrupt decrease of capacitance in the positive electrode. Start condition: $C_{d1}, C_{d2} = 205nF||R_{d1}, R_{d2} = 20k\Omega$.

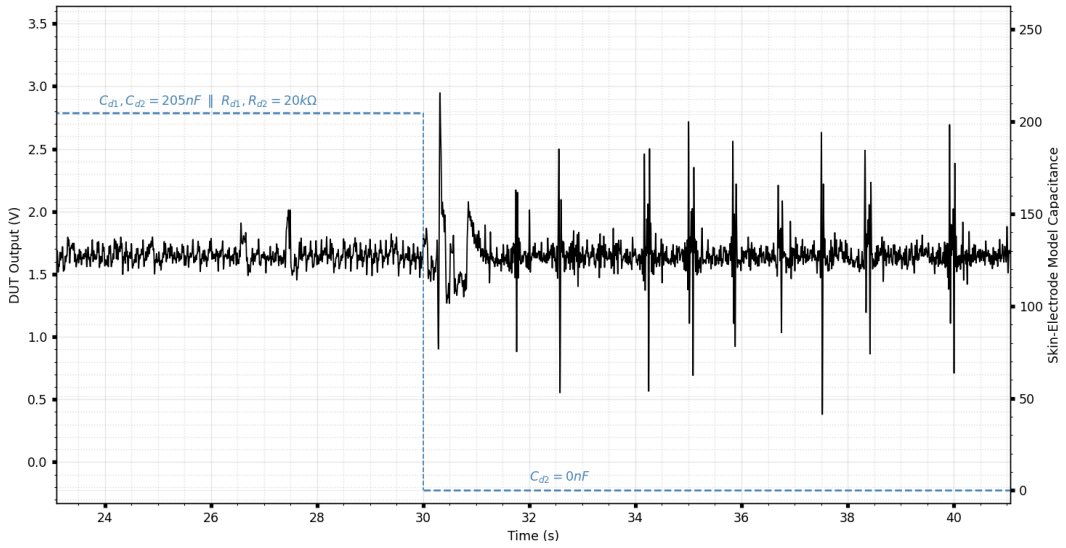


Figure 5.18 Simulation of a swift decrease of capacitance in the negative electrode. Start condition: $C_{d1}, C_{d2} = 205nF||R_{d1}, R_{d2} = 20k\Omega$.

The lead-off events, associated with a complete loss of contact of the electrode with the surface area, were simulated with the development of an impedance profile comprising of a time frame with maximum resistance and minimum capacitance. In Figure 5.19, a simulation of positive electrode disconnection is performed. The contact is lost at approximately 27.5 seconds, following transient and high amplitude waves. The electrode is 'reconnected' after 8 seconds, reestablishing the signal acquisition and waveform amplitude. The acquisition device takes about 1 second to restore the signal. The negative electrode lead-off emulation is illustrated in Figure 5.20. The signal is restored after 1.5 seconds; the lead-off ECG waveform exhibits multiple transients and decreased P and T wave clarity due to increased noise levels compared

to Figure 5.19. Furthermore, the device requires an additional 0.5 seconds to reestablish the previous ECG waveform.

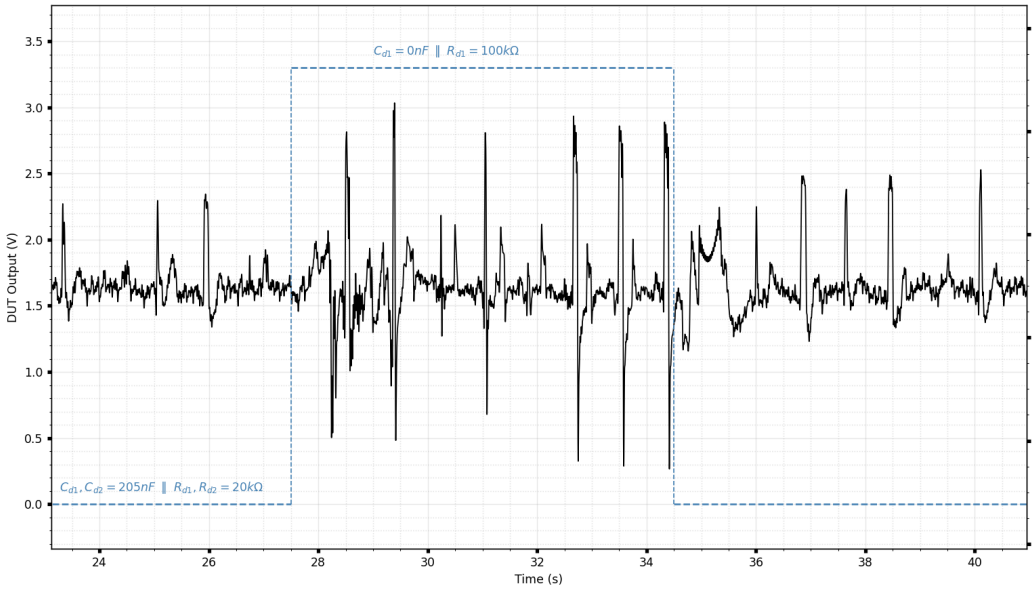


Figure 5.19 Simulation of a positive electrode disconnection. Lead-off occurs at instant 30 seconds and maintains throughout the signal reproduction. Start condition: $C_{d1}, C_{d2} = 205nF||R_{d1}, R_{d2} = 20k\Omega$.

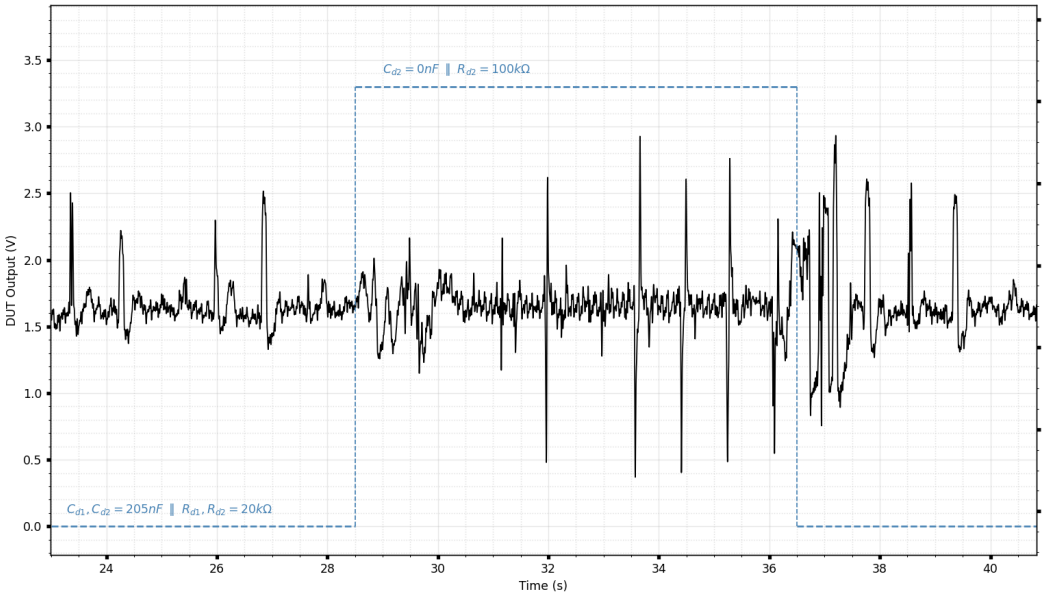


Figure 5.20 Simulation of a negative electrode disconnection. Lead-off occurs at instant 30 seconds and maintains throughout the signal reproduction. Start condition: $C_{d1}, C_{d2} = 205nF||R_{d1}, R_{d2} = 20k\Omega$.

5.4.2 LTSpice Simulation Results

Figure 5.21 shows the circuit implemented in SPICE simulations using the CardioWheel Front-End Equivalent Model provided (*CW-FE Eq Model*). The operational amplifiers in the circuit are modeled as ideal components.

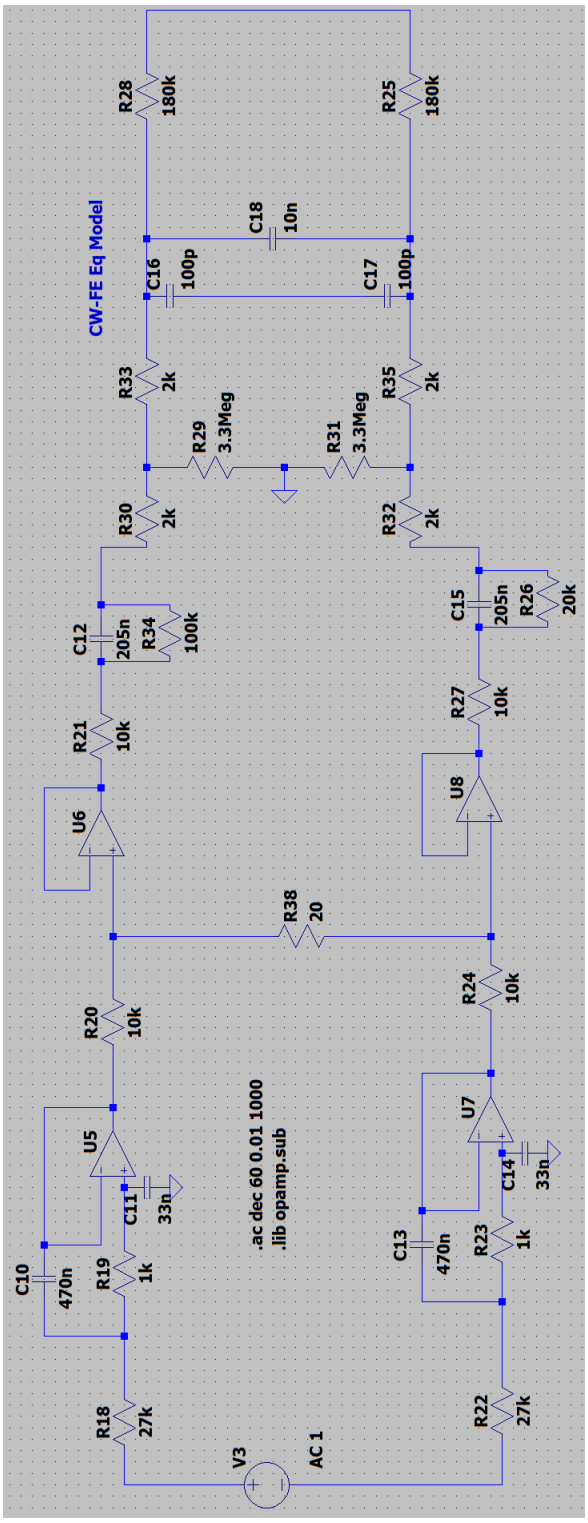


Figure 5.21 Equivalent generation circuit implemented in LTSpice. The circuit incorporates the required SPICE directives for ideal operational amplifier modeling and step parameter definitions for variable analysis.

Multiple simulations were performed in LTSpice using AC analysis. Therefore, to determine the frequency response of the circuit, and related parameters, the circuit was exposed to a frequency sweep from the source. The resulting Bode plots are presented in Figure below.

The implemented *CW-FE Eq Model* creates a passive high-pass shelving filter in the

positive and negative electrodes. In the positive node, it consists of a parallel C12 and R34, R30 and R29. In the negative node, it consists of a parallel C15 and R26, R32 and R31.

First, the impact of resistance on the resulting signals associated with the manipulation of both leads (Figures 5.15 and 5.16) was assessed. Subsequently, the impact of capacitance on the resulting signals associated with the manipulation of both leads (Figures 5.17 and 5.18) was simulated.

In Figure 5.22, the Bode plot presents a stable attenuation until $\approx$ 2 Hz, following a slope and a peak gain of $\approx$ 3 dB at 58.4 Hz. Afterwards, the frequency response drops due the reconstruction filter. Low frequency components are positioned in the slope region of the frequency response; it is expected that T spectrum presents the smallest gain, followed by P wave. QRS complex spectrum is majorly positioned in the 'boost' region of the filter, thus presents higher amplitudes (Figure 5.15 and 5.16).

The Bode plot presented in Figure 5.23 illustrates an extremely high attenuation of $\approx$ -93 dB, associated with a null capacitance that decreases linearly with the frequency until $\approx$ 200 Hz to a value of $\approx$ -11 dB. In an ECG, low frequency components, such as P and T waves, may be removed due the high attenuation created by the rapid decrease of capacitance to a null value, as shown by Figure 5.17 and, particularly, Figure 5.18. In addition, the frequency spectrum is exposed to decreasing attenuation values and, therefore, due the difference in attenuation values between various components, high frequency components (QRS complex) will present higher amplitudes in the waveform.

The impact of impedance in the signal is presented in Figure 5.24, illustrating a stable attenuation until $\approx$ 10 Hz, following a slope that has an impact in P wave spectra and a boost shelf affecting high frequency components overall, thus reflected on the signal in Figure 5.13.

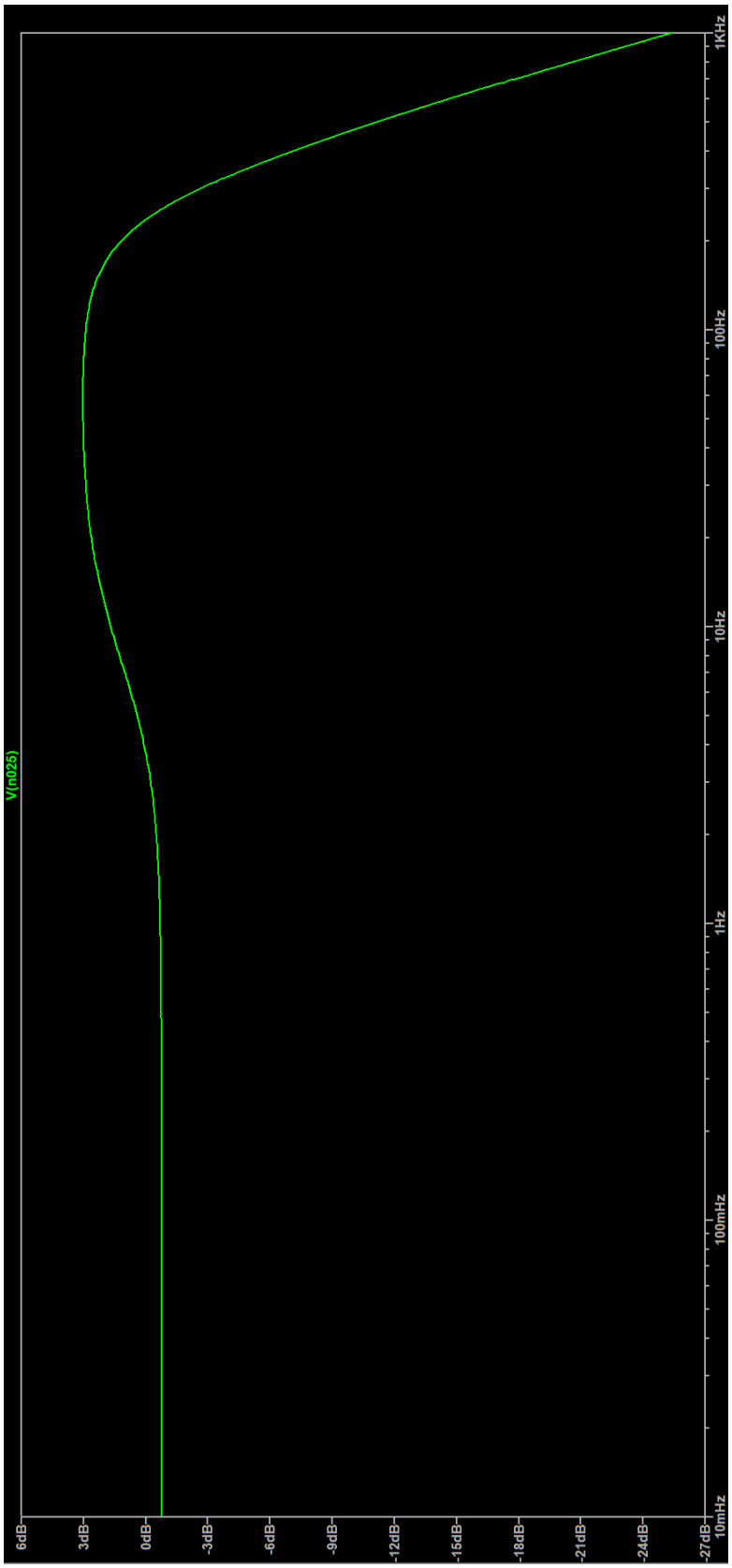


Figure 5.22 SPICE simulation of the frequency response of the signal generation circuit, presenting the magnitude. The results illustrate the behavior of the circuit after the increase of resistance, and were performed for the positive lead.

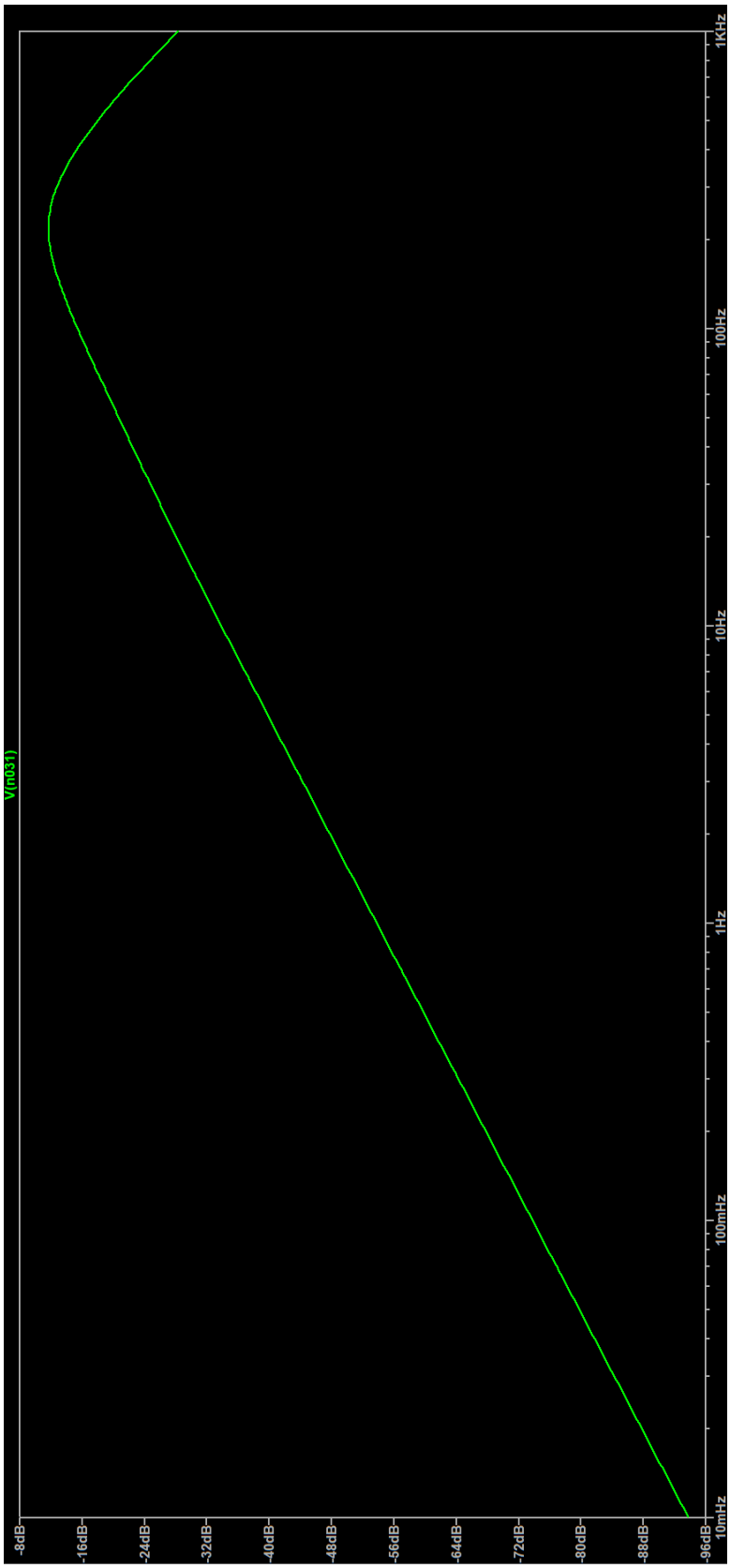


Figure 5.23 SPICE simulation of the frequency response of the signal generation circuit, presenting the magnitude. The results illustrate the behavior of the circuit after the decrease of capacitance, and were performed for the positive lead.

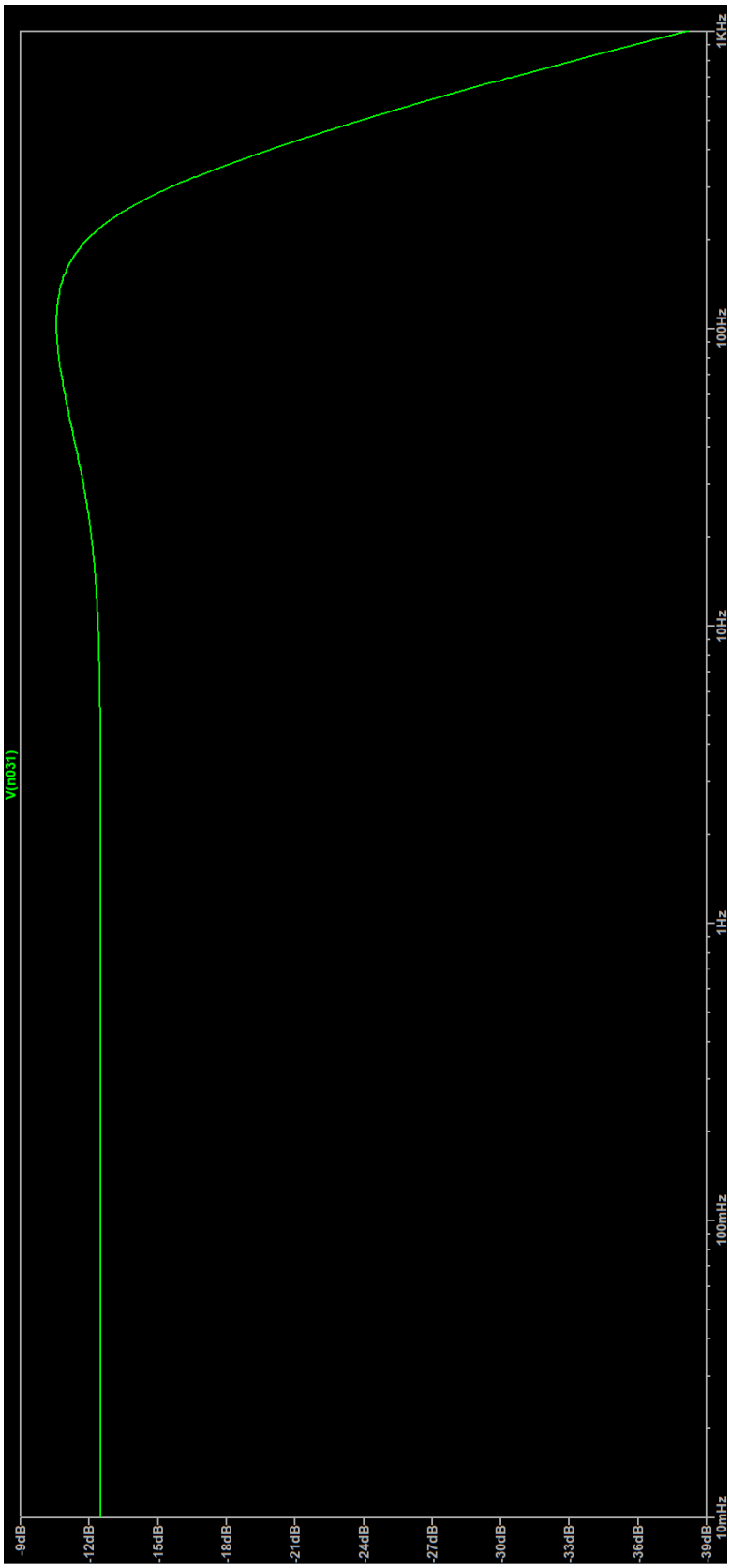


Figure 5.24 SPICE simulation of the frequency response of the signal generation circuit, presenting the magnitude. The results illustrate the behavior of the circuit after the increase of impedance, and were performed for the positive lead.

Chapter 6

Conclusions

Advances in physiological signal acquisition technologies drive the continuous development of systems with increasingly accurate representation of bioelectric interactions and tissue-electrode interface dynamics. However, it is impossible to consider tissue as a homogeneous material, specially living tissue. The human body is composed of numerous tissue types, with different bioelectrical properties; in addition, these values dependent on frequency, temperature, water content, blood perfusion, among other properties [59]. The second property plays a crucial role in ECG acquisitions, particularly in the surface area in contact between the electrode and the skin, being the scope of the project.

6.1 Main Findings

The performance of the analog section of signal generation circuit is within the expected, and simulated, outcome, demonstrating the stability of this component in the system. The digital potentiometer demonstrated a fairly linear voltage change over step. Despite the voltage discrepancies, the results were measured using jump wires and bread board, susceptible to mechanical disturbances and consequent voltage oscillations.

The production time execution was excellent for the developed system, and synchronous with the consumer task. However, the development of SD card in a shared bus with peripherals responsible for signal generation can be considered a design flaw, as the producer and consumer main processes, and related peripherals, should be separated. The overall performance should be improved. The system presents good results for long-term signal generation, usually the purpose of simulation devices, in comparison to short term signals.

In conclusion, the system demonstrated the impact of impedance manipulation through resistance and capacitance variations at individual electrodes, illustrating the critical importance of skin-electrode interface electrical properties in the reliability of acquired signals, extending to the maintenance of these equipments, clinical decision making or training of clinical professionals.

This project intends to be a contribution to further development of this system; the *firmware* was developed to promote expansion of the number of threads and allocated resources, the

command structure can hold additional commands, controlled with the computer interface, and multiple data structures can be established according to the data organization sent by the host.

6.2 Limitations

The main obstacle associated with this work was the initial unfamiliarity associated with the main requirements of the system to develop, requiring extensive research in areas such as signal processing and filtering, embedded systems architecture and an extended time of programming.

Hardware development was straightforward. The signal generation circuit follows the design from [33, 34] with component updates for obsolete parts and I/O expander series change to implement a standardized SPI communication protocol across the board. Both SPI bus provided by ESP32 extend to the signal generation circuit and the SD card. The daisy-chain power supply section was well design; however, the necessity of connecting the board to the programmer defeated its purpose by providing power through the programming connection. Finally, the board features two button slots associated with boot mode (BOOT) and reset (RST); the absence of the auto-flash circuit component requires the user to manually restart the device and select the download mode. While this limitation does not affect the current system, which requires a USB/UART bridge controller for computer interfacing, future Bluetooth integration could compromise the 'hands-free' operation of a system designed without reset functionality.

The board required corrections along the development of the code and concurrent testing. The first obstacle was assessing the RX/TX lines cross-over between the ESP-PROG connector and the MCU. One of the most significant changes was addressing the I/O expanders using hardware address bits, removing the necessity of two CS lines, and externally biasing RESET pin, required for proper function. Additionally, the $10k\Omega$ digital potentiometer MCP4251 was replaced by $100k\Omega$ MCP42100 due to the presence of higher resistance values in literature. A substantial firmware improvement involved replacing the binary semaphore in the consumer task, previously used for SPI operation management upon ISR events, with a task notification mechanism functioning as a counter. This approach mitigated sample loss encountered in the previous implementation while maintaining signal reproduction duration within expected parameters.

The main improvements address the limitations above. Regarding the computer interface, user experience could be improved by implementing a GUI to replace menu-driven interactions, potentially simplifying code development while providing additional functionality. A critical improvement for the Python codebase involves comprehensive restructuring and refractoring through object-oriented design, organizing functionality into dedicated classes for communication protocols, system state management, signal processing, and SD card operations to improve code maintainability, extensibility, and modularity. The *firmware* also presents a window for optimization and further organization of different structures. *ESP-IDF* presents a large amount of libraries and methods that can be used to improve overall code efficiency, reduce the size of the program and incorporate IoT functionalities.

An additional objective that was not fully developed was the GUI, shown in Appendix F. This approach would allow a better flow of operations between the user, the interface and the embedded system. In addition, WFDB for Python includes functions that allow a direct view of the existing files in the selected database; the integration of widgets removes the necessity of multiple menu-driven interactions, simplifying the code development, extensibility and avoid repeated code blocks.

6.3 Future Work

The current stage of the system enables the user to retrieve multiple records from PhysioNet, design the intended impedance profile for each record and send to the simulation device for storage and later reproduction. Furthermore, the system provides the user with useful SD management functions, eliminating the need for physical card removal.

The development of a Python codebase GUI program that promotes maintainability, extensibility, and modularity is critical for further improvement of the system. The WFDB Python package provides useful functions that could be used in the user interface to select available ECG record among different databases. Afterwards, IoT features can be introduced in the *firmware* allowing 'hand-free' operation. Additionally, the device could be integrated into an application that enables direct playback of signals stored on the SD card.

IoT system integration along with a well-structured GUI, opens the possibility of managing multiple ECG simulation boards with a single interface, creating a network of ECG simulators. In clinical environments, individual maintenance of numerous ECG acquisition systems could be improved through simultaneous multi-device testing.

Appendix A

Computer Interface - Python Software Code

A.1 Packages & Main Variables

```
import struct
import time
import numpy as np
import serial
import wfdb
import random
from scipy.interpolate import interp1d
import scipy.signal as signal
from matplotlib.widgets import Button
import matplotlib.pyplot as plt
import pandas as pd

CMD_ECG_START = 0
CMD_ECG_STOP = 1
CMD_ECG_DATA = 2
CMD_ECG_OUT = 3
CMD_CONFIG_UPDATE = 4
CMD_SD_LIST = 5
CMD_SD_DELETE = 6
CMD_SD_RENAME = 7
CMD_SD_INFO = 8


class ecgSystem: (...)


class SignalDesigner: (...)


def main():

    test = ecgSystem("COM4")  # connect w/ device on port COMx
    test.main_menu()


if __name__ == "__main__":
    main()
```

A.2 *ECG System* Class

A.2.1 Constructor & Configurations

```
def __init__(self, port="COM14", baudrate=115200):
    try:
        self.ser = serial.Serial(port, baudrate, timeout=1)
        time.sleep(2)
        print(f"Connected to {port}")
    except Exception as e:
        print(f"Connection failed: {e}")
        self.ser = None

    self.designed_signals = None

current_config = {
    # System settings
    'file_name': '/sdcard/teste.bin',   # /sdcard/filename.bin
    'sampling_frequency': 360,  # auto-updated from ECG record
    'resampling_frequency': 360,    # user selection

    # ECG signal settings
    'record': '100',        # Physionet record
    'database': 'mitdb',    # Physionet database
    'start_sec': 0,         # start time
    'duration_sec': 10,     # duration
    'channel': 0,           # channel (changes w/ database)

    # Test functions
    'width': 50,            # sawtooth/triangle wave
    'duty_cycle': 50,       # square wave
    'frequency': 1,
    'samples': 3600  # Default samples (10 sec * 360 Hz)
}
```

A.2.2 Menus & Submenus

```
def main_menu(self):
    if not self.ser:
        return

    while True:
        print("\n Configuration Options:")
        print("1. Set signal parameters")
        print("2. SD Card file management")
        print("3. Read serial single-shot")
        print("4. Test files")
        sub_choice = input("Choose (0-6): ").strip()
        match sub_choice:
            case '1':
                self.ecg_parameters_menu()
            case '2':
                self.sd_card_management_menu()
            case '3':
                    self.read_response()
            case '4':
                self.test_management_menu()
            case _:
                continue
```

```
def ecg_parameters_menu(self):
    while True:
        print("\n Signal Operations:")
        print("1. Start signal recording")
        print("2. Stop signal reproduction")
        print("3. Change ECG parameters")
        print("4. Read single-shot")
        print("0. Back to configuration menu")
        signal_choice = input("Choose (0-3): ").strip()
        match signal_choice:
            case '1':
                self.send_single_command(CMD_ECG_START)
            case '2':
                self.send_single_command(CMD_ECG_STOP)
            case '3':
                ecg_keys = ['file_name', 'resampling_frequency', '
                    database', 'record', 'start_sec', 'duration_sec', '
                    channel']
                config_changed = self.change_dict_parameters(ecg_keys)

                design_signals = input("\n Impedance design? (y/N): ").
                    strip().lower()
                if design_signals == 'y':
                    duration = self.current_config['duration_sec'] -
                        self.current_config['start_sec']
                    samples = int(duration * self.current_config['
                        resampling_frequency'])
                    self.design_signals_gui(
                        samplingFreq=self.current_config['
                            resampling_frequency'],
                        total_samples=samples
                    )

                if config_changed:
                    confirm = input("\nSend signal? (y/N): ").strip().
                        lower()
                    if confirm == 'y':
                        self.send_single_command(CMD_ECG_START)
                        time.sleep(0.1)
                        try:
                            normalized_signal = self.load_ecg_data(
                                self.current_config['record'],
                                self.current_config['database'],
                                self.current_config['start_sec'],
                                self.current_config['duration_sec'],
                                self.current_config['channel']
                            )

                            if normalized_signal:
                                self.send_stream(CMD_ECG_DATA,
                                    normalized_signal, chunk_size=500)
                                time.sleep(0.01)
                                self.send_single_command(CMD_ECG_STOP)
                        except Exception as e:
                            pass
                    else:
                        print(" Signal not sent.")
            case '4':
                self.read_response()
            case '0':
                break
            case _:
                continue
```

```
def sd_card_management_menu(self):
    while True:
        print("\n SD Card Operations:")
        print("1. File operations menu")
        print("2. Read response single-shot")
        print("0. Back to configuration menu")

        sd_choice = input("Choose (0-2): ").strip()
        match sd_choice:
            case '1':
                file_list = self.get_sd_file_list()
                if file_list:
                    self.file_operations_menu(file_list)
                else:
                    print("No files found/failed to retrieve files")
            case '2':
                self.read_response()
            case '0':
                break
            case _:
                continue
```

```
def test_management_menu(self):
    while True:
        print("\n Test options:")
        print("1. Sine wave")
        print("2. Sawtooth/triangle wave")
        print("3. Square wave")
        print("4. Change wave parameters")
        print("5. Read single-shot")
        print("6. SD Card management menu")
        print("7. Cardiowheel file (PC hardcoded)")
        print("0. Back to main menu")

        wave = input(f"Select wave to test: ").strip()
        match wave:
            case '1':
                self.generate_and_send_test_wave('sine')
            case '2':
                self.generate_and_send_test_wave('sawtooth')
            case '3':
                self.generate_and_send_test_wave('square')
            case '4':
                self.change_dict_parameters(['file_name', 'frequency', '
                    start_sec', 'duration_sec', 'sampling_frequency'])
            case '5':
                self.read_response()
            case '6':
                self.sd_card_management_menu()
            case '7':
                self.generate_and_send_test_wave('cardiowheel')
            case '0':
                break
            case _:
                continue
```

A.2.3 Utility Methods

```
def normalize_data(self, ecg_data):
    if not ecg_data:
        return None
```

```
    ecg_array = np.array(ecg_data)      # convert list to Numpy array
    ecg_max_value = np.max(ecg_array)   # calculate max of array
    ecg_min_value = np.min(ecg_array)   # calculate min of array

    # Normalize the data to range [0, 1]
    if ecg_max_value != ecg_min_value:
        normalized_data = (ecg_array - ecg_min_value) / (ecg_max_value -
            ecg_min_value)
    else:
        normalized_data = np.zeros_like(ecg_array) # avoid zero-division

    return normalized_data.tolist()     # return list
```

```
def send_single_command(self, command):
    if not self.ser:
        print("No connection")
        return

    # send a command without data
    header = self.create_stream_header(command, 0)
    bytes_sent = self.ser.write(header)
    time.sleep(0.1)
    self.read_response()
```

```
def read_response(self):
    if self.ser and self.ser.in_waiting > 0:
        response = self.ser.read(self.ser.in_waiting).decode('utf-8',
           errors='ignore')
        print(f"ESP32: {response.strip()}")
```

```
def send_configuration(self):
    try:
        config_data =
            struct.pack('<64sI',
                        self.current_config['file_name'].encode('utf-8')
                           .ljust(64, b'\0'),
                        self.current_config['sampling_frequency'])

        # create header w/ 1 data pack information
        header = self.create_stream_header(CMD_CONFIG_UPDATE, 1)

        if self.ser and self.ser.is_open:
            self.ser.write(header)      # send header
            self.ser.write(config_data)    # send data

            time.sleep(0.1)
            self.read_response()
        else:
            print(" Serial port not open")

    except Exception as e:
        print(f" Error sending configuration: {e}")
```

```
def change_dict_parameters(self, keys_to_change):

    try:
        new_config = {}     # container for new configs
        print(" Press enter to skip\n")

        for key in keys_to_change:
            match key:
```

```
case 'file_name':
    new_file = input(f" File path [{self.current_config
       ['file_name']}]: ").strip()
    if new_file:
        if new_file.startswith('/sdcard/') and new_file.
           endswith('.bin'):
            new_config['file_name'] = new_file
        else:
            print( ath must start with '/sdcard/' and
               end with '.bin', keeping current")

case 'resampling_frequency':
    new_fs = input(f" Resampling frequency [{self.
       current_config['resampling_frequency']}]: ").
       strip()
    if new_fs:
        new_config['resampling_frequency'] = int(new_fs)
        self.current_config['sampling_frequency'] =
           new_config['resampling_frequency']

case 'record':
    new_record = input(f" Record [{self.current_config['
       record']}]: ").strip()
    if new_record:
        new_config['record'] = new_record

case 'database':
    new_database = input(f" Database [{self.
       current_config['database']}]: ").strip()
    if new_database:
        new_config['database'] = new_database

case 'start_sec':
    start_input = input(f" Start time (seconds) [{self.
       current_config['start_sec']}]: ").strip()
    if start_input:
        new_start = int(start_input)
        if new_start >= 0:
            new_config['start_sec'] = new_start

case 'duration_sec':
    duration_input = input(
        f" Duration (seconds) [{self.current_config['
           duration_sec']}]: ").strip()
    if duration_input:
        new_duration = int(duration_input)
        start_time = new_config.get('start_sec', self.
           current_config['start_sec'])
        if new_duration > start_time:
            new_config['duration_sec'] = new_duration

case 'channel':
    channel_input = input(f" Channel [{self.
       current_config['channel']}]: ").strip()
    if channel_input:
        new_config['channel'] = int(channel_input)

case 'frequency':
    freq_input = input(f" Wave frequency Hz [{self.
       current_config.get('frequency', 1.0)}]: ").strip
       ()
    if freq_input:
        freq_val = int(freq_input)
        if freq_val > 0:
```

```
                            new_config['frequency'] = freq_val

                    case 'sampling_frequency':
                        sampling_input = input(f" Sampling frequency [{self.
                            current_config['sampling_frequency']}]: ").strip
                            ()
                        if sampling_input:
                            freq_val = int(sampling_input)
                            if freq_val > 0:
                                new_config['sampling_frequency'] = freq_val

                    case 'width':
                        width_input = input(f" Width % [{self.current_config
                            ['width']}]: ").strip()
                        if width_input:
                            width_val = int(width_input)
                            if 0 <= width_val <= 100:
                                new_config['width'] = width_val

                    case 'duty_cycle':
                        duty_input = input(f" Duty cycle % [{self.
                            current_config['duty_cycle']}]: ").strip()
                        if duty_input:
                            duty_val = int(duty_input)
                            if 0 <= duty_val <= 100:
                                new_config['duty_cycle'] = duty_val
                    case _:
                        print(f" Unknown parameter: {key}")

        if new_config:
            print(f"\n Changes to apply:")
            for key, value in new_config.items():
                print(f"   {key}: {value}")

            confirm = input("\nApply these settings? (y/N): ").strip().
                lower()
            if confirm == 'y':
                self.current_config.update(new_config)  # update configs
                if any(key in new_config for key in ['start_sec', '
                    duration_sec', 'sampling_frequency']):
                    duration = self.current_config['duration_sec'] -
                        self.current_config['start_sec']
                    self.current_config['samples'] = int(duration * self
                        .current_config['sampling_frequency'])
                    print(f" Updated samples: {self.current_config['
                        samples']}")
                self.send_configuration()
                return True
            else:
                return False        # configuration not sent
        else:
            return False    # no changes made
    except Exception as e:
        print(f" Error: {e}")
        return False
```

A.2.4 ECG Records - Physionet

```
def load_ecg_data(self, record_name='100', database='mitdb/', start_time
    =0, end_time=10, signal_number=0, chunk_size=500):

    try:
```

```
        # load header and update sampling frequency
        header = wfdb.rdheader(record_name, pn_dir=database)
        fs = int(header.fs)
        if fs != self.current_config['sampling_frequency']:
            self.current_config['sampling_frequency'] = fs

        # update start/end time configurations
        start_sample = int(start_time * self.current_config['
            sampling_frequency'])
        end_sample = int(end_time * self.current_config['
            sampling_frequency']) if end_time else header.sig_len

        # extract record
        record = wfdb.rdrecord(
            record_name,
            pn_dir=database,
            sampfrom=start_sample,
            sampto=min(end_sample, header.sig_len),
            channels=[signal_number]
        )

        signal_data = record.p_signal.flatten()

        # resample to user set frequency
        confirm = input("Resample signal? (y/N): ").strip().lower()
        if confirm == 'y':
            if (self.current_config['resampling_frequency'] != 0 and
                    self.current_config['resampling_frequency'] != self.
                        current_config['sampling_frequency']):

                original_time = np.linspace(0, len(signal_data) / self.
                    current_config['sampling_frequency'], len(signal_data
                    ))

                duration = len(signal_data) / self.current_config['
                    sampling_frequency']
                target_samples = int(duration * self.current_config['
                    resampling_frequency'])
                new_time = np.linspace(0, duration, target_samples)

                interpolator = interp1d(original_time, signal_data, kind
                    ='cubic', fill_value='extrapolate')
                signal_data = interpolator(new_time)

                self.current_config['sampling_frequency'] = self.
                    current_config['resampling_frequency']
        else:
            print("Signal not resampled")

        # update samples in configs
        self.current_config['samples'] = len(signal_data)

        normalized_signal = self.normalize_data(signal_data.tolist())
        if normalized_signal is None:
            print(" Failed to normalize ECG data")
            return

        return normalized_signal

    except Exception as e:
        print(f" Error loading ECG data: {e}")
        return None
```

A.2.5 Transmission & Data Packing

```
def send_stream(self, command, ecg_data, chunk_size=200):
    if not self.ser:
        return  # no connection

    digipot1_values, digipot2_values, io_exp1_pins, io_exp2_pins = self.
       get_designed_signal_values()

    total_samples = len(ecg_data)

    try:
        for i in range(0, total_samples, chunk_size):
            chunk_ecg = ecg_data[i:i + chunk_size]
            chunk_dig1 = digipot1_values[i:i + chunk_size]
            chunk_dig2 = digipot2_values[i:i + chunk_size]
            chunk_io1 = io_exp1_pins[i:i + chunk_size]
            chunk_io2 = io_exp2_pins[i:i + chunk_size]

            chunk_num = (i // chunk_size) + 1
            total_chunks = (total_samples + chunk_size - 1) //
               chunk_size

            stream_packet = self.create_stream_packet(command, chunk_ecg
               , chunk_dig1, chunk_dig2, chunk_io1, chunk_io2)

            if stream_packet:
                bytes_sent = self.ser.write(stream_packet)
                self.read_response()
                time.sleep(0.5)

    except Exception as e:
        print(f" Error during transmission: {e}")

    self.current_config['resampling_frequency'] = 0
    print(" Transmission completed")
```

```
def create_stream_packet(self, command, ecg_data_mv, digipot1, digipot2,
    io_exp1, io_exp2):

    # command stream w/ no data
    if not ecg_data_mv:
        return self.create_stream_header(command, 0)

    # data present: [HEADER][DATA 1]...[DATA N]
    n_packets = len(ecg_data_mv)
    stream_data = bytearray()

    # add stream header
    header = self.create_stream_header(command, n_packets)
    stream_data.extend(header)      # add to byte array

    # add ECG data packets
    for i in range(n_packets):
        data_packet = self.create_data_packet(
            ecg_data_mv[i],
            digipot1_value=digipot1[i],
            digipot2_value=digipot2[i],
            io_exp1_pins=io_exp1[i],
            io_exp2_pins=io_exp2[i]
        )
        stream_data.extend(data_packet)     # add to byte array
```

```
    return bytes(stream_data)       # return byte array
```

```
def create_stream_header(self, command, n_packets):
    # stream header [start_2][n_packs_2][cmd_1][end_2]
    header = bytearray()
    header.extend(struct.pack('<H', 0xBB66))  # start marker
    header.extend(struct.pack('<H', n_packets))  # number of packets
    header.append(command)  # command
    header.extend(struct.pack('<H', 0x66BB))  # end marker
    return bytes(header)
```

```
def create_data_packet(self, ecg_sample_mv, digipot1_value=0,
    digipot2_value=128, io_exp1_pins=None, io_exp2_pins=None):

    # convert ECG values float -> int -> DAC range
    ecg_dac = int(ecg_sample_mv * 65535)
    ecg_bytes = struct.pack('<H', ecg_dac)  # convert DAC to HEX

    # generate digipot command bytes
    digipot1_value = self.mcp4251_command(address=0, operation='write',
        data=digipot1_value)
    digipot2_value = self.mcp4251_command(address=1, operation='write',
        data=digipot2_value)

    # generate io expander command bytes
    if io_exp1_pins is None:
        io_exp1_pins = [1, 2, 3]  # Default pins
    if io_exp2_pins is None:
        io_exp2_pins = [4, 5, 6]  # Default pins

    io_exp1_value = self.mcp23s08_commmand('OLAT', device_addr=0, data=
        io_exp1_pins)
    io_exp2_value = self.mcp23s08_commmand('OLAT', device_addr=1, data=
        io_exp2_pins)

    # build pack for a sample
    packet = bytearray()
    packet.extend(ecg_bytes)    # 2 bytes
    packet.append(digipot1_value[1])    # 1 byte
    packet.append(digipot2_value[1])    # 1 byte
    packet.append(io_exp1_value[2])    # 1 byte
    packet.append(io_exp2_value[2])    # 1 byte

    return bytes(packet)
```

A.2.6 Digital Potentiometer and I/O Expander Commands

```
def mcp42100_command(self, command='write', pots='both', data=0x80):

    # Command codes (bits 4-5 of command byte)
    command_bits = {
        'nop': 0b00,  # no operation
        'write': 0b01,  # write data
        'shutdown': 0b10,  # shutdown mode
        'nop2': 0b11  # no operation
    }

    # Potentiometer selection (bits 0-1 of command byte)
    pot_select = {
        'none': 0b00,  # no operation
        'pot0': 0b01,  # wiper 0
```

```
        'pot1': 0b10,  # wiper 1
        'both': 0b11   # both wipers
    }

    command_byte = (command_bits[command] << 4) | pot_select[pots]

    # Data byte is the 8-bit wiper position
    data_byte = data & 0xFF

    return [command_byte, data_byte]
```

```
def mcp23s08_commmand(self, register_addr, operation='write',
    device_addr=0, data=None):

    # MCP23S08 register addresses
    REGISTERS = {
        'IODIR': 0x00,  # I/O Direction Register (1=input, 0=output)
        'IPOL': 0x01,  # Input Polarity Register
        'GPPU': 0x06,  # Pull-up Resistor Register
        'GPIO': 0x09,  # GPIO Register
        'OLAT': 0x0A  # output latch register
    }

    if register_addr.upper() in REGISTERS:
        reg_addr = REGISTERS[register_addr.upper()]
    else:
        raise ValueError(f"Invalid register name: {register_addr}")

    # command byte: [01000][A1][A0][R/W]
    mask_byte = 0x40  # 01000000
    address_bits = (device_addr & 0x03) << 1  # A1, A0 shifted to bits
        2-1
    operation_bit = 0

    command_byte = mask_byte | address_bits | operation_bit

    if data is None:
        raise ValueError("Data must be provided for write operations")

    data_byte = self.pack_pins(data)

    return [command_byte, reg_addr, data_byte]
```

```
def pack_pins(self, pin_list):
    # if the pin is in list the bit is set high
    result = 0
    for pin in pin_list:
        if 0 <= pin <= 7:
            result |= (1 << pin)
    return result
```

A.2.7 SD Card Management

```
def file_operations_menu(self, file_list):
    if not file_list:
        return

    while True:
        for i, file_info in enumerate(file_list):
            print(f"   {i + 1}. {file_info['display']}")
```

```
        print("1. Play selected file")
        print("2. Delete selected file")
        print("3. Rename selected file")
        print("4. Get detailed info for selected file")
        print("5. Refresh file list")
        print("6. Read response single-shot")
        print("0. Back to SD menu")

        choice = input("Choose operation (0-6): ").strip()
        match choice:
            case '1':
                self.play_selected_file(file_list)
            case '2':
                self.delete_selected_file(file_list)
            case '3':
                self.rename_selected_file(file_list)
            case '4':
                self.get_selected_file_info(file_list)
            case '5':
                # Refresh file list
                new_file_list = self.get_sd_file_list()
                if new_file_list is not None:
                    file_list.clear()
                    file_list.extend(new_file_list)
            case '6':
                self.read_response()
            case '0':
                break
            case _:
                continue
```

```
def play_selected_file(self, file_list):
    selected_file = self.select_file_from_list(file_list, "play")
    if selected_file:
        filename = selected_file['name']

        # update config to use the file
        old_file = self.current_config['file_name']
        old_sampling = self.current_config['sampling_frequency']
        self.current_config['file_name'] = filename

        new_freq = input(f"Enter sampling frequency [{self.
            current_config['sampling_frequency']}]: ").strip()
        if new_freq:
            try:
                self.current_config['sampling_frequency'] = int(new_freq
                    )
                print(f" Sampling frequency updated to {int(new_freq)}
                    Hz")
            except ValueError:
                print(" Invalid frequency value, keeping current")

        # send configuration with new file and sample frequency
        self.send_configuration() # update configs of system

        # Start playback
        play_now = input("Start playback now? (y/N): ").strip().lower()
        if play_now == 'y':
            self.send_single_command(CMD_ECG_OUT)
        else:
            self.current_config['file_name'] = old_file
            self.current_config['sampling_frequency'] = old_sampling
```

```
def delete_selected_file(self, file_list):
    selected_file = self.select_file_from_list(file_list, "delete")
    if selected_file:
        filename = selected_file['name']

        confirm = input(f"  Are you sure you want to delete '{filename
            }'? (y/N): ").strip().lower()
        if confirm == 'y':
            print(f" Deleting file: {filename}")
            self.send_sd_command(CMD_SD_DELETE, filename)

            # Remove from cached list
            if hasattr(self, 'cached_file_list'):
                self.cached_file_list = [f for f in self.
                    cached_file_list if f['name'] != filename]
            file_list[:] = [f for f in file_list if f['name'] !=
                filename]
```

```
def rename_selected_file(self, file_list):
    selected_file = self.select_file_from_list(file_list, "rename")
    if selected_file:
        old_name = selected_file['name']

        # enter name
        new_name = input("Enter new filename (e.g., /sdcard/new_name.bin
            ): ").strip()

        if new_name:
            if new_name.startswith('/sdcard/') and new_name.endswith('.
                bin'):
                self.send_sd_command(CMD_SD_RENAME, old_name, new_name)

                # Update cached list
                selected_file['name'] = new_name
                selected_file['display'] = f"{new_name} ({selected_file
                    ['size']:,} bytes)"

        else:
            print(" No new filename provided")
```

```
def get_sd_file_list(self):
    # Send list command
    self.send_sd_command(CMD_SD_LIST)

    files = []
    try:
        # read and parse file information
        if self.ser and self.ser.in_waiting > 0:
            response_lines = []
            start_time = time.time()

            # see serial monitor for 3 secs
            while time.time() - start_time < 3:
                if self.ser.in_waiting > 0:
                    line = self.ser.readline().decode('utf-8', errors='
                        ignore').strip()
                    if line:
                        response_lines.append(line)
                        print(f"ESP32: {line}")
                else:
                    time.sleep(0.1)
```

```
            # Parse the response, extract filename and size
            for line in response_lines:
                if '/sdcard/' in line and '(' in line and 'bytes)' in
                    line:
                    # Expected format: /sdcard/filename.bin (12 bytes)
                    try:
                        parts = line.strip().split('(')
                        if len(parts) >= 2:
                            filename = parts[0].strip()
                            size_part = parts[1].split(' bytes)')[0]
                            size = int(size_part)

                            files.append({
                                'name': filename,
                                'size': size,
                                'display': f"{filename} ({size:} bytes)"
                            })
                    except (ValueError, IndexError) as e:
                        continue  # skip malformed lines

        # Cache the file list
        self.cached_file_list = files

        if files:
            return files    # files found
        else:
            return []       # files not found

    except Exception as e:
        print(f" Error retrieving file list: {e}")
        return []
```

```
def select_file_from_list(self, file_list, operation_name):
    if not file_list:
        return None     # no files in list

    print(f"\n Select file to {operation_name}:")
    for i, file_info in enumerate(file_list):
        print(f" {i + 1}. {file_info['display']}")

    try:
        selection = input(f"Enter file number (1-{len(file_list)}) or 0
            to cancel: ").strip()
        if selection == '0':
            return None

        file_index = int(selection) - 1
        if 0 <= file_index < len(file_list):
            return file_list[file_index]
    except ValueError:
        print(" Invalid input")
        return None
```

A.2.8 Test Management

```
    def generate_test_wave(self, wave_type):

        # calculate samples from duration and sampling frequency
        duration = self.current_config['duration_sec'] - self.
            current_config['start_sec']
        self.current_config['samples'] = int(duration*self.
            current_config['sampling_frequency'])
```

```
        # create time data
        t = np.linspace(0, duration, self.current_config['samples'])

        match wave_type:
            case 'sine':
                wave_name = "Sine Wave"
                sine_values = np.sin(2 * np.pi * self.current_config['
                    frequency'] * t)
                wave_data = (sine_values * 1000).tolist()

            case 'sawtooth':
                wave_name = "Sawtooth Wave"
                sawtooth_values = signal.sawtooth(2*np.pi*self.
                    current_config['frequency'] * t, self.current_config[
                    'width'] / 100)
                wave_data = (sawtooth_values * 1000).tolist()

            case 'square':
                wave_name = "Square Wave"
                square_values = signal.square(2 * np.pi * self.
                    current_config['frequency'] * t, self.current_config[
                    'duty_cycle'] / 100)
                wave_data = (square_values * 1000).tolist()

            case 'cardiowheel':
                wave_name = "cardiowheel"
                wave_data, fs = self.cardiowheel_txt("C:\\Users\\(...)\\
                    Cardiowheel\\header_cardioID.txt")
                self.current_config['sampling_frequency'] = fs
                self.current_config['samples'] = len(wave_data)

            case _:
                return None, None

        return wave_data, wave_name
```

```
def generate_and_send_test_wave(self, wave_type):

    wave_data, wave_name = self.generate_test_wave(wave_type)

    if wave_data is None:
        return      # invalid selection

    normalized_signal = self.normalize_data(wave_data)

    duration = self.current_config['samples'] / self.current_config['
        sampling_frequency']

    design_now = input(f"\n Impedance design for {wave_name}? (y/N): ").
        strip().lower()
    if design_now == 'y':
        self.design_signals_gui(
            self.current_config['sampling_frequency'],
            self.current_config['samples']
        )

    send_confirm = input(f"\n Send signal {wave_name}? (y/N): ").strip()
        .lower()
    if send_confirm == 'y':
        self.send_single_command(CMD_ECG_START)     # start task
            resources
        time.sleep(0.05)        # delay for MCU process
```

```
        chunk_size = int(input("Enter chunk size (default 200): ").strip
            () or "200")

        try:
            self.send_stream(CMD_ECG_DATA, normalized_signal, chunk_size
                =chunk_size)
            time.sleep(0.5)     # delay for MCU process
            self.send_single_command(CMD_ECG_STOP)      # finish
                SDWriteTask
        except Exception as e:
            print(f" Error during transmission: {e}")
    else:
        print(" Transmission cancelled")
```

A.2.9 *Signal Designer* Related Methods

```
def design_signals_gui(self, samplingFreq=None, total_samples=None):

    # assume a sampling frequency if not given
    if samplingFreq is None:
        samplingFreq = self.current_config['sampling_frequency']

    # calculate time axis w/ samples or time in configs
    if total_samples is None:
        if hasattr(self, 'current_config') and 'samples' in self.
            current_config:
            total_samples = self.current_config['samples']
        else:
            # Calculate from duration
            duration = self.current_config['duration_sec'] - self.
                current_config['start_sec']
            total_samples = int(duration * samplingFreq)

    # retrieve data from designer
    self.designed_signals = self.get_user_signal_values(
        total_samples=total_samples,
        sampling_rate=samplingFreq
    )
```

```
def get_user_signal_values(self, total_samples, sampling_rate):

    fig = plt.figure("Signal Designer", figsize=(10, 6))
    designer = SignalDesigner(total_samples, sampling_rate, fig)

    plt.show(block=True)

    final_data = designer.extract_data

    designer.cleanup()      # delete designer plot
    return final_data
```

```
def get_designed_signal_values(self):

    try:
        signals = self.designed_signals['signals']

        digipot1_values = signals['digipot1']['normalized']  # 0-100
        digipot2_values = signals['digipot2']['normalized']  # 0-100
        io_exp1_values = signals['capacitance1']['normalized']  # 0-255
        io_exp2_values = signals['capacitance2']['normalized']  # 0-255
```

```
        # convert io values to pins
        io_exp1_pins = self.capacitance_to_pins(io_exp1_values)
        io_exp2_pins = self.capacitance_to_pins(io_exp2_values)

        return (digipot1_values.tolist(),
                digipot2_values.tolist(),
                io_exp1_pins,
                io_exp2_pins)
    except (KeyError, TypeError) as e:
        print(f"Error getting signals: {e}")
```

```
def capacitance_to_pins(self, capacitance_values):
    capacitor_values = [1, 2, 4, 8, 16, 33, 64, 128] # cap values in pcb
    sorted_caps = sorted(enumerate(capacitor_values), key=lambda x: x
        [1], reverse=True)
    pin_combinations = []       # combination to create

    for cap_val in capacitance_values:
        selected_pins = []
        if cap_val == 0:
            pin_combinations.append([])
            continue

        # for every cap value -> start with the highest to the lowest
            possible fit
        # each cap number is then append to the list
        for pin, value in sorted_caps:
            if value <= cap_val:
                selected_pins.append(pin)
                cap_val -= value
                if cap_val == 0:
                    break

        pin_combinations.append(sorted(selected_pins))

    return pin_combinations
```

A.3 *Signal Designer* Class

A.3.1 Constructor

```
def __init__(self, total_samples=3600, sampling_rate=360, fig=None):

    self.total_samples = total_samples
    self.sampling_rate = sampling_rate
    self.duration = total_samples / sampling_rate

    # Time axis
    self.t = np.linspace(0, self.duration, self.total_samples)

    # mark points for 4 parameters to design the impedance over time
    self.control_points = {
        'digipot1': [],
        'digipot2': [],
        'capacitance1': [],
        'capacitance2': []
    }

    # current mode
```

```
    self.current_mode = 'digipot1'
    self.mode_colors = {
        'digipot1': 'blue',
        'digipot2': 'red',
        'capacitance1': 'green',
        'capacitance2': 'orange'
    }
    self.mode_labels = {
        'digipot1': 'Digipot 1',
        'digipot2': 'Digipot 2',
        'capacitance1': 'Capacitance 1',
        'capacitance2': 'Capacitance 2'
    }

    # point selected to drag
    self.selected_point = None
    self.selected_signal = None
    self.drag_threshold = 0.05

    # setup the figure and axes
    self.setup_gui()

    # initial signal generation
    self.update_signal()
```

A.3.2 UI Related Methods

```
def setup_gui(self):

    # Create matplotlib plot w/ 2 axis (digipot and io expander)
    self.ax1 = self.fig.add_subplot(111)    # digipot axis
    self.fig.subplots_adjust(bottom=0.15)

    self.ax2 = self.ax1.twinx()     # io expander axis

    # setup digipot axis
    self.ax1.set_xlim(0, self.duration)
    self.ax1.set_ylim(0, 100)
    self.ax1.set_xlabel('Time (s)')
    self.ax1.set_ylabel('Digipot Value (k)', fontweight='bold')
    self.ax1.tick_params(axis='y')
    self.ax1.grid(True, alpha=0.3)

    # setup io expander axis
    self.ax2.set_ylim(0, 255)
    self.ax2.set_ylabel('Capacitance (nF)', fontweight='bold')
    self.ax2.tick_params(axis='y')

    # Reference lines
    self.ax1.axhline(y=0, color='gray', linestyle='--', alpha=0.5)

    # Initialize signal lines
    self.signal_lines = {}  # holder for Line2D objects created with .
       plot
    self.points_plots = {}  # holder for points in the plot

    # digipot signals
    self.signal_lines['digipot1'], = self.ax1.plot(self.t, np.zeros(len(
       self.t)), 'b-', linewidth=2, label='Digipot 1')
    self.signal_lines['digipot2'], = self.ax1.plot(self.t, np.zeros(len(
       self.t)), 'r-', linewidth=2, label='Digipot 2')
```

```
        self.points_plots['digipot1'], = self.ax1.plot([], [], 'bo',
            markersize=8, zorder=5)
        self.points_plots['digipot2'], = self.ax1.plot([], [], 'ro',
            markersize=8, zorder=5)

        # io expander signals
        self.signal_lines['capacitance1'], = self.ax2.plot(self.t, np.zeros(
            len(self.t)), 'g-', linewidth=2, label='Capacitance 1')
        self.signal_lines['capacitance2'], = self.ax2.plot(self.t, np.zeros(
            len(self.t)), color='orange', linewidth=2, label='Capacitance 2')
        self.points_plots['capacitance1'], = self.ax2.plot([], [], 'go',
            markersize=8, zorder=5)
        self.points_plots['capacitance2'], = self.ax2.plot([], [], 'o',
            color='orange', markersize=8, zorder=5)

        # setup title and legend
        self.update_title()
        lines1, labels1 = self.ax1.get_legend_handles_labels()
        lines2, labels2 = self.ax2.get_legend_handles_labels()
        self.ax1.legend(lines1 + lines2, labels1 + labels2, loc='upper left'
            )

        # connect mouse events
        self.cid_press = self.fig.canvas.mpl_connect('button_press_event',
            self.on_click)
        self.cid_release = self.fig.canvas.mpl_connect('button_release_event
            ', self.on_release)
        self.cid_motion = self.fig.canvas.mpl_connect('motion_notify_event',
             self.on_drag)

        # add control buttons
        self.setup_controls()
```

```
def setup_controls(self):

    # Mode selection button
    ax_mode = plt.axes((0.10, 0.05, 0.15, 0.04))
    self.btn_mode = Button(ax_mode, f'Mode: {self.mode_labels[self.
        current_mode]}')
    self.btn_mode.on_clicked(lambda x: self.cycle_mode())

    # Extract button
    ax_extract = plt.axes((0.27, 0.05, 0.12, 0.04))
    self.btn_extract = Button(ax_extract, 'Extract Values')
    self.btn_extract.on_clicked(lambda x: self.extract_and_normalize())

    # Exit button
    ax_exit = plt.axes((0.8, 0.05, 0.08, 0.04))
    self.btn_exit = Button(ax_exit, 'Exit')
    self.btn_exit.on_clicked(self.on_exit)
```

```
def on_exit(self, event):

    self.cleanup()
    if self.owns_figure:
        plt.close(self.fig)
    else:
        plt.close('all')

def cycle_mode(self):

    modes = ['digipot1', 'digipot2', 'capacitance1', 'capacitance2']
    current_index = modes.index(self.current_mode)
```

```
        self.current_mode = modes[(current_index + 1) % len(modes)]

        # Update button text
        self.btn_mode.label.set_text(f'Mode: {self.mode_labels[self.
            current_mode]}')

        self.update_title()     # update title

        self.fig.canvas.draw()      # Refresh display

    def on_click(self, event):

        # get plot coordinates on click
        clicked_ax = None
        x, y = None, None

        if event.inaxes == self.ax1:
            clicked_ax = self.ax1
            x, y = event.xdata, event.ydata
        elif event.inaxes == self.ax2:
            clicked_ax = self.ax2
            x, y = event.xdata, event.ydata
        else:
            return

        if x is None or y is None:
            return

        # associate coordinates to the plot mode
        x_converted, y_converted = self.convert_coordinates_to_mode_axis(x,
            y, clicked_ax)

        # clip to axis limits
        target_ax = self.get_appropriate_axis(self.current_mode)
        y_min, y_max = target_ax.get_ylim()
        y_converted = max(y_min, min(y_converted, y_max))

        # check if clicking near an existing point of the current mode
        point_idx = self.find_nearest_point(x_converted, y_converted)

        if point_idx is not None:
            # drag a point
            self.selected_point = point_idx
            self.selected_signal = self.current_mode
        else:
            # add new point
            self.add_control_point(x_converted, y_converted, self.
                current_mode)

    def on_release(self, event):

        self.selected_point = None
        self.selected_signal = None

    def on_drag(self, event):

        if self.selected_point is None or self.selected_signal is None:
            return

        # get plot coordinates on click
        clicked_ax = None
        x, y = None, None

        if event.inaxes == self.ax1:
```

```
            clicked_ax = self.ax1
            x, y = event.xdata, event.ydata
        elif event.inaxes == self.ax2:
            clicked_ax = self.ax2
            x, y = event.xdata, event.ydata
        else:
            return

        if x is None or y is None:
            return

        # associate coordinates to the plot mode
        x_converted, y_converted = self.convert_coordinates_to_mode_axis(x,
            y, clicked_ax)

        # clip coordinates
        x_converted = max(0, min(x_converted, self.duration))
        target_ax = self.get_appropriate_axis(self.selected_signal)
        y_min, y_max = target_ax.get_ylim()
        y_converted = max(y_min, min(y_converted, y_max))

        # change coordinates in the points holder list
        self.control_points[self.selected_signal][self.selected_point] = (
            x_converted, y_converted)
        self.update_signal()
```

```
def get_appropriate_axis(self, signal_name):

    if signal_name.startswith('digipot'):
        return self.ax1
    else:
        return self.ax2

def convert_coordinates_to_mode_axis(self, x, y, clicked_ax):

    target_ax = self.get_appropriate_axis(self.current_mode)

    if clicked_ax == target_ax:     # mode matches the axis
        return x, y

def find_nearest_point(self, x, y):

    current_points = self.control_points[self.current_mode]
    current_ax = self.get_appropriate_axis(self.current_mode)

    if not current_points:
        return None

    best_dist = float('inf')
    best_point = None

    for i, (px, py) in enumerate(current_points):
        dist = self.calculate_distance(x, y, px, py, current_ax)
        if dist < best_dist:
            best_dist = dist
            best_point = i

    if best_dist < self.drag_threshold:
        return best_point
    return None

def calculate_distance(self, x1, y1, x2, y2, ax):
```

```
    x_range = ax.get_xlim()[1] - ax.get_xlim()[0]
    y_range = ax.get_ylim()[1] - ax.get_ylim()[0]

    dx = (x1 - x2) / x_range
    dy = (y1 - y2) / y_range
    return np.sqrt(dx ** 2 + dy ** 2)
```

```
def add_control_point(self, x, y, signal_name):

    # constrain x to be within the time bounds
    x = max(0, min(x, self.duration))
    self.control_points[signal_name].append((x, y))
    self.update_signal()

def update_title(self):

    title = (f'Signal Designer - Mode: {self.mode_labels[self.
        current_mode]}\n'
             f'Duration: {self.duration:.2f}s, Samples: {self.
                 total_samples}, Fs: {self.sampling_rate} Hz')
    self.ax1.set_title(title)

def update_signal(self):

    # generate signals for 4 parameters
    signals = {}
    for signal_name in self.control_points.keys():
        signals[signal_name] = self.interpolate_signal(self.
            control_points[signal_name])

    # update signal lines
    for signal_name, signal_data in signals.items():
        self.signal_lines[signal_name].set_ydata(signal_data)

    # update points displayed
    for signal_name, points in self.control_points.items():
        if len(points) > 0:
            times = np.array([p[0] for p in points])
            amplitudes = np.array([p[1] for p in points])
            self.points_plots[signal_name].set_data(times, amplitudes)
        else:
            self.points_plots[signal_name].set_data([], [])


    self.fig.canvas.draw()   # Refresh display

def cleanup(self):

    try:
        # disconnect events
        self.fig.canvas.mpl_disconnect(self.cid_press)
        self.fig.canvas.mpl_disconnect(self.cid_release)
        self.fig.canvas.mpl_disconnect(self.cid_motion)
    except Exception as e:
        print(f"Warning: Error during cleanup: {e}")
```

A.3.3 Data Handling Methods

```
def interpolate_signal(self, control_points):
```

```
        if len(control_points) < 1:
            return np.zeros(len(self.t))
        elif len(control_points) == 1:
            return np.full(len(self.t), control_points[0][1])

        # Sort control points by time
        sorted_points = sorted(control_points, key=lambda p: p[0])
        times = np.array([p[0] for p in sorted_points])
        amplitudes = np.array([p[1] for p in sorted_points])

        # Extend to cover full duration
        if times[0] > 0:
            times = np.insert(times, 0, 0)
            amplitudes = np.insert(amplitudes, 0, amplitudes[0])
        if times[-1] < self.duration:
            times = np.append(times, self.duration)
            amplitudes = np.append(amplitudes, amplitudes[-1])

        # Use linear interpolation
        interp_func = interp1d(times, amplitudes, kind='linear',
           bounds_error=False, fill_value='extrapolate')

        return interp_func(self.t)

    def normalize_values(self, values, source_range, target_range=(0, 255)):
        source_min, source_max = source_range
        target_min, target_max = target_range

        # Clip values to source range
        values = np.clip(values, source_min, source_max)

        if source_max > source_min:
            normalized = (values - source_min) / (source_max - source_min)
            scaled = normalized * (target_max - target_min) + target_min
        else:
            scaled = np.full_like(values, target_min)

        return np.round(scaled).astype(int)

    def extract_and_normalize(self):

        # Generate signals
        signals = {}
        for signal_name in self.control_points.keys():
            signals[signal_name] = self.interpolate_signal(self.
               control_points[signal_name])

        # Get axis ranges for normalization
        digipot_range = self.ax1.get_ylim()
        capacitance_range = self.ax2.get_ylim()

        # Normalize each signal type
        normalized_data = {}

        # Digipot signals
        for signal_name in ['digipot1', 'digipot2']:
            raw_values = signals[signal_name]
            normalized_values = self.normalize_values(raw_values,
               digipot_range)
            normalized_data[signal_name] = {
                'raw': raw_values,
                'normalized': normalized_values,
                'control_points': self.control_points[signal_name]
            }
```

```
    # Capacitance signals
    for signal_name in ['capacitance1', 'capacitance2']:
        raw_values = signals[signal_name]
        normalized_values = self.normalize_values(raw_values,
            capacitance_range)
        normalized_data[signal_name] = {
            'raw': raw_values,
            'normalized': normalized_values,
            'control_points': self.control_points[signal_name]
        }

    # Store for later access
    self.extracted_data = {
        'time': self.t,
        'duration': self.duration,
        'sampling_rate': self.sampling_rate,
        'total_samples': self.total_samples,
        'signals': normalized_data
    }

    return self.extracted_data
```

Appendix B

ESP32-WROVER-E - C/C++ Firmware Code

B.1 Packages & Main Variables

```
#include <freertos/semphr.h>
#include "freertos/ringbuf.h"
#include "freertos/queue.h"

#include "driver/spi_master.h"
#include "driver/spi_common.h"
#include "hal/spi_types.h"

#include "esp_heap_caps.h"
#include "esp_psram.h"
#include "esp_err.h"
#include "driver/gpio.h"
#include "driver/gptimer.h"

#include "esp_vfs_fat.h"
#include "sdmmc_cmd.h"
#include "driver/sdmmc_host.h"
#include "driver/sdspi_host.h"

#include <dirent.h>
#include <sys/stat.h>

#define ECG_PAYLOAD_SIZE 2          // 2 bytes
#define DIGIPOT_PAYLOAD_SIZE 1      // 1 byte
#define IO_EXPANDER_PAYLOAD_SIZE 1  // 1 byte

// ======== DIGIPOT MCP42100 ======
#define DIGIPOT1_COMMAND_BYTE 0x11
#define DIGIPOT2_COMMAND_BYTE 0x12
#define DIGIPOT_CS GPIO_NUM_22      // CS pin

// ======= MCP23S08 opcodes
#define MCP23S08_DEV1_WRITE 0x40  // ADDR A0=0, A1=0 , io topo
#define MCP23S08_DEV2_WRITE 0x42  // ADDR A0=1, A1=0,   io meio
#define IO_EXP_CS_1 GPIO_NUM_2    // CS pin

// ======= MCP23S08 register addresses ====
#define MCP23S08_IODIR 0x00    // Direction register
#define MCP23S08_IOCON 0x05    // Configuration register
```

```
#define MCP23S08_GPPU 0x06      // Pull-up resistor register
#define MCP23S08_OLAT 0x0A      // Output latch register

// ====== DAC8552 ======
#define DAC_A_WRITE_AND_LOAD 0x10  // Write to register A, update DAC A
#define DAC_B_WRITE_AND_LOAD 0x24  // Write to register B, update DAC B
#define DAC_CS GPIO_NUM_15         // CS pin
```

B.2 Organization & Declarations

```
/* =============== Command List =============== */
typedef enum {
  CMD_ECG_START = 0,
  CMD_ECG_STOP,
  CMD_ECG_DATA,
  CMD_ECG_OUT,
  CMD_CONFIG_UPDATE,
  CMD_SD_LIST,
  CMD_SD_DELETE,
  CMD_SD_RENAME,
  CMD_SD_INFO,
  DATA_TYPE_COUNT
} DataType_t;

/* ====== Structs Organization ======== */
// 1 ===== Task Management
// Task calling organization
typedef enum {
  SUBSYSTEM_PACKET_PARSER = 0,  // packet parser task
  SUBSYSTEM_ECG_STORE,          // SD writing task
  SUBSYSTEM_ECG_OUT,            // signal generation tasks
  SUBSYSTEM_COUNT
} SubsystemType_t;

// Subsystem organization
typedef struct {
  volatile bool active;         // flag: system in execution

  // task configuration
  struct {
    TaskHandle_t primaryHandle;    // handle of 1st task
    TaskHandle_t secondaryHandle;  // handle of 2nd task (optional)
    const char* primaryName;       // name of 1st task
    const char* secondaryName;     // name of 2nd task (optional)
    uint32_t stackSize;            // task RAM consumption
    UBaseType_t primaryPriority;   // priority of 1st task
    UBaseType_t secondaryPriority; // priority of 2nd task (optional)
    BaseType_t primaryCoreId;    // core affinity, 1st task
    BaseType_t secondaryCoreId;  // core affinity, 2nd task (optional)
  } tasks;

  // resources
  struct {
    RingbufHandle_t ringBuffer;     // subsystem ringbuffer handle
    QueueHandle_t queue;            // subsystem queue handle
    size_t bufferSize;              // buffer size
    size_t queueLength;             // queue size
  } resources;

  // task lifecycle functions
  esp_err_t (*init)(void);          // start resources
```

```
  esp_err_t (*start)(void);          // start task
  esp_err_t (*stop)(void);           // delete task
  void (*clean)(void);               // clean resources

} SubsystemConfig_t;

// 2 ===== Data Management
// Stream header structure
typedef struct {
  uint16_t startMarker;  // 0xBB66
  uint16_t numPackets;   // Number of data packets following
  uint8_t command;       // Command type
  uint16_t endMarker;    // 0x66BB
} __attribute__((packed)) StreamHeader_t;

// Single pack struct
typedef struct {
  uint16_t ecgData;                     // 16-bit dac
  uint8_t digipot1Data;                 // wiper 1
  uint8_t digipot2Data;                 // wiper 2
  uint8_t ioExpander1Data;              // io 1
  uint8_t ioExpander2Data;              // io 2
} __attribute__((packed)) DataPacket_t;

// Stream data block for direct SD card writing
typedef struct {
  uint8_t* data;            // stream data pack received
  size_t size;              // stream data size
  uint16_t numPackets;      // number of single data packs
} StreamDataBlock_t;

// Parsing case sequence
typedef enum {
  PARSE_STREAM_START = 0,        // first case
  PARSE_STREAM_NUM_PACKETS,      // second case
  PARSE_STREAM_CMD,              // third case
  PARSE_STREAM_END,              // final header case
  PARSE_RAW_DATA_BLOCK,          // conditional data case
  PARSE_STATE_COUNT
} ParseState_t;

// Configuration pack structure
typedef struct {
  char fileName[64];           // File path
  uint32_t samplingFrequency;  // Sampling frequency (Hz)
  int getTimerAlarmUs() {
    return (1 * 1000 * 1000) / samplingFrequency;
  };                           // Sampling interval
  int productionCountMS(int chunkSize) {
    return (pdMS_TO_TICKS((chunkSize * getTimerAlarmUs() + 1000) / 999)
        - 2);
  }                            // Production interval
} __attribute__((packed)) ConfigPacket_t;

// SD pack structure
typedef struct {
  uint8_t command;        // SD command
  char filename[32];      // current filename
  char new_filename[32];  // new filename
} __attribute__((packed)) SDCommandPacket_t;

// 3 ============= Command Management
typedef struct {
  DataType_t type;                  // command number
```

```
  size_t dataSize;                    // packet size
  bool requiresData;                  // flag
} CommandData_t;
```

B.3 Configuration

```
/* ============== Configurations and definitions ==============*/
// ============ Global configuration
ConfigPacket_t currentConfig = {
  .fileName = "/sdcard/teste.bin",      // default filename
  .samplingFrequency = 360,             // default sampling frequency
};

SemaphoreHandle_t sdCardSemaphore = NULL;       // SD semaphore handle
static const char* SD_MOUNT_POINT = "/sdcard";  // SD card mount point
sdmmc_card_t* card = NULL;                      // SD card info
#define SD_CARD_CS GPIO_NUM_5          // CS pin of SD SPI interface

spi_device_handle_t digipotHandle;            // spi digipot handle
spi_device_handle_t ioExpanderHandle;         // spi io expander handle
spi_device_handle_t dacHandle;                // spi dac handle

SemaphoreHandle_t timerSemaphore = NULL;      // timer semaphore handle
gptimer_handle_t gptimer = NULL;              // timer handle
portMUX_TYPE timerMux = portMUX_INITIALIZER_UNLOCKED;   // ISR mutex
volatile DataPacket_t packedData;       // create a DataPacket_t object

// Task configurations
SubsystemConfig_t subsystems[SUBSYSTEM_COUNT] = {
  [SUBSYSTEM_PACKET_PARSER] = {
    .active = false,
    .tasks = {
      .primaryHandle = NULL,
      .primaryName = "packetParser",
      .stackSize = 32768,
      .primaryPriority = 1,
      .primaryCoreId = 0
    },
    .resources = {
      .ringBuffer = NULL,
      .queue = NULL
    },
    .start = packetParserStart      // parser task always active
  },
  [SUBSYSTEM_ECG_STORE] = {
    .active = false,
    .tasks = {
      .primaryHandle = NULL,
      .primaryName = "SDWriteTask",
      .stackSize = 16384,
      .primaryPriority = 2,
      .primaryCoreId = 1,
    },
    .resources = {
      .ringBuffer = NULL,
      .queue = NULL
    },
    .init = ecgStoreInit,
    .start = ecgStoreStart,
    .stop = ecgStoreStop,
    .clean = ecgStoreClean
```

```
  },
  [SUBSYSTEM_ECG_OUT] = {
    .active = false,
    .tasks = {
      .primaryHandle = NULL,
      .secondaryHandle = NULL,
      .primaryName = "outputBufferAddTask",
      .secondaryName = "outputBufferGetTask",
      .stackSize = 8192, .primaryPriority = 2,
      .secondaryPriority = 3,
      .primaryCoreId = 0,
      .secondaryCoreId = 1
    },
    .resources = {
      .ringBuffer = NULL,
      .queue = NULL
    },
    .init = ecgOutInit,
    .start = ecgOutStart,
    .stop = ecgOutStop,
    .clean = ecgOutClean
  }
};

// Command configurations
static const CommandData_t CommandData[] = {
  [CMD_ECG_START] = {
    .type = CMD_ECG_START,
    .dataSize = 0,
    .requiresData = false,
  },
  [CMD_ECG_STOP] = {
    .type = CMD_ECG_STOP,
    .dataSize = 0,
    .requiresData = false,
  },
  [CMD_ECG_DATA] = {
    .type = CMD_ECG_DATA,
    .dataSize = 0,              // Depends on numPackets
    .requiresData = true,
  },
  [CMD_ECG_OUT] = {
    .type = CMD_ECG_OUT,
    .dataSize = 0,
    .requiresData = false,
  },
  [CMD_CONFIG_UPDATE] = {
    .type = CMD_CONFIG_UPDATE,
    .dataSize = sizeof(ConfigPacket_t),     // Configuration pack
    .requiresData = true,
  },
  [CMD_SD_LIST] = {
    .type = CMD_SD_LIST,
    .dataSize = 0,
    .requiresData = false,
  },
  [CMD_SD_DELETE] = {
    .type = CMD_SD_DELETE,
    .dataSize = sizeof(SDCommandPacket_t), // SD specific pack
    .requiresData = true,
  },
  [CMD_SD_RENAME] = {
    .type = CMD_SD_RENAME,
    .dataSize = sizeof(SDCommandPacket_t), // SD specific pack
```

```
    .requiresData = true,
  },
  [CMD_SD_INFO] = {
    .type = CMD_SD_INFO,
    .dataSize = sizeof(SDCommandPacket_t), // SD specific pack
    .requiresData = true,
  }
};

/* ================== Prototype Functions ================== */
// 1 ============== Lifecycle
esp_err_t ecgStoreInit(void);
esp_err_t ecgStoreStart(void);
esp_err_t ecgStoreStop(void);
void ecgStoreClean(void);
esp_err_t ecgOutInit(void);
esp_err_t ecgOutStart(void);
esp_err_t ecgOutStop(void);
void ecgOutClean(void);

// 2 ============== Tasks
void SDWriteTask(void* parameter);
void packetParserTask(void* parameter);
void outputBufferAddTask(void* parameter);
void outputBufferGetTask(void* parameter);

// 3 ============== SPI & peripherals
esp_err_t spiStart();
esp_err_t sdCardInit();
bool testSDCard();
esp_err_t digipotStart();
esp_err_t IoSpiStart();
esp_err_t dacStart();
esp_err_t timerStart();
void setMCP23S08(uint8_t deviceOpCode);

void setWiper(uint8_t commandByte, uint8_t valueByte);
void setIOExpander(uint8_t deviceAddress, uint8_t registerAddress,
    uint8_t dataByte);
void setDAC(uint8_t command, uint16_t dataBytes);

void listSDFiles(void);
void deleteSDFile(const char* filename);
void renameSDFile(const char* oldName, const char* newName);
void getFileInfo(const char* filename);

// 4 ============== Parser methods
esp_err_t packetParserStart(void);
void processStreamData(uint8_t* streamData, size_t dataSize, uint16_t
    numPackets, uint8_t command);
void processStreamHeader(StreamHeader_t* header);
void executeCommand(uint8_t command, uint8_t* data, size_t dataSize,
    uint16_t numPackets);
void applyConfiguration(ConfigPacket_t* newConfig);
```

B.4 *SDWriteTask* task

```
void SDWriteTask(void* parameter) {
  SubsystemConfig_t* ecgStoreConfig = &subsystems[SUBSYSTEM_ECG_STORE];

  bool fileIsOpen = false;              // Flag: file status
```

```
FILE* dataFile = NULL;                  // Object type (POSIX)
uint32_t totalSamplesWritten = 0;
uint32_t totalBytesWritten = 0;

// Open file for writing
if (xSemaphoreTake(sdCardSemaphore, pdMS_TO_TICKS(2000))) {
  dataFile = fopen(currentConfig.fileName, "wb+");
  if (dataFile) {
    fileIsOpen = true;
    setvbuf(dataFile, NULL, _IOFBF, 16384);  // 16KB buffer
  } else {
    perror("Error opening file");
  }
  xSemaphoreGive(sdCardSemaphore);
}

for (;;) {
  if (ecgStoreConfig->active) {
    StreamDataBlock_t dataBlock;
    if (xQueueReceive(ecgStoreConfig->resources.queue, &dataBlock,
       pdMS_TO_TICKS(500)) == pdTRUE) {

      if (dataBlock.data != NULL && dataBlock.size > 0) {

        // Write stream block to SD card in one operation
        if (fileIsOpen && dataFile && xSemaphoreTake(sdCardSemaphore,
           pdMS_TO_TICKS(2000))) {

          size_t bytesWritten = fwrite(dataBlock.data, 1, dataBlock.
             size, dataFile);

          if (bytesWritten != dataBlock.size) {
            perror("Stream write error");
          } else {
            totalSamplesWritten += dataBlock.numPackets;
            totalBytesWritten += bytesWritten;

            // Flush after each stream (size=chunk * DataPacket_t size
                (6 bytes))
            int flushResult = fflush(dataFile);
            if (flushResult != 0) {
              perror("stream flush error");
            }
          }
          xSemaphoreGive(sdCardSemaphore);
        }

        heap_caps_free(dataBlock.data);   // Free the stream data
           memory
        dataBlock.data = NULL;
      }
    }
    vTaskDelay(pdMS_TO_TICKS(2));
  } else {
    goto cleanup;     // task self-deletion uppon inactivation
  }
}

cleanup:
  if (fileIsOpen && xSemaphoreTake(sdCardSemaphore, pdMS_TO_TICKS
     (2000))) {
    if (dataFile) {

      int syncResult = fsync(fileno(dataFile));   // Final sync
```

```
        if (syncResult != 0) {
          perror("Final sync error");
        }

        // Get final file size before closing
        if (fseek(dataFile, 0, SEEK_END) == 0) {
          long fileSize = ftell(dataFile);
        }

        int closeResult = fclose(dataFile);     // Close file
        if (closeResult != 0) {
          perror("Close error");
        }

        fileIsOpen = false;     // reset flag
        dataFile = NULL         // reset flag
      }
      xSemaphoreGive(sdCardSemaphore);      // return semaphore
    }

    // deelte handle and task
    ecgStoreConfig->tasks.primaryHandle = NULL;
    vTaskDelete(NULL);
}
```

B.5 *packetParserTask*

```
void packetParserTask(void* parameter) {
  SubsystemConfig_t* packetConfig = &subsystems[SUBSYSTEM_PACKET_PARSER
     ];
  ParseState_t parseState = PARSE_STREAM_START;

  uint16_t bytesRead = 0;     // Counter: parse the header data
  uint16_t packetsToProcess = 0;     // nr packets of single packs
  uint8_t currentCommand = 0;      // command in the header

  uint8_t* streamData = NULL;    // address for a buffer
  size_t streamDataSize = 0;     // data size information in stream
  size_t streamBytesRead = 0;    // data read in stream
  StreamHeader_t* header = (StreamHeader_t*)heap_caps_malloc(sizeof(
     StreamHeader_t), MALLOC_CAP_8BIT);     // memory allocation

  if (header == NULL) {
    packetConfig->active = false;
    vTaskDelete(NULL);
    return;
  }

  // reset parse struct
  memset(header, 0, sizeof(StreamHeader_t));

  for (;;) {
    if (Serial.available() > 0) {
      switch (parseState) {
        case PARSE_STREAM_START: {    // search for start bytes

            ((uint8_t*)&header->startMarker)[bytesRead++] = Serial.read
               ();
            if (bytesRead >= 2) {
              if (header->startMarker == 0xBB66) {
                parseState = PARSE_STREAM_NUM_PACKETS;
```

```
        } else {
          parseState = PARSE_STREAM_START;   // reset parser
        }
        bytesRead = 0;    // reset counter
      }
      break;
    }
    case PARSE_STREAM_NUM_PACKETS: {
      ((uint8_t*)&header->numPackets)[bytesRead++] = Serial.read()
          ;
      if (bytesRead >= 2) {
        packetsToProcess = header->numPackets;
        parseState = PARSE_STREAM_CMD;
        bytesRead = 0;    // reset counter
      }
      break;
    }
    case PARSE_STREAM_CMD: {
      header->command = Serial.read();
      currentCommand = header->command;

      if (header->command >= DATA_TYPE_COUNT) {
        parseState = PARSE_STREAM_START;   // reset parser
        bytesRead = 0;        // reset counter
        memset(header, 0, sizeof(StreamHeader_t));  // reset parse
            struct
        break;
      }

      parseState = PARSE_STREAM_END;
      bytesRead = 0;      // reset counter
      break;
    }
    case PARSE_STREAM_END: {
      ((uint8_t*)&header->endMarker)[bytesRead++] = Serial.read();
      if (bytesRead >= 2) {
        if (header->endMarker == 0x66BB) {
          // Validate command
          if (currentCommand >= DATA_TYPE_COUNT) {
            parseState = PARSE_STREAM_START;      // reset parser
            memset(header, 0, sizeof(StreamHeader_t));   // reset
                parse struct
            bytesRead = 0;        // reset counter
            break;
          }

          // create command struct
          const CommandData_t* cmdMeta = &CommandData[
              currentCommand];

          // process header
          processStreamHeader(header);

          // check if command require data
          if (cmdMeta->requiresData && packetsToProcess > 0) {
            // Calculate data size
            if (currentCommand == CMD_ECG_DATA) {
              streamDataSize = packetsToProcess * sizeof(
                  DataPacket_t);
            } else {
              streamDataSize = cmdMeta->dataSize;
            }

            // allocate memory for stream
```

```
            streamData = (uint8_t*)heap_caps_malloc(streamDataSize
                , MALLOC_CAP_8BIT);
            if (streamData == NULL) {
              parseState = PARSE_STREAM_START;      // reset
                 parser
              memset(header, 0, sizeof(StreamHeader_t));   //
                 reset parse struct
              break;
            }

            streamBytesRead = 0;
            parseState = PARSE_RAW_DATA_BLOCK;
          } else {
            // No data needed, reset for next stream
            parseState = PARSE_STREAM_START;  // reset parser
            memset(header, 0, sizeof(StreamHeader_t));  // reset
               parse struct
          }
        } else {
          parseState = PARSE_STREAM_START;   // reset parser
          memset(header, 0, sizeof(StreamHeader_t));
        }
        bytesRead = 0;        // reset counter
      }
      break;
  }
  case PARSE_RAW_DATA_BLOCK: {
      // read data bytes directly into stream buffer
      while (Serial.available() > 0 && streamBytesRead <
          streamDataSize) {
          streamData[streamBytesRead++] = Serial.read();
      }

      // check if the data in stream is read in full
      if (streamBytesRead >= streamDataSize) {

          // process the entire stream at once
          processStreamData(streamData, streamDataSize,
              packetsToProcess, currentCommand);

          streamData = NULL;
          streamDataSize = 0;     // reset counter
          streamBytesRead = 0;    // reset counter
          parseState = PARSE_STREAM_START;    // reset parser
          memset(header, 0, sizeof(StreamHeader_t)); // reset
              parse struct
          packetsToProcess = 0;
      }
      break;
  }
  default: {
      parseState = PARSE_STREAM_START;        // reset parser
      bytesRead = 0;      // reset counter

      // Clean up stream data if allocated
      if (streamData != NULL) {
          heap_caps_free(streamData);     // free allocated memory
          streamData = NULL;
      }
      streamDataSize = 0;     // reset counter
      streamBytesRead = 0;    // reset counter
      memset(header, 0, sizeof(StreamHeader_t));
      break;
      }
```

```
        }
    }
    vTaskDelay(pdMS_TO_TICKS(1));
  }
}
```

B.6 *outputBufferAddTask* task

```
void outputBufferAddTask(void* parameter) {
  SubsystemConfig_t* ecgOutConfig = &subsystems[SUBSYSTEM_ECG_OUT];
  size_t filePosition = 0;                // file positioning
  FILE* readFile = NULL;                  // Object type (POSIX)
  bool fileIsOpen = false;                // Flag: file status
  uint32_t packetsRead = 0;               // Counter: single packs
 #define CHUNK_SIZE 100                   // default value of packs in a
    chunk
  DataPacket_t packetChunk[CHUNK_SIZE]; // CHUNK_SIZE number of single
     packs
  bool isPreFilled = false;               // Flag: buffer prefill stage

  // Check file existance and data availability
  struct stat st;
  if (stat(currentConfig.fileName, &st) != 0) {
    perror("Stat error");                 // check file existance
    vTaskDelete(NULL);                    // Task deletion
    return;
  } else {
    if (st.st_size == 0) {                // check file size (bytes)
      vTaskDelete(NULL);                  // Task deletion
      return;
    }
  }

  // calculate number of single packs
  uint32_t totalPackets = st.st_size / sizeof(DataPacket_t);

  // Open file for reading (binary mode)
  if (xSemaphoreTake(sdCardSemaphore, portMAX_DELAY)) {
    readFile = fopen(currentConfig.fileName, "rb");
    if (readFile) {
      fileIsOpen = true;                          // Flag: file open
      fseek(readFile, filePosition, SEEK_SET); // set file position at 0
    } else {
      perror("Open error");
    }
    xSemaphoreGive(sdCardSemaphore);      // return semaphore
  }

  //  Pre-fill stage
  uint32_t preFillPackets = 0;
  int targetPreFill = 300;         // 1800 bytes

  // Pre-fill loop condition
  while (!isPreFilled && fileIsOpen && packetsRead < totalPackets &&
     ecgOutConfig->active) {
    size_t availableSpace = xRingbufferGetCurFreeSize(ecgOutConfig->
       resources.ringBuffer);
    uint32_t maxPacketsToFit = availableSpace / sizeof(DataPacket_t);

    if (maxPacketsToFit >= CHUNK_SIZE) {
      if (xSemaphoreTake(sdCardSemaphore, pdMS_TO_TICKS(1000))) {
```

```
        uint32_t packetsToRead = (totalPackets - packetsRead >
            CHUNK_SIZE) ? CHUNK_SIZE : (totalPackets - packetsRead);

        size_t itemsRead = fread(packetChunk, sizeof(DataPacket_t),
            packetsToRead, readFile);          // read data chunk
        filePosition = ftell(readFile);          // update file position

        xSemaphoreGive(sdCardSemaphore);         // return semaphore

        if (itemsRead > 0) {
          // Send packets individually to ring buffer
          for (size_t i = 0; i < itemsRead; i++) {
            BaseType_t result = xRingbufferSend(
                ecgOutConfig->resources.ringBuffer, &packetChunk[i],
                    sizeof(DataPacket_t), pdMS_TO_TICKS(1000));
            if (result != pdTRUE) {
              break;
            }
            preFillPackets++;
          }
          packetsRead += itemsRead;

          if (preFillPackets >= targetPreFill) { // check pre-fill
            isPreFilled = true;       // Flag: pre-fill complete
            break;
          }
        } else {
          if (feof(readFile)) {
            isPreFilled = true;  // pre-fill complete with end-of-file
            break;
          } else {
            perror("Pre-fill read error");
            break;
          }
        }
      }
    }
    vTaskDelay(pdMS_TO_TICKS(2));
  }

  vTaskDelay(pdMS_TO_TICKS(50));    // stabilize SD operations

  // Start the timer
  if (gptimer_start(gptimer) != ESP_OK) {
    gptimer_disable(gptimer);
    gptimer_del_timer(gptimer);
    return;
  }

  // Main reading loop
  for (;;) {
    if (ecgOutConfig->active) {
      if (fileIsOpen && packetsRead < totalPackets) {

        // Check ring buffer space
        if (xRingbufferGetCurFreeSize(ecgOutConfig->resources.ringBuffer
            ) >= sizeof(DataPacket_t) * CHUNK_SIZE) {
          if (xSemaphoreTake(sdCardSemaphore, portMAX_DELAY)) {
            uint32_t packetsToRead = (totalPackets - packetsRead >
                CHUNK_SIZE) ? CHUNK_SIZE : (totalPackets - packetsRead);

            size_t itemsRead = fread(packetChunk, sizeof(DataPacket_t),
                packetsToRead, readFile);       // read data chunk
            filePosition = ftell(readFile);     // update file position
```

```
            xSemaphoreGive(sdCardSemaphore);    // return semaphore

            // try to send data to ring buffer. 5 attempts max
            if (itemsRead > 0) {
              const uint8_t MAX_RETRIES = 5;

              for (size_t i = 0; i < itemsRead; i++) {
                bool packetSent = false;

                for (uint8_t retry = 0; retry < MAX_RETRIES && !
                    packetSent; retry++) {
                  BaseType_t result = xRingbufferSend(ecgOutConfig->
                      resources.ringBuffer, &packetChunk[i], sizeof(
                      DataPacket_t), pdMS_TO_TICKS(2000));

                  if (result == pdTRUE) {      // data sent successfully
                    packetSent = true;
                  } else {                     // counter: retry logic
                    if (retry < MAX_RETRIES - 1) {
                      vTaskDelay(pdMS_TO_TICKS(10));
                    }
                  }
                }
              }
            } else {
              if (feof(readFile)) {     // check for end of file
                break;
              } else {
                perror("Read error");
                break;
              }
            }
          }
        }
      } else if (packetsRead >= totalPackets) {
        break;
      }
    } else {
      goto cleanup;
    }
    // production rate
    if (elapsedTime >= currentConfig.productionCountMS(CHUNK_SIZE)) {
        vTaskDelay(pdMS_TO_TICKS(currentConfig.productionCountMS(
            CHUNK_SIZE)));
    }
    else {
        vTaskDelay(currentConfig.productionCountMS(CHUNK_SIZE) -
            elapsedTime);
    }
  }

  // Cleanup
  cleanup:
    if (fileIsOpen && xSemaphoreTake(sdCardSemaphore, portMAX_DELAY)) {
      fclose(readFile);         // close SD file
      fileIsOpen = false;       // reset flag
      xSemaphoreGive(sdCardSemaphore);      // return semaphore
    }

    ecgOutConfig->tasks.primaryHandle = NULL;   // Delete task handle
    vTaskDelete(NULL);                  // Delete task
}
```

B.7 *outputBufferGetTask* task

```
// ================ ISR ==================
bool IRAM_ATTR onTimer(gptimer_handle_t timer, const
    gptimer_alarm_event_data_t* edata, void* user_ctx) {
  SubsystemConfig_t* ecgOutConfig = &subsystems[SUBSYSTEM_ECG_OUT];
  BaseType_t higherPriorityTaskWoken = pdFALSE;
  size_t itemSize;

  DataPacket_t* packetData = (DataPacket_t*)xRingbufferReceiveFromISR(
    ecgOutConfig->resources.ringBuffer, &itemSize);

  if (packetData != NULL && itemSize == sizeof(DataPacket_t)) {
    portENTER_CRITICAL_ISR(&timerMux);      // spinlock enter
    // packetData is copied to packedData. packedData is send between
       ISR and output task
    // w/ a void* acting as a container
    memcpy((void*)&packedData, packetData, sizeof(DataPacket_t));
    portEXIT_CRITICAL_ISR(&timerMux);      // spinlock exit

    // return the item to the ring buffer
    vRingbufferReturnItemFromISR(ecgOutConfig->resources.ringBuffer,
                                 (void*)packetData, &
                                    higherPriorityTaskWoken);

    // return timer semaphore
    BaseType_t semResult = xSemaphoreGiveFromISR(timerSemaphore, &
       higherPriorityTaskWoken);
  }

  // if a higher priority task above the current starts executing,
  // the ISR should yield its execution to that same task
  if (higherPriorityTaskWoken) {
    portYIELD_FROM_ISR();
  }

  return true;
}

// ========== Consumer Task ==============
void outputBufferGetTask(void* parameter) {
  SubsystemConfig_t* ecgOutConfig = &subsystems[SUBSYSTEM_ECG_OUT];
  DataPacket_t packetData;      // create DataPacket_t object

  for (;;) {
    if (ecgOutConfig->active) {

      // take timer semaphore (ISR controlled)
      if (xSemaphoreTake(timerSemaphore, portMAX_DELAY)) {

        portENTER_CRITICAL(&timerMux);      // spinlock enter
        // ISR critical section (copy data)
        memcpy(&packetData, (const void*)&packedData, sizeof(
           DataPacket_t));
        portEXIT_CRITICAL(&timerMux);       // spinlock exit

        // Output to hardware
        spi_device_acquire_bus(dacHandle, portMAX_DELAY);
        setDAC(DAC_A_WRITE_AND_LOAD, packetData.ecgData);
        setDAC(DAC_B_WRITE_AND_LOAD, packetData.ecgData);
        spi_device_release_bus(dacHandle);

        spi_device_acquire_bus(digipotHandle, portMAX_DELAY);
```

```
        setWiper(DIGIPOT1_COMMAND_BYTE, packetData.digipot1Data);
        setWiper(DIGIPOT2_COMMAND_BYTE, packetData.digipot2Data);
        spi_device_release_bus(digipotHandle);

        spi_device_acquire_bus(ioExpanderHandle, portMAX_DELAY);
        setIOExpander(MCP23S08_DEV1_WRITE, MCP23S08_OLAT, packetData.
            ioExpander1Data);
        setIOExpander(MCP23S08_DEV2_WRITE, MCP23S08_OLAT, packetData.
            ioExpander2Data);
        spi_device_release_bus(ioExpanderHandle);

      } else {
        if (ecgOutConfig->tasks.primaryHandle == NULL &&
            xRingbufferGetCurFreeSize(ecgOutConfig->resources.ringBuffer)
             == 16384) {
          goto cleanup;     // producer task was deleted and ringbuffer
             is empty
        }
      }
    } else {
      break;
    }
  }

  cleanup:
    setDAC(DAC_A_WRITE_AND_LOAD, 0x00);     // reset DAC
    setDAC(DAC_B_WRITE_AND_LOAD, 0x00);     // reset DAC
    gptimer_stop(gptimer);          // stop timer ISR events

    // Small process delay. handle and task deletion.
    vTaskDelay(pdMS_TO_TICKS(2));
    ecgOutConfig->tasks.secondaryHandle = NULL;
    vTaskDelete(NULL);
}
```

B.8 SPI interface & Peripherals

```
/* ============== SPI Interface Establishment ==============*/
esp_err_t spiStart() {
  esp_err_t ret;
  // VSPI Configuration
  spi_bus_config_t vspi_bus_config = {
    .mosi_io_num = GPIO_NUM_23,
    .miso_io_num = GPIO_NUM_19,
    .sclk_io_num = GPIO_NUM_18,
    .quadwp_io_num = -1,
    .quadhd_io_num = -1,
    .max_transfer_sz = 4096,
  };

  spi_bus_config_t hspi_bus_config = {
    .mosi_io_num = GPIO_NUM_13,
    .miso_io_num = -1,
    .sclk_io_num = GPIO_NUM_14,
    .quadwp_io_num = -1,
    .quadhd_io_num = -1,
    .max_transfer_sz = 4096,
  };

  // initialize VSPI and HSPI.
  // DMA channels selected by the mcu automatically
```

```
  ret = spi_bus_initialize(VSPI_HOST, &vspi_bus_config, SPI_DMA_CH_AUTO)
      ;
  if (ret != ESP_OK) {
    return ret;
  }

  ret = spi_bus_initialize(HSPI_HOST, &hspi_bus_config, SPI_DMA_CH_AUTO)
      ;
  if (ret != ESP_OK) {
    return ret;
  }

  return ret;
}

// ========== Digital Potentiometers
esp_err_t digipotStart() {
  esp_err_t ret;
  // digipot Configuration
  spi_device_interface_config_t vspi_digipot_config = {
    .command_bits = 0,
    .address_bits = 0,
    .mode = 0,
    .duty_cycle_pos = 128,
    .clock_speed_hz = 1000000,  // 1MHz
    .spics_io_num = DIGIPOT_CS,
    .queue_size = 5,
  };

  // add digipot to VSPI
  ret = spi_bus_add_device(VSPI_HOST, &vspi_digipot_config, &
      digipotHandle);

  return ret;
}

void setWiper(uint8_t commandByte, uint8_t valueByte) {

  // Prepare transaction
  uint8_t tx_data[2];  // 2 elements

  tx_data[0] = commandByte;
  tx_data[1] = valueByte;

  spi_transaction_t transaction;
  memset(&transaction, 0, sizeof(transaction));

  // Set up transaction
  transaction.length = 16;           // size of transmission
  transaction.tx_buffer = tx_data;   // data to transmit

  // Execute transaction
  esp_err_t result = spi_device_polling_transmit(digipotHandle, &
      transaction);
}

// ========== Digital to Analog Converter
esp_err_t dacStart() {
  esp_err_t ret;
  // dac Configuration
  spi_device_interface_config_t hspi_dac_config = {
    .command_bits = 0,
    .address_bits = 0,
    .mode = 1,
```

```
    .duty_cycle_pos = 128,
    .clock_speed_hz = 1000000,  // 1MHz
    .spics_io_num = DAC_CS,
    .queue_size = 5,
  };

  // add dac to HSPI
  ret = spi_bus_add_device(HSPI_HOST, &hspi_dac_config, &dacHandle);

  return ret;
}

void setDAC(uint8_t command, uint16_t dataBytes) {

  uint8_t tx_data[3];  // 2 elements

  tx_data[0] = command;
  tx_data[1] = (dataBytes >> 8) & 0xFF;
  tx_data[2] = (dataBytes & 0xFF);

  spi_transaction_t transaction;
  memset(&transaction, 0, sizeof(transaction));

  // Set up transaction
  transaction.length = 24;
  transaction.tx_buffer = tx_data;
  transaction.flags = 0;

  esp_err_t ret = spi_device_polling_transmit(dacHandle, &transaction);
}

// ========== I/O Expanders
esp_err_t IoSpiStart() {
  esp_err_t ret;

  // io exp Configuration
  spi_device_interface_config_t vspi_io_config = {
    .command_bits = 0,
    .address_bits = 0,
    .mode = 0,
    .duty_cycle_pos = 128,
    .clock_speed_hz = 1000000,  // 1MHz
    .spics_io_num = IO_EXP_CS_1,
    .queue_size = 1,
  };

  // add io expander to VSPI
  ret = spi_bus_add_device(VSPI_HOST, &vspi_io_config, &ioExpanderHandle
      );

  return ret;
}

void setMCP23S08(uint8_t deviceOpCode) {

  setIOExpander(deviceOpCode, MCP23S08_IOCON, 0x08);  // Enable hardware
       addressing
  setIOExpander(deviceOpCode, MCP23S08_IODIR, 0x00);  // Configure all
     pins as outputs
  setIOExpander(deviceOpCode, MCP23S08_GPPU, 0x00);   // Disable pull-
     ups
  setIOExpander(deviceOpCode, MCP23S08_OLAT, 0x00);   // Set all pins
     low initially
}
```

```
void setIOExpander(uint8_t deviceAddress, uint8_t registerAddress,
    uint8_t dataByte) {

  uint8_t tx_data[3];  // 3 elements

  tx_data[0] = deviceAddress;
  tx_data[1] = registerAddress;
  tx_data[2] = dataByte;

  spi_transaction_t transaction;
  memset(&transaction, 0, sizeof(transaction));

  // Set up transaction
  transaction.length = 24;
  transaction.tx_buffer = tx_data;
  transaction.flags = 0;

  esp_err_t ret = spi_device_polling_transmit(ioExpanderHandle, &
      transaction);
}

// ========== SD Card
esp_err_t sdCardInit() {
  esp_err_t ret;

  // mount the SD card
  esp_vfs_fat_sdmmc_mount_config_t mount_config = {
    .format_if_mount_failed = false,
    .max_files = 5,
    .allocation_unit_size = 16 * 1024
  };

  // SD card configuration
  sdspi_device_config_t slot_config = SDSPI_DEVICE_CONFIG_DEFAULT();
  slot_config.host_id = VSPI_HOST;
  slot_config.gpio_cs = SD_CARD_CS;

  // initialize SD SPI host
  sdmmc_host_t host = SDSPI_HOST_DEFAULT();
  host.slot = VSPI_HOST;
  host.max_freq_khz = 4000;

  ret = esp_vfs_fat_sdspi_mount(SD_MOUNT_POINT, &host, &slot_config, &
      mount_config, &card);

  if (ret != ESP_OK) {
    return ret;
  }

  sdmmc_card_print_info(stdout, card);

  return ret;
}

// ========== Timer
esp_err_t timerStart() {

  esp_err_t ret;

  // Timer configuration
  gptimer_config_t timer_config = {
    .clk_src = GPTIMER_CLK_SRC_DEFAULT,
    .direction = GPTIMER_COUNT_UP,
```

```
    .resolution_hz = 1000000,  // 1MHz, 1 tick = 1us
  };

  // Create timer
  ret = gptimer_new_timer(&timer_config, &gptimer);
  if (ret != ESP_OK) {
    return ESP_FAIL;
  }

  // Event callbacks configuration
  gptimer_event_callbacks_t cbs = {
    .on_alarm = onTimer
  };

  ret = gptimer_register_event_callbacks(gptimer, &cbs, NULL);
  if (ret != ESP_OK) {
    gptimer_del_timer(gptimer);
    return ESP_FAIL;
  }

  // Set alarm configuration
  gptimer_alarm_config_t alarm_config = {};
  alarm_config.alarm_count = currentConfig.getTimerAlarmUs();
  alarm_config.reload_count = 0;
  alarm_config.flags.auto_reload_on_alarm = true;

  ret = gptimer_set_alarm_action(gptimer, &alarm_config);
  if (ret != ESP_OK) {
    gptimer_disable(gptimer);
    gptimer_del_timer(gptimer);
    return ESP_FAIL;
  }

  // Enable timer
  ret = gptimer_enable(gptimer);
  if (ret != ESP_OK) {
    gptimer_del_timer(gptimer);
    return ESP_FAIL;
  }

  return ESP_OK;
}
```

B.9 SD Card methods

```
/* ============== SD Card methods ==============*/
// ========== Test SD card
bool testSDCard() {
  FILE* testFile = fopen("/sdcard/test.txt", "w");
  if (testFile == NULL) {
    Serial.println("Failed to open text test file for writing!");
    return false;
  }

  fprintf(testFile, "SD Card Text Test\n");
  fclose(testFile);
  Serial.println("Text test file written to SD card");

  // Now test with a binary file
  uint32_t testData[5] = { 0xDEADBEEF, 0x12345678, 0xABCDEF01, 0
     x87654321, 0xFEEDBEEF };
```

```
  testFile = fopen("/sdcard/test.bin", "wb");  // Note: 'wb' mode for
     binary writing
  if (testFile == NULL) {
    Serial.println("Failed to open binary test file for writing!");
    return false;
  }

  size_t bytesWritten = fwrite(testData, sizeof(uint32_t), 5, testFile);
  if (bytesWritten != 5) {
    Serial.printf("Binary write test failed: wrote %d of 5 elements\n",
       bytesWritten);
    fclose(testFile);
    return false;
  }
  fclose(testFile);
  Serial.println("Binary test data written successfully");

  // Try to read the binary file back
  uint32_t readData[5] = { 0 };
  testFile = fopen("/sdcard/test.bin", "rb");  // Note: 'rb' mode for
     binary reading
  if (testFile == NULL) {
    Serial.println("Failed to open binary test file for reading!");
    return false;
  }

  size_t bytesRead = fread(readData, sizeof(uint32_t), 5, testFile);
  fclose(testFile);

  if (bytesRead != 5) {
    Serial.printf("Binary read test failed: read %d of 5 elements\n",
       bytesRead);
    return false;
  }

  // Verify the data
  bool dataValid = true;
  for (int i = 0; i < 5; i++) {
    if (readData[i] != testData[i]) {
      Serial.printf("Data mismatch at index %d: wrote 0x%08X, read 0x%08
         X\n",
                    i, testData[i], readData[i]);
      dataValid = false;
    }
  }

  if (dataValid) {
    Serial.println("Binary read/write test passed!");
    return true;
  } else {
    Serial.println("Binary read/write test failed - data mismatch");
    return false;
  }
}

// ========== List SD card files
void listSDFiles() {
  Serial.println("SD Card Files:");
  vTaskDelay(pdMS_TO_TICKS(1000));
  if (xSemaphoreTake(sdCardSemaphore, pdMS_TO_TICKS(2000))) {
    DIR* dir = opendir(SD_MOUNT_POINT);
    if (dir) {
      struct dirent* entry;
```

```
      int fileCount = 0;

      while ((entry = readdir(dir)) != NULL) {
        if (entry->d_type == DT_REG) {  // Regular file
          char fullPath[64];
          snprintf(fullPath, sizeof(fullPath), "%s/%s", SD_MOUNT_POINT,
              entry->d_name);

          struct stat fileStat;
          if (stat(fullPath, &fileStat) == 0) {
            Serial.printf("   %s (%ld bytes)\n", fullPath, fileStat.
                st_size);
            fileCount++;
          }
        }
      }
      closedir(dir);

      if (fileCount == 0) {
        Serial.println("No files found");
      } else {
        Serial.printf("Total files: %d\n", fileCount);
      }
    }
    xSemaphoreGive(sdCardSemaphore);
  }
}

// ========== Delete SD card files
void deleteSDFile(const char* filename) {
  Serial.printf("Deleting file: %s\n", filename);

  if (xSemaphoreTake(sdCardSemaphore, pdMS_TO_TICKS(2000))) {
    if (unlink(filename) == 0) {                    // file deletion
      Serial.println("File deleted successfully");
    } else {
      perror("Delete error");
    }
    xSemaphoreGive(sdCardSemaphore);
  }
}

// ========== Rename SD card files
void renameSDFile(const char* oldName, const char* newName) {
  Serial.printf("Renaming '%s' to '%s'\n", oldName, newName);

  if (xSemaphoreTake(sdCardSemaphore, pdMS_TO_TICKS(2000))) {
    if (rename(oldName, newName) == 0) {            // rename file
      Serial.println("File renamed successfully");
    } else {
      perror("Rename error");
    }
    xSemaphoreGive(sdCardSemaphore);
  }
}

// ========== Retrieve information from a file
void getFileInfo(const char* filename) {
  Serial.printf(" File info for: %s\n", filename);

  if (xSemaphoreTake(sdCardSemaphore, pdMS_TO_TICKS(2000))) {
    struct stat fileStat;
    if (stat(filename, &fileStat) == 0) {
      Serial.printf(" Size: %ld bytes\n", fileStat.st_size);
```

```
      Serial.printf(" Packets: %ld (estimated)\n", fileStat.st_size /
          sizeof(DataPacket_t));
      Serial.printf(" Duration: %.2f seconds (estimated @ %d Hz)\n",
                    (float)(fileStat.st_size / sizeof(DataPacket_t)) /
                        currentConfig.samplingFrequency,
                    currentConfig.samplingFrequency);

      // Format modification time
      char timeStr[64];
      struct tm* timeinfo = localtime(&fileStat.st_mtime);
      strftime(timeStr, sizeof(timeStr), "%Y-%m-%d %H:%M:%S", timeinfo);
      Serial.printf(" Modified: %s\n", timeStr);
    } else {
      perror("Stat error");
    }
    xSemaphoreGive(sdCardSemaphore);
  }
}
```

B.10 Data Processing methods

```
/* ============== Parser initiation ==============*/
esp_err_t packetParserStart(void) {
  SubsystemConfig_t* packetConfig = &subsystems[SUBSYSTEM_PACKET_PARSER
      ];


  if (packetConfig->tasks.primaryHandle != NULL) {
    return ESP_OK;
  }

  // create task
  BaseType_t result = xTaskCreatePinnedToCore(
    packetParserTask,
    packetConfig->tasks.primaryName,
    packetConfig->tasks.stackSize,
    NULL,
    packetConfig->tasks.primaryPriority,
    &packetConfig->tasks.primaryHandle,
    packetConfig->tasks.primaryCoreId);

  if (result != pdPASS) {
    return ESP_FAIL;
  }

  return ESP_OK;
}

/* ============ Stream Processing ============*/
// =========== Process header
void processStreamHeader(StreamHeader_t* header) {
  SubsystemConfig_t* ecgStoreConfig = &subsystems[SUBSYSTEM_ECG_STORE];
  SubsystemConfig_t* ecgOutConfig = &subsystems[SUBSYSTEM_ECG_OUT];

  const CommandMetadata_t* cmdMeta = &commandMetadata[header->command];

  if (!cmdMeta->requiresData) {     // does not require data
    executeCommand(header->command, NULL, 0, header->numPackets);
}

// =========== Process data
```

```
void processStreamData(uint8_t* streamData, size_t dataSize, uint16_t
    numPackets, uint8_t command) {
  executeCommand(command, streamData, dataSize, numPackets);
}

// ========== Command execution
void executeCommand(uint8_t command, uint8_t* data, size_t dataSize,
    uint16_t numPackets) {
  const CommandMetadata_t* cmdMeta = &commandMetadata[command];

  switch (command) {
    case CMD_ECG_START:
      {
        SubsystemConfig_t* ecgStoreConfig = &subsystems[
            SUBSYSTEM_ECG_STORE];
        if (!ecgStoreConfig->active) {
          if (ecgStoreConfig->init && ecgStoreConfig->init() != ESP_OK)
              {
            return;     // failed to start resources
          }
          if (ecgStoreConfig->start && ecgStoreConfig->start() != ESP_OK
              ) {
            return;    // failed to start task
          }
        }
        break;
      }

    case CMD_ECG_STOP:
      {
        SubsystemConfig_t* ecgStoreConfig = &subsystems[
            SUBSYSTEM_ECG_STORE];
        SubsystemConfig_t* ecgOutConfig = &subsystems[SUBSYSTEM_ECG_OUT
            ];

        // elimination of signal storage resources and task
        if (ecgStoreConfig->active) {
          if (ecgStoreConfig->stop && ecgStoreConfig->stop() == ESP_OK)
              {
            if (ecgStoreConfig->clean) ecgStoreConfig->clean();
          }
        }

        // elimination of signal generation resources and tasks
        if (ecgOutConfig->active) {
          if (ecgOutConfig->stop && ecgOutConfig->stop() == ESP_OK) {
            Serial.println("ECG OUT stopped");
            if (ecgOutConfig->clean) ecgOutConfig->clean();
          }
        }
        break;
      }

    case CMD_ECG_DATA:
      {
        SubsystemConfig_t* ecgStoreConfig = &subsystems[
            SUBSYSTEM_ECG_STORE];

        StreamDataBlock_t dataBlock = {
          .data = data,                 // data block
          .size = dataSize,             // data size (bytes)
          .numPackets = numPackets      // nr single packs
        };
```

```
      if (xQueueSend(ecgStoreConfig->resources.queue, &dataBlock,
          pdMS_TO_TICKS(1000)) != pdTRUE) {
        if (data) {heap_caps_free(data);}
      }
      break;
    }

  case CMD_ECG_OUT:
    {
      // start output task and resources
      SubsystemConfig_t* ecgOutConfig = &subsystems[SUBSYSTEM_ECG_OUT
          ];
      if (!ecgOutConfig->active) {
        if (ecgOutConfig->init && ecgOutConfig->init() != ESP_OK) {
          return;
        }
        if (ecgOutConfig->start && ecgOutConfig->start() != ESP_OK) {
          return;
        }
      }
      break;
    }

  case CMD_CONFIG_UPDATE:
    {
      if (!data || dataSize != sizeof(ConfigPacket_t)) {
        if (data) {heap_caps_free(data);}
        return;
      }

      // update signal parameters
      applyConfiguration((ConfigPacket_t*)data);
      heap_caps_free(data);
      break;
    }

  case CMD_SD_LIST:
    {
      listSDFiles();      // return an SD card file list
      break;
    }

  case CMD_SD_DELETE:
    {
      if (!data || dataSize < sizeof(SDCommandPacket_t)) {
        Serial.println("  Invalid SD delete command data");
        if (data) heap_caps_free(data);
        return;
      }
      SDCommandPacket_t* sdCmd = (SDCommandPacket_t*)data;
      deleteSDFile(sdCmd->filename);      // delete SD files
      heap_caps_free(data);
      break;
    }

  case CMD_SD_RENAME:
    {
      if (!data || dataSize < sizeof(SDCommandPacket_t)) {
        Serial.println("Invalid SD rename command data");
        if (data) heap_caps_free(data);
        return;
      }
      SDCommandPacket_t* sdCmd = (SDCommandPacket_t*)data;
```

```
        renameSDFile(sdCmd->filename, sdCmd->new_filename); // rename
            file
        heap_caps_free(data);
        break;
      }

    case CMD_SD_INFO:
      {
        if (!data || dataSize < sizeof(SDCommandPacket_t)) {
          Serial.println("  Invalid SD info command data");
          if (data) heap_caps_free(data);
          return;
        }
        SDCommandPacket_t* sdCmd = (SDCommandPacket_t*)data;
        getFileInfo(sdCmd->filename);    // return info of a file
        heap_caps_free(data);
        break;
      }

    default:
      {
        Serial.printf("  Unhandled command: %d\n", command);
        if (data) heap_caps_free(data);
        break;
      }
  }
}

// ========== Update System Configuration
void applyConfiguration(ConfigPacket_t* newConfig) {
  Serial.println(" Applying configuration update:");
  Serial.printf(" File: %s -> %s\n", currentConfig.fileName, newConfig->
      fileName);
  Serial.printf(" Sampling: %d Hz -> %d Hz\n",
                currentConfig.samplingFrequency, newConfig->
                    samplingFrequency);

  // Apply new configuration
  memcpy(&currentConfig, newConfig, sizeof(ConfigPacket_t));

  // Update timer if needed
  if (gptimer) {
    gptimer_alarm_config_t alarm_config = {};
    alarm_config.alarm_count = currentConfig.getTimerAlarmUs();
    alarm_config.reload_count = 0;
    alarm_config.flags.auto_reload_on_alarm = true;

    gptimer_set_alarm_action(gptimer, &alarm_config);
    Serial.printf("  Timer updated to %d us\n", currentConfig.
        getTimerAlarmUs());
  }
}
```

B.11 Setup

```
void setup() {
  Serial.begin(115200);

  if (spiStart() != ESP_OK) {           // start SPI interface
    Serial.println("SPI start failed");
  }
```

```
  if (sdCardInit() != ESP_OK) {          // create SD card interface w/
      SPI
    Serial.println("SD card initialization failed");
  }

  if (!testSDCard()) {                   // test SD card
    Serial.println("SD card test failed");
  }

  if (digipotStart() != ESP_OK) {        // add digipot to SPI bus
    Serial.println("Digipots start failed");
  }

  if (IoSpiStart() != ESP_OK) {          // add IO to SPI bus
    Serial.println("IO start failed");
  }

  if (ecgStoreInit() != ESP_OK) {        // start storage resources
    Serial.println("Failed to initialize ECG store resources");
  }

  if (timerStart() != ESP_OK) {          // create hardware timer
    Serial.println("Failed to initialize timer");
  }

  if (dacStart() != ESP_OK) {            // add DAC to SPI bus
    Serial.println("Failed to initialize DAC");
  }

  if (packetParserStart() == ESP_OK) {   // start parser task
    subsystems[SUBSYSTEM_PACKET_PARSER].active = true;
  } else {
    Serial.println("Failed to start packet parser");
  }

  setMCP23S08(MCP23S08_DEV1_WRITE);      // set initial IO1 configs
  setMCP23S08(MCP23S08_DEV2_WRITE);      // set initial IO2 configs

  sdCardSemaphore = xSemaphoreCreateMutex();
  timerSemaphore = xSemaphoreCreateBinary();

  vTaskDelay(pdMS_TO_TICKS(1000));
}
```

Appendix C

Digital Potentiometer Test code

C.1 Packages & Main Variables

```
#include "driver/gpio.h"
#include "esp_err.h"
#include "driver/spi_master.h"
#include "driver/spi_common.h"
#include "hal/spi_types.h"
#include "esp_rom_sys.h"

#include <dirent.h>
#include <sys/stat.h>

// Pin definitions
#define ADC_CS    GPIO_NUM_5
#define ADC_CLOCK GPIO_NUM_33
#define ADC_DATA  GPIO_NUM_32

// MCP42100 command bytes
#define DIGIPOT1_COMMAND_BYTE 0x11  // Command to write to pot 0
#define DIGIPOT2_COMMAND_BYTE 0x12  // Command to write to pot 1
#define DIGIPOT_BOTH_COMMAND_BYTE 0x13  // Command to write to both pots

spi_device_handle_t digipotHandle;
uint8_t currentWiperValue = 0;
#define DIGIPOT_CS GPIO_NUM_22
```

C.2 SPI Interface

```
esp_err_t spiStart() {
  esp_err_t ret;
  // VSPI Configuration
  spi_bus_config_t vspi_bus_config = {
    .mosi_io_num = GPIO_NUM_23,
    .miso_io_num = GPIO_NUM_19,
    .sclk_io_num = GPIO_NUM_18,
    .quadwp_io_num = -1,    // config disabled
    .quadhd_io_num = -1,    // config disabled
    .max_transfer_sz = 4096,
  };

  // initialize VSPI bus
```

```
  ret = spi_bus_initialize(VSPI_HOST, &vspi_bus_config, SPI_DMA_CH_AUTO)
      ;
  if (ret != ESP_OK) {
    Serial.printf("Failed to initialize VSPI bus: %s\n", esp_err_to_name
        (ret));
  }
  return ret;
}

esp_err_t digipotStart() {
  esp_err_t ret;
  // digipot Configuration
  spi_device_interface_config_t vspi_digipot_config = {
    .command_bits = 0,
    .address_bits = 0,
    .mode = 0,
    .duty_cycle_pos = 128,
    .clock_speed_hz = 1000000,  // 1MHz
    .spics_io_num = DIGIPOT_CS,
    .queue_size = 5,
  };

  // add digipot to VSPI
  ret = spi_bus_add_device(VSPI_HOST, &vspi_digipot_config, &
      digipotHandle);
  if (ret != ESP_OK) {
    Serial.printf("Failed to add Digital Potentiometer to VSPI: %s\n",
        esp_err_to_name(ret));
  } else {
    Serial.println("Digital Potentiometer added to VSPI");
  }
  return ret;
}
```

C.3 Main Methods

```
uint16_t readADS7822() {
  uint16_t result = 0;

  // Ensure clock starts low
  gpio_set_level((gpio_num_t)ADC_CLOCK, 0);  // CLK set low
  esp_rom_delay_us(10);      // wait for 10 us

  // Start conversion
  gpio_set_level((gpio_num_t)ADC_CS, 0); // CS set low
  esp_rom_delay_us(10);

  // Skip first two clocks/bits (see datasheet)
  for (int i = 0; i < 2; i++) {
    gpio_set_level((gpio_num_t)ADC_CLOCK, 1);
    esp_rom_delay_us(10);
    gpio_set_level((gpio_num_t)ADC_CLOCK, 0);
    esp_rom_delay_us(10);
  }

  // Read 12 data bits
  for (int i = 11; i >= 0; i--) {
    // Rising edge
    gpio_set_level((gpio_num_t)ADC_CLOCK, 1);
    esp_rom_delay_us(10);
```

```
    // Falling edge - new data appears
    gpio_set_level((gpio_num_t)ADC_CLOCK, 0);
    esp_rom_delay_us(5);

    // Read data
    if (gpio_get_level((gpio_num_t)ADC_DATA)) {
      result |= (1 << i);   // bit sequence construction
    }

    esp_rom_delay_us(5);
  }

  // End conversion
  gpio_set_level((gpio_num_t)ADC_CS, 1);    // CS set high
  esp_rom_delay_us(20);

  return result;     // new conversion with new function call
}

void setWiper(uint8_t commandByte, uint8_t valueByte) {

  spi_transaction_t transaction = {};
  memset(&transaction, 0, sizeof(transaction));

  // Set up transaction
  /*Send TX DATA instead of TX buffer pointer*/
  transaction.length = 16;                    // 16 bits (2 bytes)
  transaction.flags = SPI_TRANS_USE_TXDATA;

  transaction.tx_data[0] = commandByte;
  transaction.tx_data[1] = valueByte;

  // Execute transaction
  esp_err_t result = spi_device_polling_transmit(digipotHandle, &
     transaction);
  if (result != ESP_OK) {
    Serial.printf("ERROR: Failed to set wiper: %s\n", esp_err_to_name(
       result));
  }
}
```

C.4 Setup

```
void setup() {
  Serial.begin(115200);
  delay(100);

  // Initialize SPI bus first
  spiStart() != ESP_OK) {
    Serial.println(" SPI start failed");
  }

  // Initialize digital potentiometer
  if (digipotStart() != ESP_OK) {
    Serial.println(" Digipots start failed");
  }

  // Configure ADC pins (bit-banged)
  gpio_config_t io_conf = {};   // config structure

  // Output pins config
```

```
  io_conf.intr_type = GPIO_INTR_DISABLE;     // no interrupts
  io_conf.mode = GPIO_MODE_OUTPUT;  // output GPIOs
  io_conf.pin_bit_mask = (1ULL << ADC_CS) | (1ULL << ADC_CLOCK); //
     output pins
  io_conf.pull_down_en = GPIO_PULLDOWN_DISABLE;     // disable pull-down
      resistors
  io_conf.pull_up_en = GPIO_PULLUP_DISABLE;     // disable pull-up
     resistors
  gpio_config(&io_conf);

  // Input pin config
  io_conf.mode = GPIO_MODE_INPUT;   // input set
  io_conf.pin_bit_mask = (1ULL << ADC_DATA);    // input pin
  io_conf.pull_down_en = GPIO_PULLDOWN_ENABLE;  // enable pull-down
     resistors
  gpio_config(&io_conf);

  // Initialize ADC pins
  gpio_set_level((gpio_num_t)ADC_CS, 1);
  gpio_set_level((gpio_num_t)ADC_CLOCK, 0);
}
```

C.5 Loop

```
void loop() {
  // set wiper and increment
  setWiper(DIGIPOT_BOTH_COMMAND_BYTE, currentWiperValue);
  currentWiperValue += 1;

  uint16_t adcValue;
  uint16_t sum;
  for (i=0; i<20; i++) {
    adcValue = readADS7822();
    sum += adcValue;
    i++;
  }

  // read adc values and convert to voltage
  adcValue = sum;
  float voltageADC = (adcValue / 4096.0) * 3.3;

  // expected values
  float resistanceA = ((256.0 - currentWiperValue)/256) * 100000.0;
  float resistanceB = (currentWiperValue / 256.0) * 100000.0;
  float voltage = resistanceB / (resistanceA+resistanceB) * 3.3;

  // print results
  Serial.printf("Step: %d/255 | rA %.3f | rB %.3f | V= %.3f | ADC/
     Voltage = %4d/%.3f\n",
              currentWiperValue, resistanceA, resistanceB, voltage,
                 adcValue, voltageADC);

  delay(100);
}
```

Appendix D

Analog Circuit Test code

D.1 Packages & Main Variables

```
#include "driver/spi_master.h"
#include "driver/spi_common.h"
#include "hal/spi_types.h"
#include "esp_err.h"
#include <math.h>
#include <string.h>

#define M_PI 3.14159265358979323846

#define DAC_A_WRITE_AND_LOAD 0x10 // Write to input register A and
    update DAC A
#define DAC_CS GPIO_NUM_15

spi_device_handle_t dacHandle;

// Chirp parameters structure
typedef struct {
    double startFreq;  // start frequency
    double endFreq;   // end frequency
    double duration;  // one-way sweep (secs)
    double sampleRate;
    double amplitude;  // amplitude
    uint32_t currentSample;   // current sample index
    uint32_t totalSamples;  // total samples in one-way sweep
    double currentPhase;    // current phase for continuous transition
    bool upChirp;   // True=up, false=down
} ChirpGenerator;

// Global chirp generator (formerly chirpA, now just chirp)
ChirpGenerator chirp;

TickType_t lastWakeTime;
const TickType_t sampleFrequency = pdMS_TO_TICKS(1);
```

D.2 SPI Interface

```
esp_err_t spiStart() {
  esp_err_t ret;
  // HSPI Configuration
  spi_bus_config_t hspi_bus_config = {
```

```
    .mosi_io_num = GPIO_NUM_12,
    .miso_io_num = -1,
    .sclk_io_num = GPIO_NUM_14,
    .quadwp_io_num = -1,
    .quadhd_io_num = -1,
    .max_transfer_sz = 4096,
};

  // initialize HSPI bus
  ret = spi_bus_initialize(HSPI_HOST, &hspi_bus_config, SPI_DMA_CH_AUTO)
      ;
  if (ret != ESP_OK) {
    Serial.printf("Failed to initialize HSPI bus: %s\n", esp_err_to_name
        (ret));
  }
  return ret;
}
```

```
esp_err_t dacStart() {
    esp_err_t ret;
    spi_device_interface_config_t hspi_dac_config = {
        .command_bits = 0,
        .address_bits = 0,
        .mode = 1,
        .duty_cycle_pos = 128,
        .clock_speed_hz = 1000000,  // 1MHz
        .spics_io_num = DAC_CS,
        .queue_size = 5,
    };

    ret = spi_bus_add_device(HSPI_HOST, &hspi_dac_config, &dacHandle);
    if (ret != ESP_OK) {
        Serial.printf("Failed to add DAC to HSPI: %s\n", esp_err_to_name
            (ret));
    } else {
        Serial.println("DAC added to HSPI");
    }
    return ret;
}
```

D.3 Main Methods

```
void setDAC(uint8_t command, uint16_t dataBytes) {
    uint8_t tx_data[3];

    tx_data[0] = command;
    tx_data[1] = (dataBytes >> 8) & 0xFF;
    tx_data[2] = (dataBytes & 0xFF);

    spi_transaction_t transaction;
    memset(&transaction, 0, sizeof(transaction));

    transaction.length = 24;
    transaction.tx_buffer = tx_data;
    transaction.flags = 0;

    esp_err_t ret = spi_device_polling_transmit(dacHandle, &transaction)
        ;
    if (ret != ESP_OK) {
```

```
        Serial.printf("ERROR: Failed to set DAC 0x%02X 0x%02X 0x%02X: %s
            \n", tx_data[0], tx_data[1], tx_data[2], esp_err_to_name(ret)
            );
    }
}
```

```
void initChirp(ChirpGenerator* chirp, double startFreq, double endFreq,
               double duration, double sampleRate, double amplitude) {
    chirp->startFreq = startFreq;
    chirp->endFreq = endFreq;
    chirp->duration = duration;
    chirp->sampleRate = sampleRate;
    chirp->amplitude = amplitude;
    chirp->currentSample = 0;
    chirp->totalSamples = (uint32_t)(duration * sampleRate);
    chirp->currentPhase = 0.0;
    chirp->upChirp = true;
}
```

```
double generateChirpSample(ChirpGenerator* chirp) {
    // 1. Check for reversal
    if (chirp->currentSample >= chirp->totalSamples) {
        chirp->currentSample = 0;
        chirp->upChirp = !chirp->upChirp;
    }

    double t = (double)chirp->currentSample / chirp->sampleRate;
    double f_start, f_end;

    // Frequencies of the current sweep
    if (chirp->upChirp) {
        f_start = chirp->startFreq;
        f_end = chirp->endFreq;
    } else {
        f_start = chirp->endFreq; // Start frequency for down-chirp
        f_end = chirp->startFreq; // End frequency for down-chirp
    }

    // 2. Calculate the instantaneous frequency
    double instantFreq = f_start + (f_end - f_start) * (t / chirp->
        duration);

    // 3. Calculate the phase increment for this sample and accumulate
        it
    double phaseIncrement = 2.0 * M_PI * instantFreq / chirp->sampleRate
        ;
    chirp->currentPhase += phaseIncrement;

    //Wrap phase to [0, 2pi) for numerical stability
    while (chirp->currentPhase >= 2.0 * M_PI) {
        chirp->currentPhase -= 2.0 * M_PI;
    }
    while (chirp->currentPhase < 0.0) {
        chirp->currentPhase += 2.0 * M_PI;
    }

    double sample = chirp->amplitude * sin(chirp->currentPhase);

    chirp->currentSample++;
    return sample;
}
```

```
uint16_t chirpToDAC(double sample) {
    int32_t dacValue = 32768 + (int32_t)(sample * 32767);

    if (dacValue > 65535) dacValue = 65535;
    if (dacValue < 0) dacValue = 0;

    return (uint16_t)dacValue;
}
```

```
void updateDAC_Chirp() {
    // Calculate sample for the single chirp generator
    double sample = generateChirpSample(&chirp);

    // Convert sample to 16-bit DAC value
    uint16_t dacValue = chirpToDAC(sample);

    // Send the DAC value to Channel A
    setDAC(DAC_A_WRITE_AND_LOAD, dacValue);
}
```

D.4 Setup

```
void setup() {
    Serial.begin(115200);
    vTaskDelay(pdMS_TO_TICKS(1000));

    if (spiStart() != ESP_OK) {
        Serial.println("SPI start failed. Aborting setup.");
        return;
    }

    if (dacStart() != ESP_OK) {
        Serial.println("DAC start failed. Aborting setup.");
        return;
    }

    initChirp(&chirp, 0.05, 500.0, 1.0, 1000.0, 1.0);

    lastWakeTime = xTaskGetTickCount();

    Serial.println("System ready - starting main loop.");
}
```

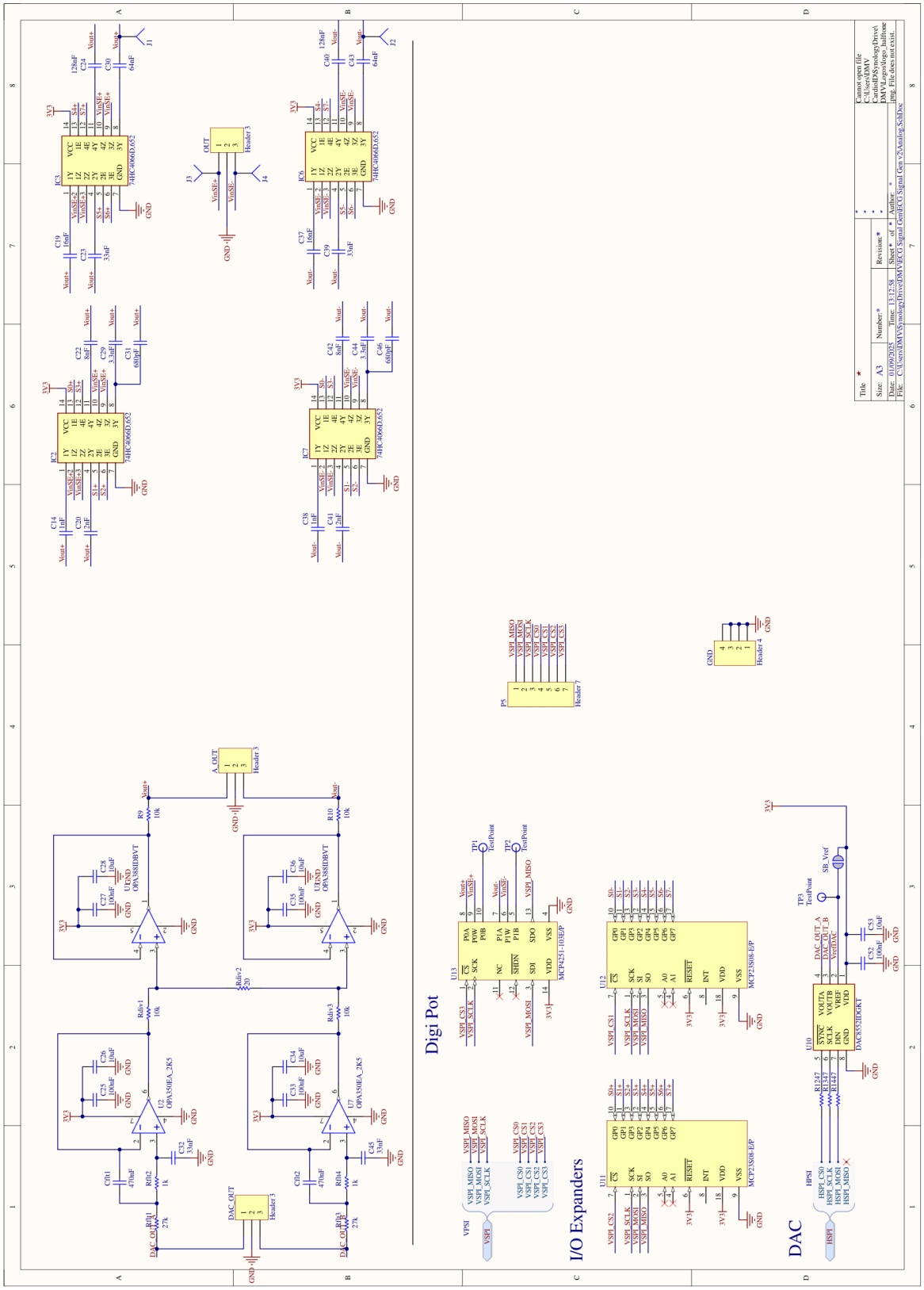
D.5 Loop

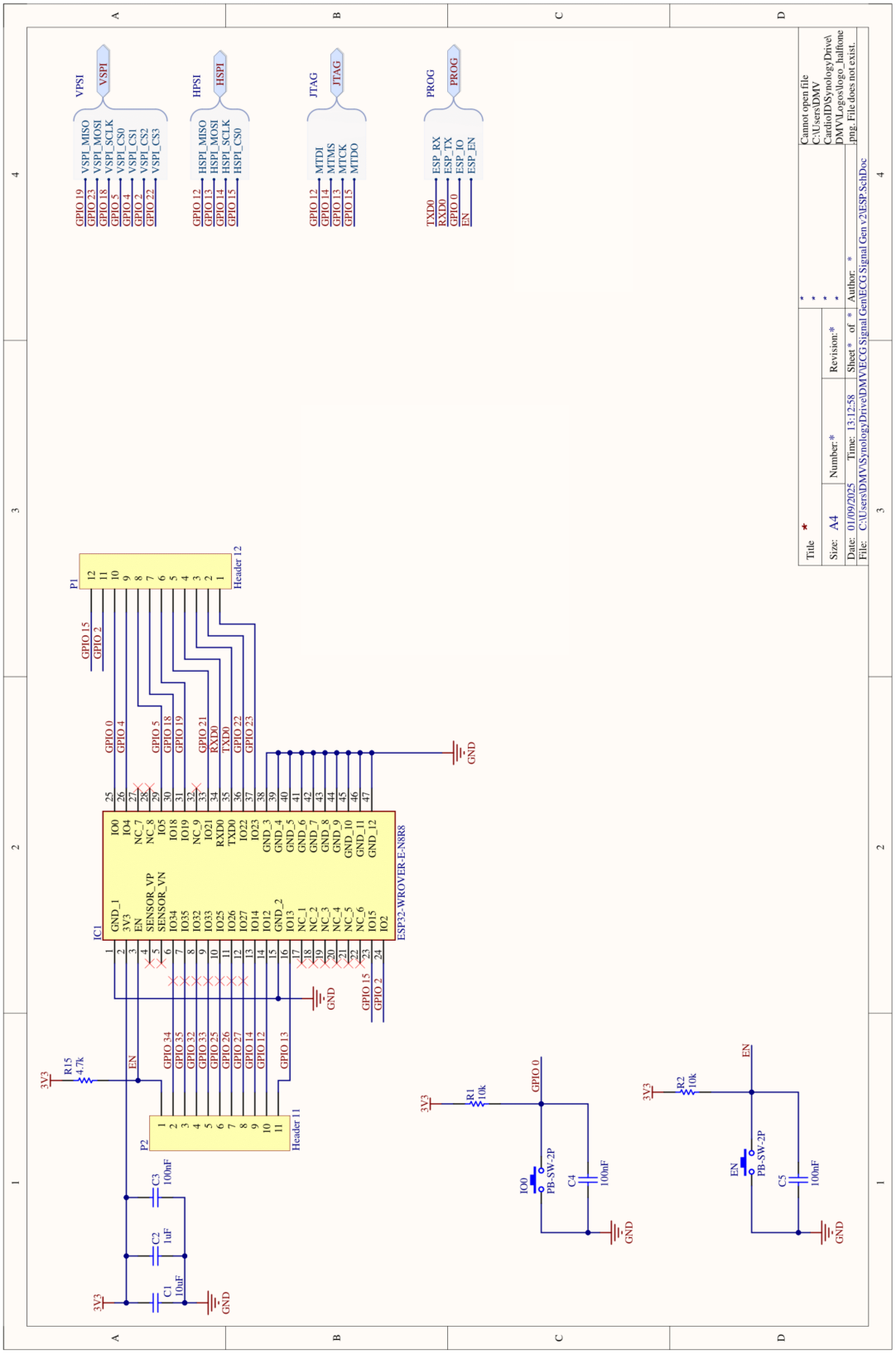
```
void loop() {
    updateDAC_Chirp();
    vTaskDelayUntil(&lastWakeTime, sampleFrequency);
}
```

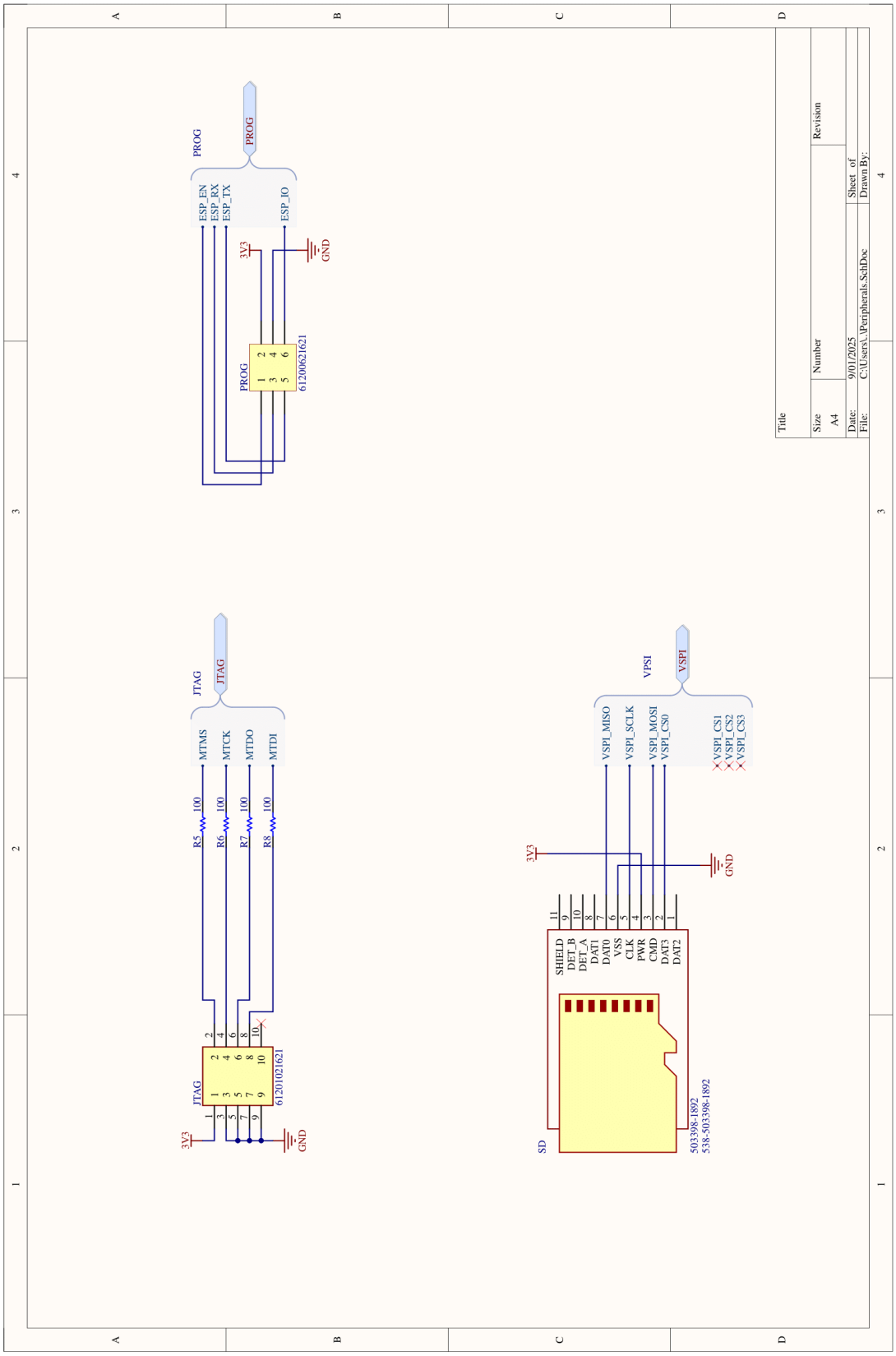
Appendix E

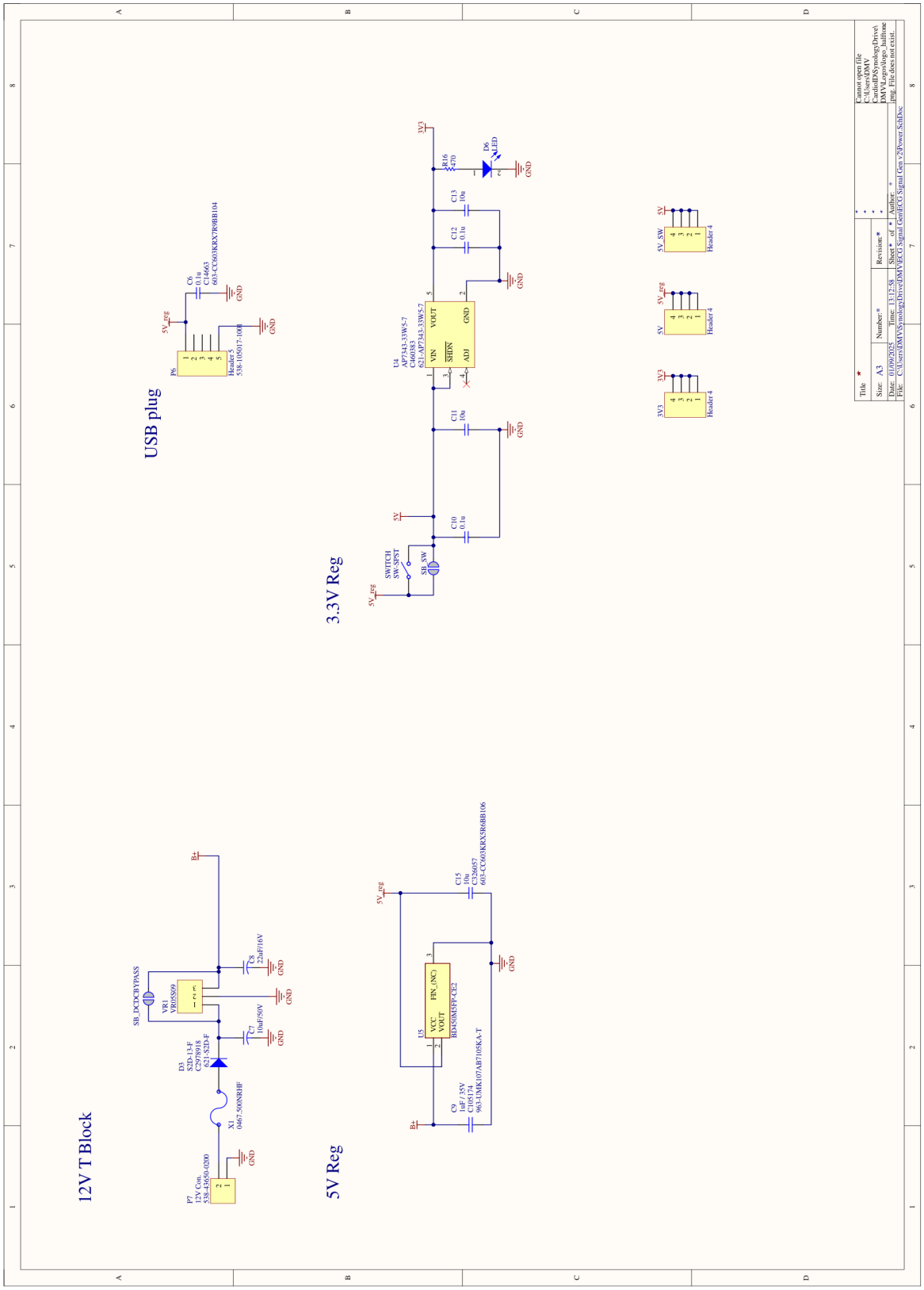
PCB Design Schematic

(Continue in the next page)


Digi Pot
I/O Expanders
DAC
Header3
Header4
Header7
MCP4251-103E/P
MCP23S08-E/P
DAC8552IDGKT
OPA388IDBVT
OPA388EA_2K5
74HC4066D,652


VPSI
VSPI_MISO
VSPI_MOSI
VSPI_SCLK
VSPI_CS0
VSPI_CS1
VSPI_CS2
VSPI_CS3
GPIO 19
GPIO 23
GPIO 18
GPIO 5
GPIO 4
GPIO 2
GPIO 22
HPSI
HSPI_MISO
HSPI_MOSI
HSPI_SCLK
HSPI_CS0
GPIO 12
GPIO 13
GPIO 14
GPIO 15
JTAG
MTDI
MTMS
MTCK
MTDO
PROG
ESP_RX
ESP_TX
ESP_IO
ESP_EN
TXD0
RXD0
GPIO 0
EN
P1
Header 12
P2
Header 11
IC1
ESP32-WROVER-E-N8R8
R15
4.7k
R1
10k
R2
10k
C1
10nF
C2
1uF
C3
100nF
C4
100nF
C5
100nF
IO0
PB-SW-2P
3V3
GND
Title *
Size: A4
Number:*
Revision:*
Date: 01/09/2025
Time: 13:12:58
Sheet * of *
Author: *
File: C:\Users\DMV\Synology Drive\DMV\ECG Signal Gen\ECG Signal Gen.v2\ESP-SchDoc
Cannot open file C:\Users\DMV Cardio\DS\Synology Drive\DMV\Logos\logo_halftone.png. File does not exist.

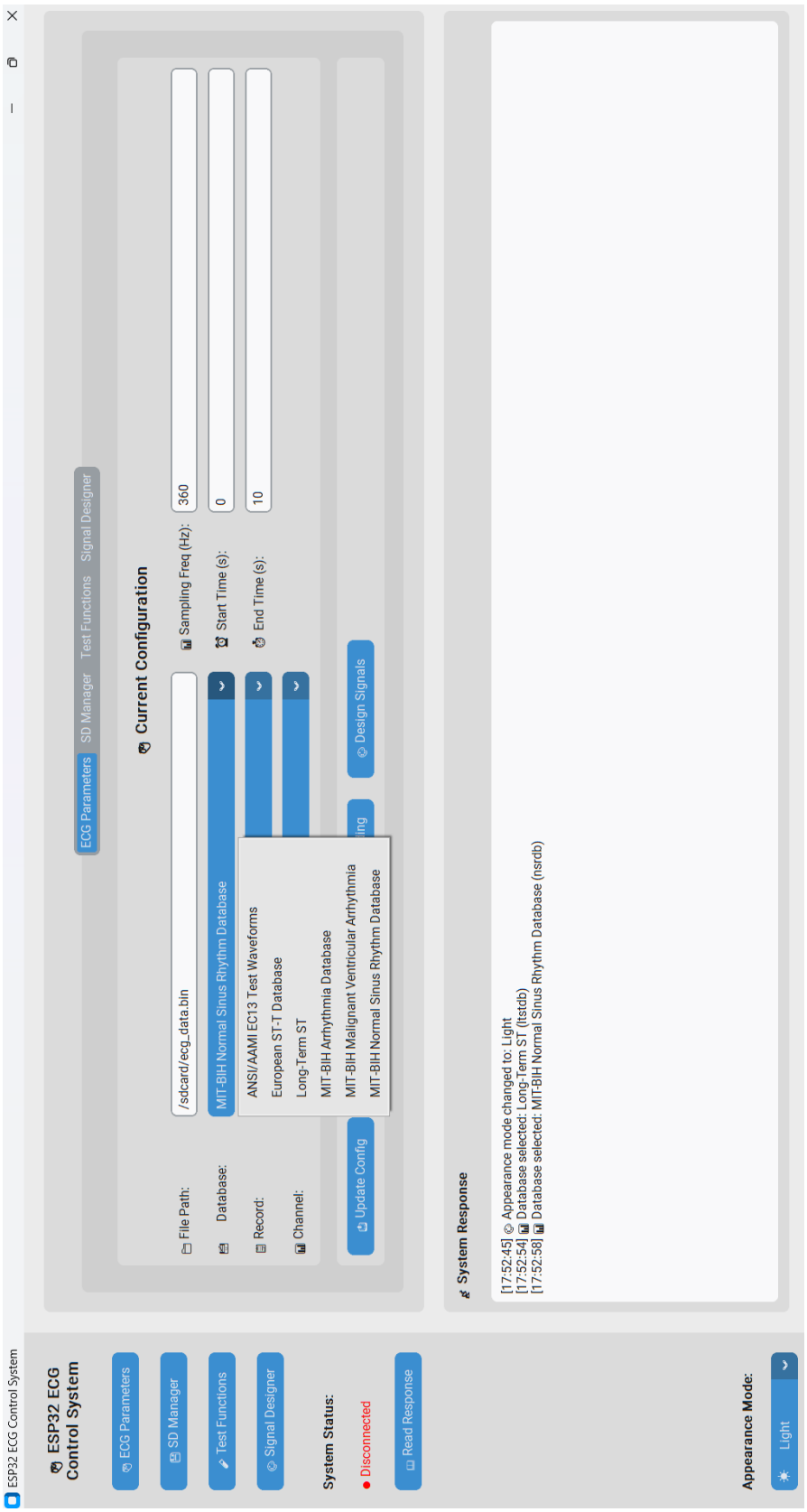

PROG
ESP_EN
ESP_RX
ESP_TX
ESP_IO
3V3
GND
PROG
61200621621
JTAG
MTMS
MTCK
MTDO
MTDI
R5
R6
R7
R8
100
61201021621
VPSI
VSPI_MISO
VSPI_SCLK
VSPI_MOSI
VSPI_CS0
VSPI_CS1
VSPI_CS2
VSPI_CS3
SHIELD
DET_B
DET_A
DAT1
DAT0
VSS
CLK
PWR
CMD
DAT3
DAT2
SD
503398-1892
538-503398-1892
Title
Size
A4
Number
Revision
Date:
9/01/2025
Sheet of
File:
C:\Users\...\Perripherals.SchDoc
Drawn By:


12V T Block
5V Reg
3.3V Reg
USB plug

Appendix F

Proposed GUI - ECG device

(Continue in the next page)


ESP32 ECG Control System
ESP32 ECG Control System
ECG Parameters
SD Manager
Test Functions
Signal Designer
System Status:
Disconnected
Read Response
Appearance Mode:
Light
ECG Parameters
SD Manager
Test Functions
Signal Designer
Current Configuration
File Path:
/sdcard/ecg_data.bin
Database:
MIT-BIH Normal Sinus Rhythm Database
Record:
Channel:
Sampling Freq (Hz):
360
Start Time (s):
0
End Time (s):
10
ANSI/AAMI EC13 Test Waveforms
European ST-T Database
Long-Term ST
MIT-BIH Arrhythmia Database
MIT-BIH Malignant Ventricular Arrhythmia
MIT-BIH Normal Sinus Rhythm Database
Update Config
Design Signals
System Response
[17:52:45] Appearance mode changed to: Light
[17:52:54] Database selected: Long-Term ST (ltstdb)
[17:52:58] Database selected: MIT-BIH Normal Sinus Rhythm Database (nsrdb)

References

[1] Hongling Zhu et al. “Automatic multilabel electrocardiogram diagnosis of heart rhythm or conduction abnormalities with deep learning: a cohort study”. In: *The Lancet Digital Health* 2 (7 July 2020), E348–E357. ISSN: 25897500. DOI: 10.1016/S2589-7500(20)30107-2.

[2] Krittika Goyal, David A. Borkholder, and Steven W. Day. “Dependence of Skin-Electrode Contact Impedance on Material and Skin Hydration”. In: *Sensors* 22 (21 Nov. 2022), p. 8510. ISSN: 1424-8220. DOI: 10.3390/s22218510.

[3] C M Vidhya, Yogita Maithani, and Jitendra P Singh. “Recent Advances and Challenges in Textile Electrodes for Wearable Biopotential Signal Monitoring: A Comprehensive Review.” In: *Biosensors* 13 (7 June 2023). ISSN: 2079-6374. DOI: 10.3390/bios13070679.

[4] John E. Hall and Michael E. Hall. *Guyton and Hall Textbook of Medical Physiology*. 14th ed. Elsevier, 2021, pp. 51–133. ISBN: 978-0-323-67280-1. URL: https://archive.org/details/guyton-and-hall-textbook-of-medical-physiology-14ed/page/n2/mode/1up?view=theater.

[5] Jaakko Malmivuo and Robert Plonsey. *BioelectromagnetismPrinciples and Applications of Bioelectric and Biomagnetic Fields*. Oxford University Press, Oct. 1995. ISBN: 9780195058239. DOI: 10.1093/acprof:oso/9780195058239.001.0001.

[6] Vilem Kledrowetz et al. “A 1 V 92 dB SNDR 10 kHz Bandwidth Second-Order Asynchronous Delta-Sigma Modulator for Biomedical Signal Processing”. In: *Sensors* 20 (July 2020), p. 4137. DOI: 10.3390/s20154137.

[7] K. Sembulingam and Prema Sembulingam. *Essentials of Medical Physiology*. 6th ed. Jaypee Brothers Medical Publishers (P) Ltd., 2012, pp. 519–587. ISBN: 978-93-5025-936-8. URL: https://www.siaphysio.com/e-books/Biochemistry/KSembulingam%20_EssentialsofMedical_Physiology,%206th%20Edition.pdf.

[8] OpenStax. *Cardiac Muscle and Electrical Activity | Anatomy and Physiology II*. 2025. URL: https://courses.lumenlearning.com/suny-ap2/chapter/cardiac-muscle-and-electrical-activity/.

[9] Xingyu Wei, Sandesh Yohannan, and John R. Richards. “Physiology, Cardiac Repolarization Dispersion and Reserve”. In: *StatPearls* (Apr. 2023). URL: https://www.ncbi.nlm.nih.gov/books/NBK537194/.

[10] Alfonso Bueno-Orovio et al. “Na/K pump regulation of cardiac repolarization: insights from a systems biology approach”. In: *Pflügers Archiv - European Journal of Physiology* 466 (2 Feb. 2014), pp. 183–193. ISSN: 0031-6768. DOI: 10.1007/s00424-013-1293-1.

[11] Paul A. Iaizzo. *The Cardiac Action Potentials*. 2025. URL: https://www.vhlab.umn.edu/atlas/conduction-system-tutorial/cardiac-action-potentials.shtml.

[12] Inês Pinto et al. *Electrophysiology of the Heart and the Electrocardiogram: Visual Depictions*. Dec. 2020.

[13] Moises Rivera-Ruiz, Christian Cajavilca, and Joseph Varon. “Einthoven’s string galvanometer: the first electrocardiograph.” In: *Texas Heart Institute journal* 35 (2 2008), pp. 174–8. ISSN: 0730-2347.

[14] Majd AlGhatrif and Joseph Lindsay. “A brief review: history to understand fundamentals of electrocardiography.” In: *Journal of community hospital internal medicine perspectives* 2 (1 2012). ISSN: 2000-9666. DOI: 10.3402/jchimp.v2i1.14383.

[15] Larisa G Tereshchenko and Mark E Josephson. “Frequency content and characteristics of ventricular conduction.” In: *Journal of electrocardiology* 48 (6 2015), pp. 933–7. ISSN: 1532-8430. DOI: 10.1016/j.jelectrocard.2015.08.034.

[16] The University of Nottingham. *Bipolar Leads - ECG Lead Placement - Normal Function of the Heart - Cardiology Teaching Package - Practice Learning - Division of Nursing - The University of Nottingham*. 2025. URL: https://www.nottingham.ac.uk/nursing/practice/resources/cardiology/function/bipolar_leads.php.

[17] Johnson Francis. “ECG monitoring leads and special leads”. In: *Indian Pacing and Electrophysiology Journal* 16 (3 May 2016), pp. 92–95. ISSN: 09726292. DOI: 10.1016/j.ipej.2016.07.003. URL: https://pubmed.ncbi.nlm.nih.gov/3653124/.

[18] Ateeq Mubarik and Arshad Muhammad Iqbal. *Holter Monitor*. StatPearls Publishing LLC, 2025.

[19] Abdelrahman Abdou and Sridhar Krishnan. “Horizons in Single-Lead ECG Analysis From Devices to Data”. In: *Frontiers in Signal Processing* 2 (Apr. 2022). ISSN: 2673-8198. DOI: 10.3389/frsip.2022.866047.

[20] Hani Yousef et al. *Anatomy, Skin (Integument), Epidermis*. StatsPearls Publishing, 2025.

[21] Liangtao Yang et al. “Insight into the Contact Impedance between the Electrode and the Skin Surface for Electrophysical Recordings”. In: *ACS Omega* 7 (16 Apr. 2022), pp. 13906–13912. ISSN: 2470-1343. DOI: 10.1021/acsomega.2c00282.

[22] Yulin Fu et al. “Dry Electrodes for Human Bioelectrical Signal Monitoring”. In: *Sensors* 20 (13 June 2020), p. 3651. ISSN: 1424-8220. DOI: 10.3390/s20133651.

[23] John G.. Webster and John W.. Clark. *Medical instrumentation : application and design*. John Wiley & Sons, 2010, p. 713. ISBN: 9780471676003.

[24] Fleur V Y Tjong et al. “Utilization of and perceived need for simulators in clinical electrophysiology: results from an EHRA physician survey”. In: *Europace* 26 (2 Feb. 2024). ISSN: 1099-5129. DOI: 10.1093/europace/euae037.

[25] Fluke Biomedical. “ProSim 8 / ProSim 8P Vital Signs and ECG Patient Simulator Leading Edge Performance in Vital Signs Simulation”. In: *Fluke Biomedical* (2025). URL: https://www.flukebiomedical.com/products/patient-simulators/prosim-8-and-prosim-8p-vital-signs-patient-simulator.

[26] Rigel Medical. “RIGEL MEDICAL PatSim 200 USER MANUAL 1 Rigel Medical 5 year Warranty Statement”. In: *Rigel Medical* (2021). URL: www.rigelmedical.com/calibration.

[27] MedTec&Science GmbH. “INSTRUCTION MANUAL MS410 ECG Simulator”. In: *MedTec & Science GmbH* (May 2021). URL: www.ms-gmbh.de.

[28] Peter M van Dam, Thom F Oostendorp, and Adriaan van Oosterom. “ECGSIM: Interactive simulation of the ECG for teaching and research purposes”. In: *2010 Computing in Cardiology*. 2010, pp. 841–844.

[29] ECGSIM. *ECGSIM Manual: Introduction*. 2019. URL: https://www.ecgsim.org/manual/.

[30] Mabrook Al-Rakhami and Ahmad Alhamed. “Cloud-based Graphical Simulation Tool of ECG for Educational Purpose”. In: *Proceedings of the International Conference on Internet of things and Cloud Computing*. ACM, Mar. 2016, pp. 1–6. ISBN: 9781450340632. DOI: 10.1145/2896387.2896410.

[31] Regina Célia Coelho, Nayara Consuelo Gomes Rangel Lourenço, and Carlos Marcelo Gurjão de Godoy. “A mobile device tool to assist the ECG interpretation based on a realistic 3D virtual heart simulation”. In: *SIMULATION* 94 (6 June 2018), pp. 465–476. ISSN: 0037-5497. DOI: 10.1177/0037549717733038.

[32] Fadilla Putri Devito Nur Azizah, Bambang Guruh Irianto, and Endro Yulianto. “Twelve Channel ECG Phantom Based on MEGA2560 and DAC-MCP4921”. In: *Jurnal Teknokes* 14 (2 Oct. 2021), pp. 73–79. ISSN: 2407-8964. DOI: 10.35882/teknokes.v14i2.5.

[33] Daniel Almeida, João Costa, and André Lourenço. “ECG simulator with configurable skin-electrode impedance and artifacts emulation”. In: *Biomedical Physics & Engineering Express* 7 (6 Nov. 2021), p. 065026. ISSN: 2057-1976. DOI: 10.1088/2057-1976/ac2b4e.

[34] Daniel Gonçalves Pita Santos de Almeida. *Non-intrusive ECG acquisition test-bed*. 2018.

[35] Mario Alan Quiroz-Juárez et al. “ECG Patient Simulator Based on Mathematical Models.” In: *Sensors* 22 (15 July 2022). ISSN: 1424-8220. DOI: 10.3390/s22155714.

[36] Mario A. Quiroz-Juárez et al. “Cardiac Conduction Model for Generating 12 Lead ECG Signals With Realistic Heart Rate Dynamics”. In: *IEEE Transactions on NanoBioscience* 17 (4 Oct. 2018), pp. 525–532. ISSN: 1536-1241. DOI: 10.1109/TNB.2018.2870331.

[37] A L Goldberger et al. “PhysioBank, PhysioToolkit, and PhysioNet: components of a new research resource for complex physiologic signals.” In: *Circulation* 101 (23 June 2000), E215–20. ISSN: 1524-4539. DOI: 10.1161/01.cir.101.23.e215.

[38] Chen Xie et al. “Waveform Database Software Package (WFDB) for Python (version 4.1.0)”. In: *PhysioNet* (Jan. 2023).

[39] Microchip Technology Inc. *MCP41XXX/42XXX Single/Dual Digital Potentiometer with SPI™ Interface*. July 2003. URL: https://pt.mouser.com/datasheet/3/282/1/11195c.pdf.

[40] Microchip Technology Inc. *MCP23008/MCP23S08 8-Bit I/O Expander with Serial Interface*. 2022. URL: https://pt.mouser.com/datasheet/2/268/MCP23008_MCP23S08_Data_Sheet_DS20001919-3441913.pdf.

[41] ROHM Semiconductor. *500-mA 3.3-V or 5.0-V Output LDO Regulators*. 2015. URL: https://fscdn.rohm.com/en/products/databook/datasheet/ic/power/linear_regulator/bd4xxm5xxx-c-e.pdf.

[42] Diodes Incorporated. *AP7343 300mA HIGH PSRR LOW NOISE LDO WITH ENABLE*. Jan. 2021. URL: https://4donline.ihs.com/images/VipMasterIC/IC/DIOD/DIOD-S-A0011482403/DIOD-S-A0011743607-1.pdf?hkey=CECEF36DEECDED6468708AAF2E19C0C6.

[43] Ltd Espressif Systems Co. *ESP32-WROVER-E & ESP32-WROVER-IE Datasheet Version 2.1*. 2025. URL: https://www.espressif.com/sites/default/files/documentation/esp32-wrover-e_esp32-wrover-ie_datasheet_en.pdf.

[44] TDK Corporation. *TDK Flash Storage Catalog*. 2025. URL: https://product.tdk.com/system/files/dam/doc/content/flash-storage/en/flashstorage_catalog_en.pdf.

[45] Ltd Espressif Systems Co. *Introduction to the ESP-Prog Board - - — ESP-IoT-Solution latest documentation*. 2025. URL: https://docs.espressif.com/projects/esp-iot-solution/en/latest/hw-reference/ESP-Prog_guide.html.

[46] FreeRTOS. *RTOS Fundamentals - FreeRTOS™*. 2024. URL: https://www.freertos.org/Documentation/01-FreeRTOS-quick-start/01-Beginners-guide/01-RTOS-fundamentals.

[47] Ltd Espressif Systems Co. *System - ESP32 - — ESP-IDF Programming Guide latest documentation*. 2025. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/migration-guides/release-6.x/6.0/system.html#freertos.

[48] Visual Paradigm. *UML Class Diagram Tutorial*. 2025. URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/.

[49] Piyu Dhaker. *Introduction to SPI Interface*. Sept. 2018. URL: https://www.analog.com/media/en/analog-dialogue/volume-52/number-3/introduction-to-spi-interface.pdf.

[50] Ltd Espressif Systems Co. *SPI Master Driver - ESP32 - — ESP-IDF Programming Guide latest documentation*. 2025. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/spi_master.html.

[51] Arduino Forum. *SPI MISO-MOSI arduino - Projects / Programming - Arduino Forum*. Apr. 2014. URL: https://forum.arduino.cc/t/spi-miso-mosi-arduino/224170/3.

[52] Texas Instruments. *DAC8552 16-BIT, DUAL CHANNEL, ULTRA-LOW GLITCH, VOLTAGE OUTPUT DIGITAL-TO-ANALOG CONVERTER*. Oct. 2006. URL: https://www.ti.com/lit/ds/symlink/dac8552.pdf?ts=1756732980710.

[53] Nexperia. *74HC4066 Quad single-pole single-throw analog switch*. May 2025. URL: https://assets.nexperia.com/documents/data-sheet/74HC_HCT4066.pdf.

[54] Ltd Espressif Systems Co. *FAT Filesystem Support - ESP32 - — ESP-IDF Programming Guide v5.5.1 documentation*. 2025. URL: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/storage/fatfs.html.

[55] Texas Instruments. *OPAx350 High-Speed, Single-Supply, Rail-to-Rail Operational Amplifiers MicroAmplifier Series*. Dec. 2015. URL: https://www.ti.com/lit/ds/symlink/opa2350.pdf?ts=1756664444094&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FOPA2350.

[56] Texas Instruments. *OPAx388 Precision, Zero-Drift, Zero-Crossover, True Rail-to-Rail, Input/Output Operational Amplifiers*. July 2020. URL: https://www.ti.com/lit/ds/symlink/opa388.pdf?ts=1756658354712&ref_url=https%253A%252F%252Fpt.mouser.com%252F.

[57] Alan V. Oppenheim, Alan S. Willsky, and Ian T. Young. *Signals and systems*. Prentice-Hall, 1983. ISBN: 0138111758.

[58] Texas Instruments. *ADS129x Low-Power, 2-Channel, 24-Bit Analog Front-End for Biopotential Measurements*. Apr. 2020. URL: https://www.ti.com/lit/ds/sbas502c/sbas502c.pdf?ts=1756589948842.

[59] Sverre. Grimnes and Ørjan G.. Martinsen. *Bioimpedance and bioelectricity basics*. Academic Press, 2015, p. 585. ISBN: 9780124114708.