Politecnico di Milano
Scuola di Ingegneria dell'Informazione

Polo territoriale di Como
Master of Science in Computer Engineering

ECG-ira: An efficient mobile app for ECG analysis

Supervisor: Prof. Giuseppe Pozzi
Co-Supervisor: Ulisse Pizzagalli
Assistant Supervisor: Eng. Name Surname

Master Graduation Thesis by:
Antonello Fodde - 817371
Chai Botta - 817333

Academic Year 2015-2016

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur malesuada suscipit nisl, vitae gravida odio imperdiet a. Etiam sed auctor tellus. Donec sed mauris eget nibh luctus accumsan. Sed imperdiet purus in elit iaculis, non commodo nisl varius. Integer sit amet diam laoreet, viverra lacus id, placerat nisl. Aenean ultricies sollicitudin elit in sodales. Vivamus euismod eleifend justo, ac pellentesque risus. Donec eleifend, justo a pharetra laoreet, est dolor condimentum nulla, nec auctor ligula justo eget diam. Mauris augue eros, elementum quis vulputate eget, ullamcorper eget mauris. Maecenas quis hendrerit velit. Donec vehicula dictum tellus, et aliquam sem viverra maximus. Maecenas gravida purus quis dui vulputate ornare. Morbi ac orci ut nunc tristique ultricies nec mattis massa. In imperdiet nisl ut risus faucibus, ut semper libero egestas. Nam imperdiet ullamcorper nunc, eu dictum felis dapibus a. Aliquam nec ante posuere, tristique dui semper, hendrerit erat. Nunc purus massa, lobortis a laoreet vel, posuere sit amet nisl. Duis ut viverra nisl. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

Nulla elit risus, efficitur condimentum erat id, laoreet ullamcorper tellus. Nam non rutrum massa. Suspendisse vitae mauris vitae arcu elementum molestie et at diam. Proin efficitur vehicula ligula, id rhoncus dui pulvinar et. Nunc nec ultrices nunc, ut cursus libero. Donec mattis vehicula ex eget efficitur. Ut tellus arcu, vehicula nec ullamcorper ut, vulputate eu arcu. In cursus ut justo non sodales. Sed massa urna, eleifend eu nisi eget, tincidunt efficitur nibh. Donec at molestie arcu. Duis elementum lectus at tristique scelerisque. Donec sodales purus viverra urna interdum, sit amet convallis velit lacinia. Aliquam mollis tempus rhoncus. Quisque ultricies nisi quis metus rhoncus mattis. In fermentum facilisis tristique. Fusce ultricies quam id suscipit venenatis. Vestibulum hendrerit nibh eget ligula tristique, a tincidunt metus viverra. Aliquam finibus dui velit, a accumsan lacus tempus sit amet. Fusce in justo lorem. Mauris a porttitor justo, eget tincidunt erat. Ut quis semper risus. Aliquam eu malesuada metus. Aliquam erat volutpat. Sed ac diam finibus, pellentesque enim ut, commodo mi. Nam ligula odio, semper eget diam id, tempus cursus risus. Donec viverra, elit quis molestie ultrices, odio risus ornare orci, at eleifend nisl risus elementum tellus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Suspendisse eu aliquet est, scelerisque vulputate justo.

Suspendisse egestas posuere lacinia. Integer non mi maximus, rhoncus massa eu, aliquam risus. Nam nisl nisi, semper nec efficitur eu, tincidunt vel justo. Sed vestibulum tristique consequat. Phasellus sodales nunc quis pharetra mattis. Pellentesque eu fermentum sem, vel efficitur odio. Suspendisse consectetur turpis et nisi viverra commodo. Quisque ornare porta nisi, eget aliquam nisl interdum sed. Nulla elementum quam sit amet enim mollis, in rhoncus felis lacinia. Nam vel justo purus. Nulla pellentesque ex in eros rutrum tincidunt. Morbi consequat felis a libero dictum, vel malesuada est dignissim. Maecenas diam lacus, laoreet ut quam vel, pharetra aliquet nisi. Suspendisse auctor aliquam odio eu tincidunt. Aenean lacinia semper diam. Suspendisse potenti.

Sommario

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur malesuada suscipit nisl, vitae gravida odio imperdiet a. Etiam sed auctor tellus. Donec sed mauris eget nibh luctus accumsan. Sed imperdiet purus in elit iaculis, non commodo nisl varius. Integer sit amet diam laoreet, viverra lacus id, placerat nisl. Aenean ultricies sollicitudin elit in sodales. Vivamus euismod eleifend justo, ac pellentesque risus. Donec eleifend, justo a pharetra laoreet, est dolor condimentum nulla, nec auctor ligula justo eget diam. Mauris augue eros, elementum quis vulputate eget, ullamcorper eget mauris. Maecenas quis hendrerit velit. Donec vehicula dictum tellus, et aliquam sem viverra maximus. Maecenas gravida purus quis dui vulputate ornare. Morbi ac orci ut nunc tristique ultricies nec mattis massa. In imperdiet nisl ut risus faucibus, ut semper libero egestas. Nam imperdiet ullamcorper nunc, eu dictum felis dapibus a. Aliquam nec ante posuere, tristique dui semper, hendrerit erat. Nunc purus massa, lobortis a laoreet vel, posuere sit amet nisl. Duis ut viverra nisl. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

Nulla elit risus, efficitur condimentum erat id, laoreet ullamcorper tellus. Nam non rutrum massa. Suspendisse vitae mauris vitae arcu elementum molestie et at diam. Proin efficitur vehicula ligula, id rhoncus dui pulvinar et. Nunc nec ultrices nunc, ut cursus libero. Donec mattis vehicula ex eget efficitur. Ut tellus arcu, vehicula nec ullamcorper ut, vulputate eu arcu. In cursus ut justo non sodales. Sed massa urna, eleifend eu nisi eget, tincidunt efficitur nibh. Donec at molestie arcu. Duis elementum lectus at tristique scelerisque. Donec sodales purus viverra urna interdum, sit amet convallis velit lacinia. Aliquam mollis tempus rhoncus. Quisque ultricies nisi quis metus rhoncus mattis. In fermentum facilisis tristique. Fusce ultricies quam id suscipit venenatis. Vestibulum hendrerit nibh eget ligula tristique, a tincidunt metus viverra. Aliquam finibus dui velit, a accumsan lacus tempus sit amet. Fusce in justo lorem. Mauris a porttitor justo, eget tincidunt erat. Ut quis semper risus. Aliquam

eu malesuada metus. Aliquam erat volutpat. Sed ac diam finibus, pellentesque enim ut, commodo mi. Nam ligula odio, semper eget diam id, tempus cursus risus. Donec viverra, elit quis molestie ultrices, odio risus ornare orci, at eleifend nisl risus elementum tellus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Suspendisse eu aliquet est, scelerisque vulputate justo.

Suspendisse egestas posuere lacinia. Integer non mi maximus, rhoncus massa eu, aliquam risus. Nam nisl nisi, semper nec efficitur eu, tincidunt vel justo. Sed vestibulum tristique consequat. Phasellus sodales nunc quis pharetra mattis. Pellentesque eu fermentum sem, vel efficitur odio. Suspendisse consectetur turpis et nisi viverra commodo. Quisque ornare porta nisi, eget aliquam nisl interdum sed. Nulla elementum quam sit amet enim mollis, in rhoncus felis lacinia. Nam vel justo purus. Nulla pellentesque ex in eros rutrum tincidunt. Morbi consequat felis a libero dictum, vel malesuada est dignissim. Maecenas diam lacus, laoreet ut quam vel, pharetra aliquet nisi. Suspendisse auctor aliquam odio eu tincidunt. Aenean lacinia semper diam. Suspendisse potenti.

Contents

List of Figures

Chapter 1

Introduction

ECG-ira stands for ECG (electrocardiogram) instant rapid analyzer and it is an android application developed for the acquisition, visualization and analysis of the electrocardiogram signals. The application acquires and stores the record from the zecg acquisition device (using the zecg device format) property of the Politecnico of Milan but it can also open and visualize other ecg format from other devices. Ecg-ira is part of a greater and long term project (zecg itself was part of a first step). ECG-ira aims to exploit the reliability and performance of a complex software for electrocardiogram signal acquisition, visualization and processing within a smartphone device through a mobile application. Apart of the analysis algorithm already implemented during a past thesis (by Diego Ulisse Pizzagalli), the application was designed and implemented from scratch. During the process we dealt with the devices limitation in term of performance power and limited memory. We overcome many issues, related also to small and different screen size and density of pixel, due to the great number of different devices currently on the market. We had to exploit the multithreading and efficient memory usage strategies in order to achieve responsiveness and fulfill the medical requirement for an ECG compliant application. The result is an application which is easy to use because it follows all the best design principles according to the official Google guidelines for responsive UI and UX. By taking advantage of the multithreading capabilities and the usage of all the availables cores into a device, we achieved an application which performs fast and well.
This thesis is structured as follow:

- After this introduction we give a brief but detailed overview about the heart, its functionalities and how an electrocardiogram is related to it by explaining from what its an ecg to how heart electrical signals are detected and read.

- In the State of Art chapter we show the panorama of the actual devices for an ecg acquisition and some availables application on the market which allow to open and read an ecg record.

- Then it follows the Objective chapter in which we describe and explain the goal of this thesis work.

- The Requirements chapter is where we list a set of functional and nonfunctional set of features came up during the planning phase. The final result should be compliant with all of them.

- The Problem chapter deal with all the issues related to the project development. Here we discussed about the problem of choosing a development platform with respect to another, the hardware limitation on a smartphone device and the problems strictly related to the ecg signals.

- The Solution choice chapter is where we explain and provide our reasonings to the implementation choices, from the platform choice to the programming language choice to why we decide to avoid using drawings libraries preferring a custom and proprietary implementation.

- In the System architecture chapter we provide a general overview of the project structure for both the acquisition device implementation and architecture both the mobile application structure and main functionalities.

- In the Implementation details chapter we described all the main components and Java classes. Here we provide also hint for future implementations and adaptations.

- In the Final result chapter we propose some screenshots, with descriptive captions and description, of the main screens of the application; then we go through a performance analysis of application by observing how it affect the memory, the cpu and the response time of some portion of the code. The test were conducted on many different devices but only data from the older devices (2011) and the newest (November 2015) are compared.

- In the Conclusion chapter we sum up the overall results giving an overview of what has been done and providing some post consideration about the work done.
- In the Future works chapter we provide hints for further implementation in order to make ECG-ira really an all in one solution as an ECG mobile application. We also put our consideration about the future of the mobile software with respect to the desktop one, according to the new trend and the fact that the mobile environments is penetrating even more in our daily lives and can really positively change the way patients are connected with the health care providers.

Chapter 2

Electrocardiography overview

This chapter will introduce some basic but fundamental concepts about electrocardiography starting from the heart to the ECG and all issues related to the topic. We will start introducing the heart, its functionality and the entire circulatory system. After that we will describe in details the electrical activity inside the heart and how heart beats are generated. Following there will be a description of the electrocardiogram and the ECG signals. In the last section of this chapter we will discuss about all the noises and interference related to the ECG signal during its acquisition.

2.1 The heart

This chapter's focus is to describe in details the human heart. We will start from the structure to end up describing the heart functionality.

2.1.1 Human heart structure

The human heart is an organ that pumps blood throughout the body via the circulatory system, supplying oxygen and nutrients to the tissues and removing carbon dioxide and other wastes.
This fundamental organ has four chambers: two upper chambers(the atrial) and two lower ones(the ventricles). The right atrium and the right ventricle together make up the "right heart", and the left atrium and left ventricle make up the"left heart". The two sides of the heart are separated by a muscle called the septum.
A double-walled sac called the pericardium, encases the heart, which serves to protect

the heart and anchors it inside the chest. Between the outer layer, the parietal pericardium, and the inner layer, the serous pericardium, runs pericardial fluid, which lubricates the heart during contractions and movements of the lungs and diaphragm. The heart outer wall consists of three layers. The outermost wall layer, or epicardium, is the inner wall of the pericardium. The middle layer, or myocardium, contains the muscle that contracts. The inner layer, or endocardium, is the lining that contacts the blood.

The tricuspid valve and the mitral valve make up the atrioventricular (AV) valves, which connect the atria and the ventricles. The pulmonary semilunar valve separates the right ventricle from the pulmonary artery, and the aortic valve separates the left ventricle from the aorta. The heartstrings, or chordae tendineae, anchor the valves to heart muscles.

The sinoatrial node produces the electrical pulses that drive heart contractions.

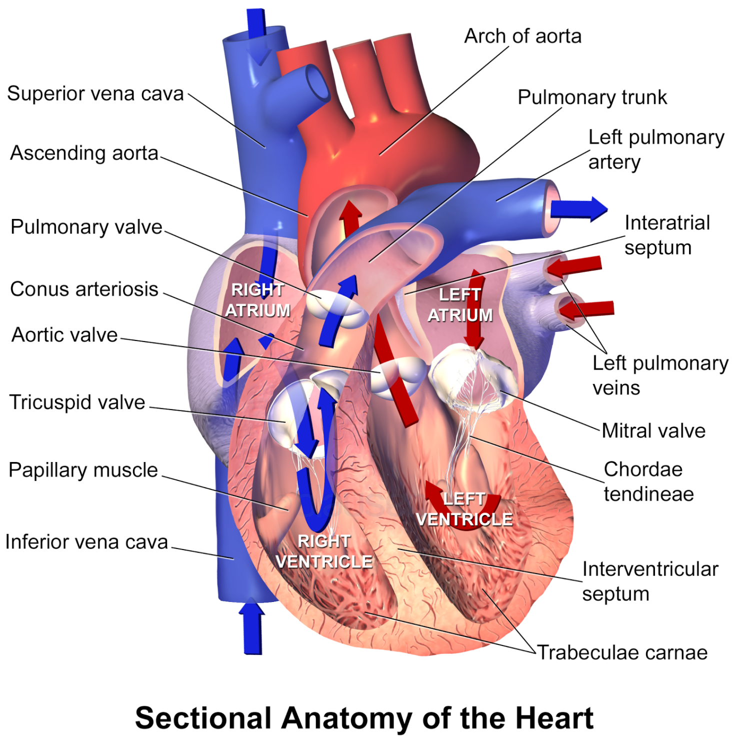


Figure 2.1: The human heart structure.[1]

2.1.2 Human heart function

The heart circulates blood through two pathways: the pulmonary circuit and the systemic circuit.
In the pulmonary circuit, deoxygenated blood leaves the right ventricle of the heart via the pulmonary artery and travels to the lungs, then returns as oxygenated blood to the left atrium of the heart via the pulmonary vein.
In the systemic circuit, oxygenated blood leaves the body via the left ventricle to the aorta, and from there enters the arteries and capillaries where it supplies the body's tissues with oxygen. Deoxygenated blood returns via veins to the venae cavae, re-entering the heart's right atrium.
Of course, the heart is also a muscle, so it needs a fresh supply of oxygen and nutrients too. After the blood leaves the heart through the aortic valve, two sets of arteries bring oxygenated blood to feed the heart muscle. The left main coronary artery, on one side of the aorta, branches into the left anterior descending artery and the left circumflex artery. The right coronary artery branches out on the right side of the aorta.
Blockage of any of these arteries can cause a heart attack, or damage to the muscle of the heart. A heart attack is distinct from cardiac arrest, which is a sudden loss of heart function that usually occurs as a result of electrical disturbances of the heart rhythm. A heart attack can lead to cardiac arrest, but the latter can also be caused by other problems.
The heart contains electrical "pacemaker" cells, which cause it to contract — producing a heartbeat.
Each cell has the ability to be the 'band leader' and have everyone follow. In people with an irregular heartbeat, or atrial fibrillation, every cell tries to be the band leader, which causes them to beat out of sync with one another.
A healthy heart contraction happens in five stages:

1. In the first stage (early diastole), the heart is relaxed.
2. Then the atrium contracts (atrial systole) to push blood into the ventricle.
3. Next, the ventricles start contracting without changing volume.
4. Then the ventricles continue contracting while empty.
5. Finally, the ventricles stop contracting and relax.

Then the cycle repeats.
Valves prevent backflow, keeping the blood flowing in one direction through the heart.

Some interesting data about the human heart are:

- A human heart is roughly the size of a large fist.
- The heart weighs between about 280 to 340 grams in men and 230 to 280 grams in women.
- The heart beats about 100,000 times per day (about 3 billion beats in a lifetime).
- An adult heart beats about 60 to 80 times per minute.
- Newborns' hearts beat faster than adult hearts, about 70 to 190 beats per minute.
- The heart pumps about 6 quarts (5.7 liters) of blood throughout the body.
- The heart is located in the center of the chest, usually pointing slightly left

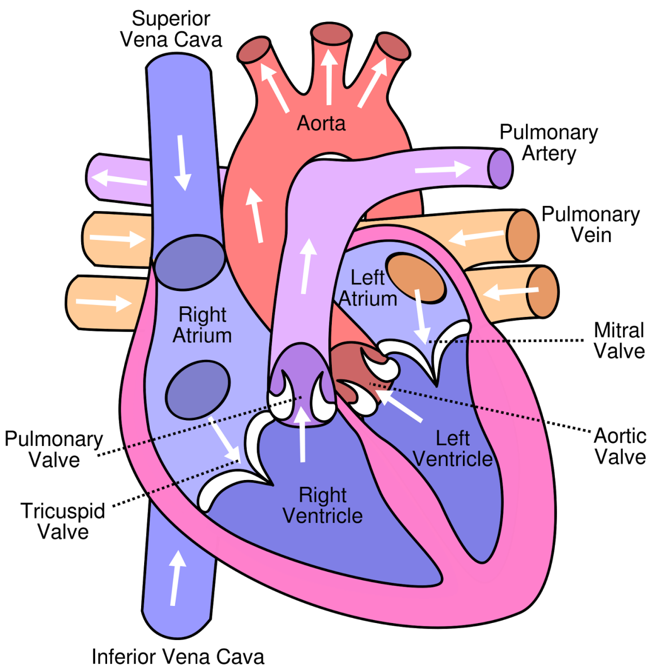


Figure 2.2: The circulatory system with blood flow.

2.2 Heart electrical activity

The heart has a natural pacemaker that regulates the pace or rate of the heart. It sits in the upper portion of the right atrium (RA) and is a collection of specializes electrical cells known as the SINUS or SINOATRIAL (SA) node.
Like the spark-plug of an automobile it generates a number of "sparks" per minute. Each "spark" travels across a specialized electrical pathway and stimulates the muscle wall of the four chambers of the heart to contract (and thus empty) in a certain sequence or pattern. The upper chambers or atria are first stimulated. This is followed by a slight delay to allow the two atria to empty. Finally, the two ventricles are electrically stimulated. In an car, the number of sparks per minute generated by a spark plug is increased when you press the gas pedal or accelerator. This revs up the motor. In case of the heart, adrenaline acts as a gas pedal and causes the sinus node to increase the number of sparks per minute, which in turn increases the heart rate. The release of adrenaline is controlled by the nervous system. The heart normally beats at around 72 times per minute and the sinus node speeds up during exertion, emotional stress, fever, etc., or whenever our body needs an extra boost of blood supply. In contrast, it slows down during rest or under the influence of certain medications. Well trained athletes also tend to have a slower heart beat.

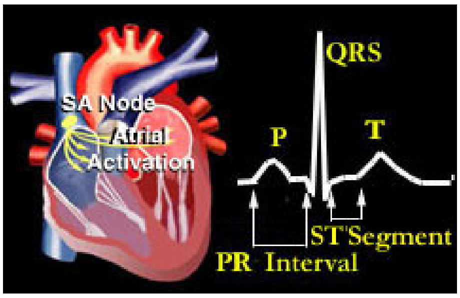


Figure 2.3: The SA node fires and electrical impulses travels through the right and left atrium.

The sequence of electrical activity within the heart is displayed in the diagrams above and occurs as follows:

1. As the SA node fires, each electrical impulse travels through the right and left atrium. This electrical activity causes the two upper chambers of the heart to

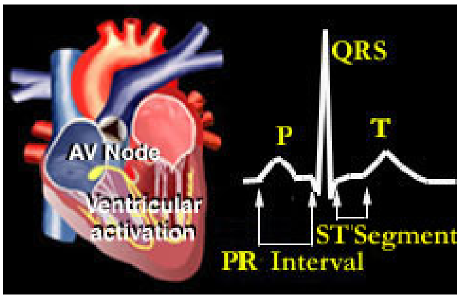


Figure 2.4: The impulse then move to the ventricular area.

contract. This electrical activity and can be recorded from the surface of the body as a "P" wave" on the patient's EKG or ECG (electrocardiogram).

2. The electrical impulse then moves to an area known as the AV (atrio-ventricular) node. This node sits just above the ventricles. Here, the electrical impulse is held up for a brief period. This delay allows the right and left atrium to continue emptying its blood contents into the two ventricles. This delay is recorded as a "PR interval." The AV node thus acts as a "relay station" delaying stimulation of the ventricles long enough to allow the two atria to finish emptying.

3. Following the delay, the electrical impulse travels through both ventricles (via special electrical pathways known as the right and left bundle branches). The electrically stimulated ventricles contract and blood is pumped into the pulmonary artery and aorta. This electrical activity is recorded from the surface of the body as a "QRS complex". The ventricles then recover from this electrical stimulation and generates an "ST segment" and T wave on the ECG.

2.3 Electrocardiogram

An electrocardiogram(abbreviated as ECG or EKG) is a test that measures the electrical activity of the heartbeat. With each beat, an electrical impulse (or wave) travels through the heart. This wave causes the muscle to squeeze and pump blood from the heart. A normal heartbeat on ECG will show the timing of the top and lower chambers.

The right and left atria or upper chambers make the first wave called a "P wave" following a flat line when the electrical impulse goes to the bottom chambers. The right and left bottom chambers or ventricles make the next wave called a "QRS complex." The final wave or "T wave" represents electrical recovery or return to a resting state for the ventricles.

Each waves in the figure 2.5 is no other than the result of different views or

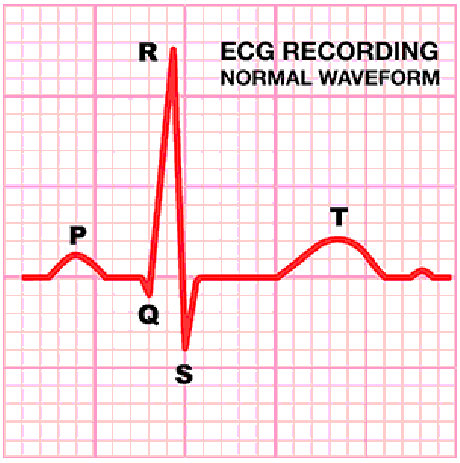


Figure 2.5: An example of a normal ECG waveform.

perspectives of the waveforms generated from the current in the heart.

There are two type of ECGs recordings: the 12-lead ECG and the rhythm strip. Both give valuable information about heart function.

We will focus our attention on the 12-lead ECG. It records information from 12 different views of the heart and provides a complete picture of electrical activity. The limb leads and the chest, or precordial, leads reflect information from the different planes of the heart. Different leads provide different information. The six limb leads I, II, III, augmented vector right (aVR), augmented vector left (aVL), and augmented vector foot (aVF) provide information about the heart's frontal (vertical) plane. Leads I, II, and III require a negative and positive electrode for monitoring, which makes those leads bipolar. The augmented leads record information from one lead and are called unipolar.

The six precordials or V leads V1, V2, V3, V4, V5, and V6 provide information about the heart's horizontal plane. Like the augmented leads, the precordial leads are also unipolar, requiring only a single electrode. The opposing pole of those leads is the center of the heart as calculated by the ECG.

The position of the leads are crucial for a right ECG recordings. It is common to use the so called Einthoven's triangle, a set of positions to set up standard limb leads. The electrodes for leads I, II, III are about equidistant from the heart and form an

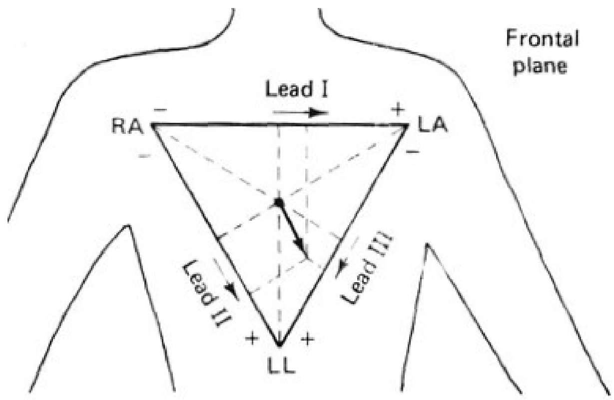


Figure 2.6: The Einthoven's triangle show the right position to place leads over the chest.

equilateral triangle.

2.3.1 Lead I

It provides a view of the heart that shows current moving from right to left. Because the current flows from negative to positive, the positive electrode for this lead is placed on the left arm or on the left side of the chest; the negative electrode is placed on the right arm. Lead I produces a positive deflection on ECG tracings and is helpful in monitoring atrial and hemiblock.

2.3.2 Lead II

Lead II produces a positive deflection. Place the positive electrode on the patient's left leg and the negative electrode on the right arm. For continuous monitoring, place the electrodes on the torso for convenience, with the positive electrode below the lowest palpable rib at the left midclavicular line and the negative electrode below the right clavicle. The current travels down and to the left in this lead. Lead II tends to produce a positive, high voltage deflection, resulting in tall P, R, and T waves. This lead is commonly used for routine monitoring and is useful for detecting sinus node and atrial arrhythmias.

2.3.3 Lead III

Lead III produces a positive deflection. The positive electrode is placed on the left leg; the negative electrode, on the left arm. Along with lead II, this lead is useful for detecting changes associated with an inferior wall myocardial infarction. The axes of the three bipolar limb leads I, II, and III form a triangle around the heart and provide a frontal plane view of the heart.

2.3.4 Augmented leads

Leads aVR, aVL, and aVF are called augmented leads. They measure electrical activity between one limb and a single electrode. Lead aVR provides no specific view of the heart. Lead aVL shows electrical activity coming from the heart's lateral wall. Lead aVF shows electrical activity coming from the heart's inferior wall.

2.3.5 Precordials leads

The six unipolar precordial leads (V1, V2, V3, V4, V5 and V6) are placed in sequence across the chest and provide a view of the heart's horizontal plane.

- Lead V1—The precordial lead V1 electrode is placed on the right side of the sternum at the fourth intercostal rib space. This lead corresponds to the modified chest lead MCL1 and shows the P wave, QRS complex, and ST segment particularly well. It helps to distinguish between right and left ventricular ectopic beats that result from myocardial irritation or other cardiac stimulation outside the normal conduction system. Lead V1 is also useful in monitoring ventricular arrhythmias, ST-segment changes, and bundle-branch blocks.

- Lead V2—Lead V2 is placed at the left of the sternum at the fourth intercostal rib space.

- Lead V3—Lead V3 goes between V2 and V4. Leads V1, V2, and V3 are biphasic, with both positive and negative deflections. Leads V2 and V3 can be used to detect ST-segment elevation.

- Lead V4—Lead V4 is placed at the fifth intercostal space at the midclavicular line and produces a biphasic waveform.

- Lead V5—Lead V5 is placed at the fifth intercostal space at the anterior axillary line. It produces a positive deflection on the ECG and, along with V4, can show changes in the ST segment or T wave.
- Lead V6—Lead V6, the last of the precordial leads, is placed level with V4 at the midaxillary line. This lead produces a positive deflection on the ECG.

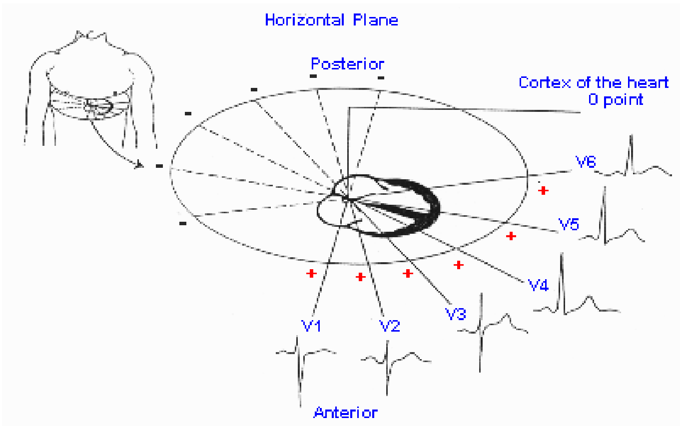


Figure 2.7: Precordial leads and their position related to the heart and the chest horizontal plane.

2.3.6 How to read a ECG record

Waveforms produced by the heart's electrical current are recorded on graphed ECG paper by a stylus. An ECG paper consists of horizontal and vertical lines forming a grid. A piece of ECG paper is called an ECG strip or tracing. The horizontal axis of

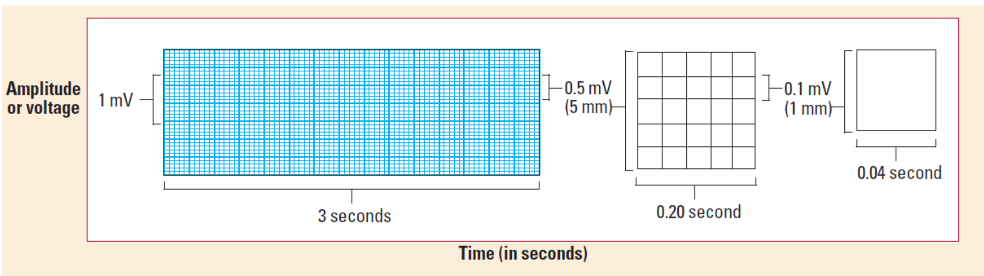


Figure 2.8: A typical ECG paper.

the ECG strip represents time. Each small block equals 0.04 second, and five small blocks form a large block, which equals 0.2 second. This time increment is determined by multiplying 0.04 second (for one small block) by 5, the number of small blocks that compose a large block. Five large blocks equal 1 second (5 x 0.2). When measuring or calculating a patient's heart rate, a 6-second strip consisting of 30 large blocks is usually used. The ECG strip's vertical axis measures amplitude in millimeters (mm) or electrical voltage in millivolts (mV). Each small block represents 1 mm or 0.1 mV; each large block, 5 mm or 0.5 mV. To determine the amplitude of a wave, segment, or interval, count the number of small blocks from the baseline to the highest or lowest point of the wave, segment, or interval.

2.4 Noises and interferences

Obtaining a reliable ECG recording is still an issue. In fact there may occur many problems interfering with the signals. Some of these problems include artifacts from patient movement and poorly placed or poorly functioning equipment.

2.4.1 Artifact

Artifact , also called waveform interference, may be seen with excessive movement (somatic tremor). The baseline of the ECG appears wavy, bumpy, or tremulous. Dry electrodes may also cause this problem to occur due to poor contact.

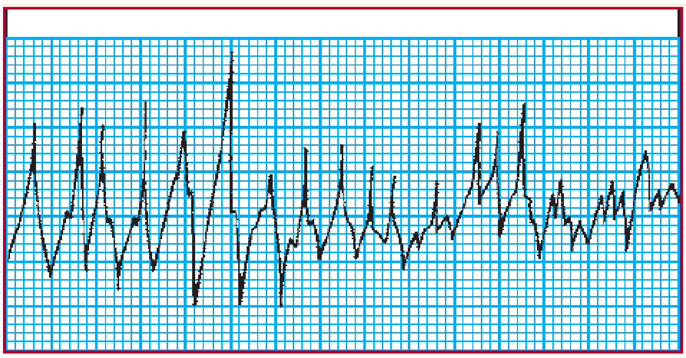


Figure 2.9: ECG waveform interference due to artifact may cause monitoring to fail due to unreadable signals.

2.4.2 Interference

Electrical interference, also called 60-cycle interference, is caused by electrical power leakage. It may occur due to interference from other room equipment or improperly grounded equipment. As a result, the lost current pulses at a rate of 60 cycles per second. This interference appears on the ECGs as a baseline that is thick and unreadable.

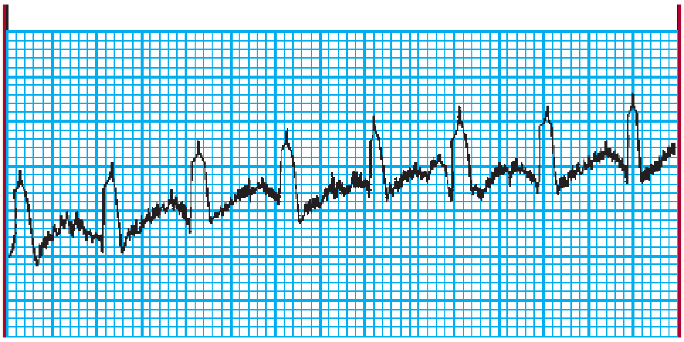


Figure 2.10: Electrical interference causes the baseline to be unstable and the signal is corrupted.

2.4.3 Wandering baseline

A wandering baseline undulates, meaning that all waveforms are present but the baseline is not stationary. It can be caused by movement if the chest wall during respiration, poor electrode placement, or poor electrode contact.

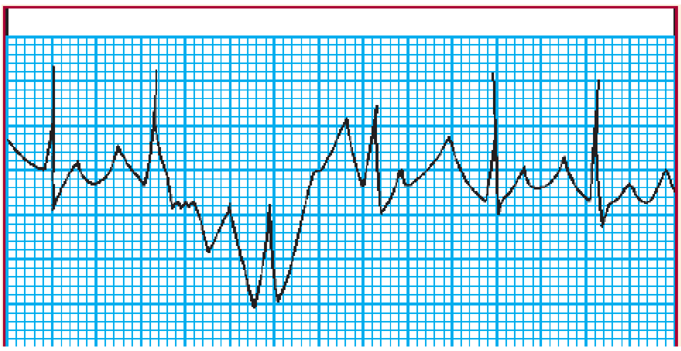


Figure 2.11: An example of baseline wandering due to artifacts.

2.4.4 Faulty equipment

Faulty equipment, such a s broken lead wires and cables, can also cause monitoring problems.

Chapter 3

State of Art

3.1 Device

The personal health care market has changed a lots and recently new products and devices are showing up on the market. We will describe briefly the most relevant and similar products as mobile ECG acquisition devices. We evaluate the following solutions:

- Mortara ELI 10 Mobile
- Philips DigiTrak XT Holter Recorder
- M-Trace (PC) Mobile
- ECG Expert

3.1.1 Mortara ELI 10 Mobile

This device offers an all in one solution for 12 leads ECG acquisition. It is compact and complete as it provides an alphanumeric keyboard and a screen for real time visualization and the possibility to send the record via GPRS/3G channels. For each devices a SIM card is required . The device can also read and interpret the ECG supporting the doctor. Interesting feature is its great interoperability with the main ECG data management systems.

Figure 3.1: Mortara ELI 10 Mobile, ECG acquisition device box.

3.1.2 Philips DigiTrak XT Holter Recorder

This is the smaller acquisition device on the market. Thanks to a proprietary algorithm from Philips it can derive all the 12 ECG leads using only 5 leads. It weighs 62g and the internal battery lasts till 7 days. It also has a small screen showing 1 real time signal at a time.

Figure 3.2: DigiTrack, the ECG visualization.

3.1.3 AliveCor ECG

An innovative solution even though it doesn't offer a complete solution for ECG acquisition and analysis. This small sensor can be attached on the back of your smartphone making it an ECG acquisition device. It can record only one ECG signal (D1), so also the analysis is limited to a few types of arrhythmias . The record length is also limited to 5 minutes.

Figure 3.3: AliveCor device real time acquisition on a tablet.

3.1.4 M-Trace (PC)Mobile

M-Trace PC is an completed 12 leads ECG acquisition device. With the device it comes a mobile application and a desktop pc application used to visualize and analyze the ecg signals. The device is really portable with dimensions 95x64x28mm. The company offers also a more portable device (M-Trace Mobile) to be used by privates at their home. The mobile version cannot acquire a full record but only test records with 6 leads. Its main purpose it to send the test records via GSM/GPRS to the doctor for a faster review.

Figure 3.4: M-Trace PC device for ECG acquisition.

3.1.5 ECG Expert

ECG Expert produced by CSE Medical is a completed solution for ECG acquisition. The device comes with fully supported software for both PC desktop (Windows and Mac) both smartphones (Android, iOS). The device is rechargeable and makes use of a wireless connection via Bluetooth as exchanges data communication with the handheld smartphone or PC software.

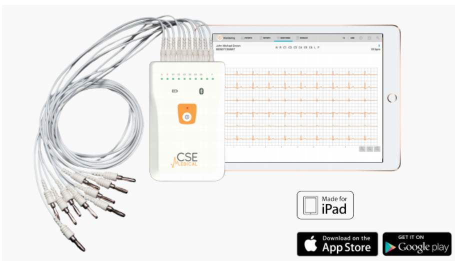


Figure 3.5: M-Trace PC device for ECG acquisition.

3.2 Mobile application

There are already mobile applications on the market store for ECG visualization and analysis supporting different formats. We can distinguish applications that only visualize the signal and the ones that also apply some analysis on the ECG signals. We listed only applications on the Google Play Store, so only Android applications because they are the only comparable with the solution we propose.

3.2.1 Visualization only application

The application on the market able to visualize the ECG signal are:

- *StribogECG*: an Android application based on an open source project under GLP v3 licence. It uses Biosig library to read ECG formats such as scp, xml (hl7), ecg and dgf. The software is only provided as it is and it requires to the user to already have the ecg files stored in those supported formats.

- *AndroidECG*: application on beta release, it was developed by Paco Gonzàles as thesis project during the Master course in Computer Science at the University of Murcia. The application is able to show ECG signals of the following formats: binary, scp, 212. As additional feature it is a basic analysis over the signal to detect QRS complex, P waves, ST segments and T waves. It is also possible to send the ECG record via email.

3.2.2 Visualization and Analysis

The applications on the market that also provide a more detailed analysis over the ECG signal are all bind to a specific proprietary acquisition device. By this way they lack the compatibility and interoperability requirement with other software and ECG formats.

- *M-Trace PC*: the application was developed by *M4Medical*, a Poland company providing medical devices for professionals and private customers. The application only works with the company 12-channel ECG M-Trace PC register device. The main features are the real-time monitoring interface, a patients' database management system and the possibility to share the record.

- *ECG Expert*: developed by *CSE Medical* the application works only connected to an ECG-Expert acquisition device. The main features are the real-time view of the acquisition, the analysis of the record providing information about QRS complex and heart rate, the possibility to manage patient information bind to the record and a heartbeat Normal/Abnormal classifier.

One last mobile application, which is not strictly related to ECG signal visualization and analysis but it worth to be mentioned, is the *ECG Interpretation*. This application instead provides enough detailed information about how to read and interpret the

ECG signals through 32 small lectures. All the lectures provide a picture and a short description and explanation.

Chapter 4

Objective

4.1 Preface

For a clear understanding of the next chapters we will make use of some terms listed below with the proper meanings:

1. Mobile application: it is a software running on smartphones and tablets
2. Desktop application: it is a software running on desktop pcs or notebooks
3. Acquisition device: named ZEcg, it is the device (hardware) used to acquire the ECG signal from the electrodes connected to a patient body.

4.2 Fully functional medical mobile app as replacement to desktop app

The main purpose for this thesis is to develop a medical mobile application as replacement to an original desktop application. The application needs to be standalone and independent from other softwares, still it can share its content and integrate other software content.
As a starting point we planned to reproduce all the desktop features such as the connection between the application and the remote device ZEcg for the ECG signal acquisition. It should also save the ECG records inside the mobile device, plot the signals and run arrhythmia recognition algorithms on them. We are aware that the user experience is different from a desktop one due to the differences in capabilities

and functionalities. Having in mind these differences, we didn't try to reproduce the desktop experience. We developed instead the application having a mobile experience at first position, following the standards of mobile application designs and principles. We took advantage of the new and latest technologies mounted on the new smartphones, trying to provide to the end user the best in term of user experience, performance and application design. The main difficulty is probably to redesign and re-imagine the desktop feature from a mobile point of view. For example, if a desktop application usually makes use of keyboard and mouse, inside a modern mobile application there is only the touch input as user interaction. The differences in term of screen size, memory and also cpu performance matters and should always be kept in mind during the initial planning phase. We will deal with these and others limitations, trying to achieve the best results and performances.
We believe this application can be really a replacement to a desktop application as the technology trends point, to future devices with better performances in term of lower power consumption and higher operational capabilities.[3]

Chapter 5

Requirement

In the project there was the need for a deep analysis of all the tied requirements. The result of this analysis was essential to identify the subsequent problems.
We will describe all of them, distinguishing between functional and nonfunctional requirements.

5.1 Functional

5.1.1 Connection management with the acquisition device

Fundamental feature to be included inside the mobile application is the capability to directly connect the smartphone device to the acquisition device ZEcg. Since this last one was designed to transmit the signals through a bluetooth channel, we have to implement and manage a bluetooth socket connection inside the application in order to receive the data.

5.1.2 Acquisition, storing and management of ECG records

For a matter of medical feature as it is a fact that there are many “standards” on saving an ECG signal, the application has to be able to manage different formats. Even though this application is designed to be used mostly for acquisition from the ZEcg device, it is also able to open and read other standard format such as the MIT-BIH, one of the most common standard in the literature of ECG. The code software behind is designed in a such way that the integration of other format is made extremely easy to add just by implementing few interfaces and classes.

5.1.3 Different ECG formats support

For a matter of medical feature as it is a fact that there are many "standards" on saving an ECG signal, the application has to be able to manage different formats. Even though this application is designed to be used mostly for acquisition from the ZEcg device, it is also able to open and read other standard format such as the MIT-BIH, one of the most common standard in the literature of ECG. The code software behind is designed in a such way that the integration of other format is made extremely easy to add just by implementing few interfaces and classes.

5.1.4 Dynamic display scaling

The mobile device market is huge and there are a very large number of devices with completely different hardware and screens. As first classification we can distinguish mobile devices into smartphones and tablets. The most obvious difference is based on the screen size and the pixel density. Building a mobile application means also to deal with these number of different devices. To achieve the same experience and look and feel the application should be able to scale its view according to the device screen and the pixel density. A typical ECG signal is plot on a paper with squares of well defined size in millimeters. The mobile application has to respect such a standard independently on the screens capabilities and pixel density, so it should be able to properly scale the view and the plotting based on the hardware provided by the device.

5.1.5 ECG record analysis integration on mobile platform

To complete the set of features for the application we plan to integrate the algorithms of ECG signal processing. To have a mobile device able not only to acquire and visualize in real time the ECG but also to analyze it at runtime, can be of vital importance, especially if the user has little knowledge about reading and interpreting an ECG graph. The integrated algorithms for arrhythmia analysis are based on a Neural Network trained to recognized the nature of the signals for the given record with high accuracy. The algorithms come from a previous thesis work[4] which belongs to Ulisse Pizzagalli, student at Politecnico of Milan.

5.1.6 Analysis results displaying

After analysis there are results that need to be shown to the user in the most friendly and understandable way. The most important results from an ECG analysis are called Istogram, Tacogram and the ST+/ST- . They are respectively graphs showing the number of heart rates of a certain value, the average heart rate at each heart beat and the difference between the area ST+ and ST-, the area above the segment ST and the one below. This last graph is useful for ischemia detection.[5]

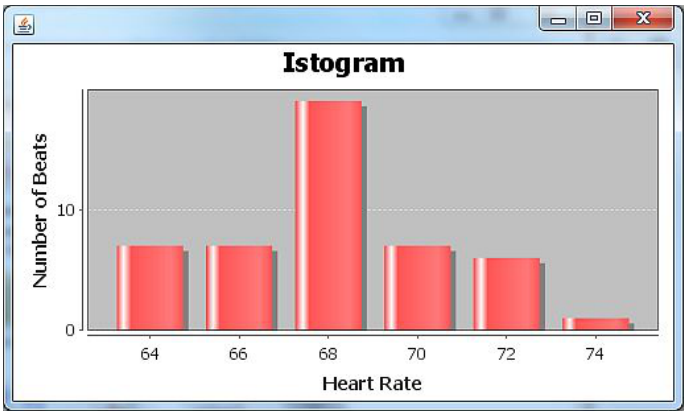


Figure 5.1: Istogram from the desktop application resulted from an analysis on a MIT/BIH record.

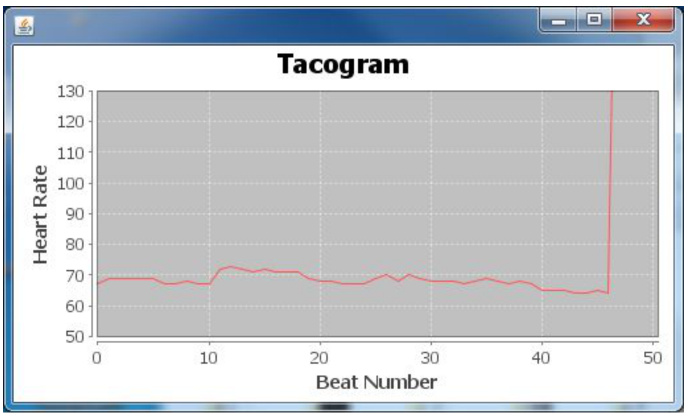


Figure 5.2: Tacogram from the desktop application resulted from an analysis on a MIT/BIH record.

Figure 5.3: ST+/ST- graph from the desktop application resulted from an analysis on a MIT/BIH record.

5.1.7 Highly parameterizable

We believe in dynamic software, that is why we planned from the beginning on making this application dynamic. Even if the application is build on top of the ZEcg standard, we plan to make the software responsive also to other standards. To achieve this, we planned to abstract all the acquisition device independent features and functionalities. In order to support as much as possible any variants of the original acquisition device, we plan to setup customizable parameters, the only related to the hardware implementation. With a little of changes the application will be able to interface with other devices as well for acquisition.

5.2 Nonfunctional

5.2.1 Reduced memory usage

This requirement is fundamental for any project related to mobile application development. In fact, if a desktop pc in general doesn't have any problem related to memory usage (even if it is a good practice not to waste memory), on mobile devices this over-usage can bring the application to crash and get killed by the OS. The memory available is higher on new devices with respect to older ones, but it is still small so it is always a good practice to use it carefully.

5.2.2 Minimum performance rate and scalability on performance

Nowadays the new high level mobile devices has quad-core or even octa-core cpu processors. Any application should take advantage of a such configuration, but on the other hand mobile application developers should always consider the fact that the market is still full of older and low-end devices. In order to cover at least most of the market devices their application have to run fine (with a minimum acceptable performance rate) starting from the low-end devices and, at the same time, taking advantage of last devices capabilities.
We believe modern applications should seriously take this aspect in consideration, because it will make their application scalable also from a performance point of view.

5.2.3 Wide platform compatibility and accessibility

Developing a mobile application implies building a software that has to be executed on many different platforms. The smartphone and tablet market is huge with many different devices mounting different hardwares and running of the three major mobile OS (iOS, Android and Windows Phone). In the next chapters we will deal with this issue.

5.2.4 Documentation

This thesis includes also a more technical documentation about the development phase and the choices we starting from the planning phase to the development phase. The software is fully documented and with annotations and comments to increase code readability and future development on top of it. The technical documentation is included in the next chapters where we are going to discuss and motivate the implementation and the results.

Chapter 6

Problems

By identifying the requirements, we could then be able to highlight the related problems that we had to face in order to fulfill all of them. We will describe the related requirement as source of each faced problem.

6.1 Mobile platform fragmentation

Considering the requirement of a wide platform compatibility, we obviously need to face a really big problem in the mobile application world: the platform fragmentation. Starting from the first smartphone release on 2007, the iPhone from Apple, the sale of such devices keeps increasing each year. Between 2007 and 2008 sales proceeded upwards reaching the same sale rate of their computing parent, the PC. On 2009 the market signed an important inflection point, representing the beginning of an inexorable vertical rise. Although PCs were still the only ones to offer some types of functionality due to their longer replacement cycle, they were sold with a ratio of $1:2$, compared to smartphones, over a 5 year period. This new market' growth leads to an obvious seeking of various participants, some by choice and some by necessity, in order to extract value. Android has been the prime beneficiary, having been announced on 2007 and having gone on to account for well over half of smartphone sales worldwide. Apple, meanwhile, has maintained a steady ship, leveraging its skill in product design and user experience with a finely honed marketing machine, attracting a customer that rarely defects.[6] Fast-forwarding to 2016, if we consider the global market share held by the leading smartphone operating systems, at first position we have Android with a share of 80.7%, followed by iOS with a 17.7%. Minor percentages are represented

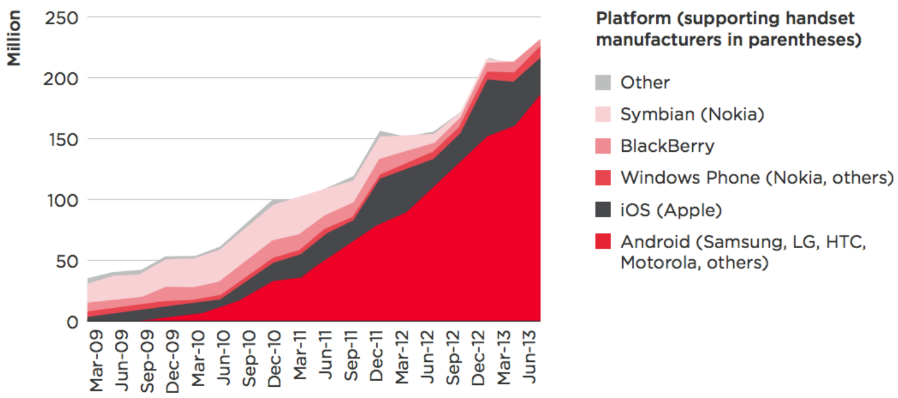


Figure 6.1: Mobile platform share evolution (smartphone sales), 2009–13.

by Windows Phone (1.1%), RIM (0.2%) and others (0.2%).[7]

6.2 Native vs Cross-Platform

As mentioned earlier, one of the main challenges when moving to a mobile solution is the software development within a technology landscape that is highly fragmented and rapidly evolving. Mobile apps require a fair amount of customization to run on diverse platforms and a constant update due to the steady stream of new hardware, OS versions and browsers. Even a single platform (Android, Windows, Blackberry, and to a lesser extent Apple) has numerous flavors that require some degree of customization. There are also other factors such as the overlay software from different manufacturers that can affect behavior of an app on a particular device.

In response, the mobile industry has spawned a rapidly growing ecosystem of cross-platform and cross-device frameworks, source code analyzers, libraries of reusable components, and other tools designed to accelerate and simplify multi-platform development. New tools are constantly emerging, with new functionality, different capabilities, strengths and weaknesses.

Developer's preferences change and evolve, particularly as new tools and capabilities become available. However, the basic goals are the same: to code less and accomplish more, to reuse and recycle across multiple platforms as much as possible, and consider developing from scratch as the last resort. In addition, any tool or framework should be able to work with current and future evolving offers, and not to be locked because of a particular platform or technology.[8]

6.2.1 Native

Native app development involves developing software specifically designed for a specific platform, hardware, processor and its set of instructions. Typical programming language are Java, Object C, Swift, C# and many other.
Native apps' major advantage over cross-platform apps is the ability to leverage device-specific hardware and software. This means that native apps can take advantage of the latest technology and API available on the mobile devices and it can well interface with other platform apps. Other advantages are a predictable increase on performance, streamlined support, native UI, native API and coherent library updates. However, the mobile platform's fragmentation makes the task of keeping up with the pace of emerging technology onerous and costly, having to develop different software for each of the different platforms (Android, iOS, Windows Phone, Symbian).
Going further on a more technical analysis, native applications are represented by executable binary files that will be installed into devices without the need for other abstraction layering to the operating system. They are able to call built in functionalities such as calendar, notifications, the dialer, email provider and others services and functionalities provided natively by the OS. Despite the fact that native applications development requires platform specific skill and expertise, this strategy delivers a higher quality user experience than other mobile application development methods (cross-platform or hybrid approach). Native apps are also best distributed through an app store.[9]

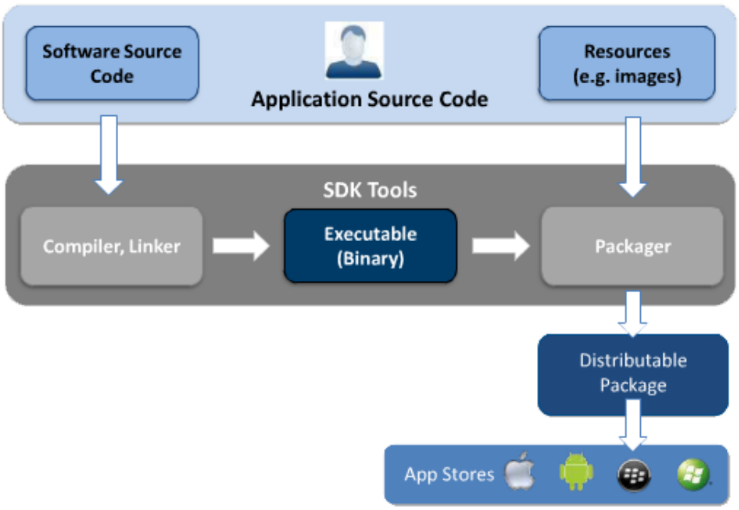


Figure 6.2: Native app development process.

6.2.2 Cross-Platform

Cross platform development produces one code base to maintain and write targeting multiple devices and platforms. It promises lower time of development and costs. The main categories of this group are web, hybrid, interpreted and generated apps. None of the previous is neither prevalent nor the best solution to the problem of developing cross-platform mobile applications. The main benefits of cross-platform development

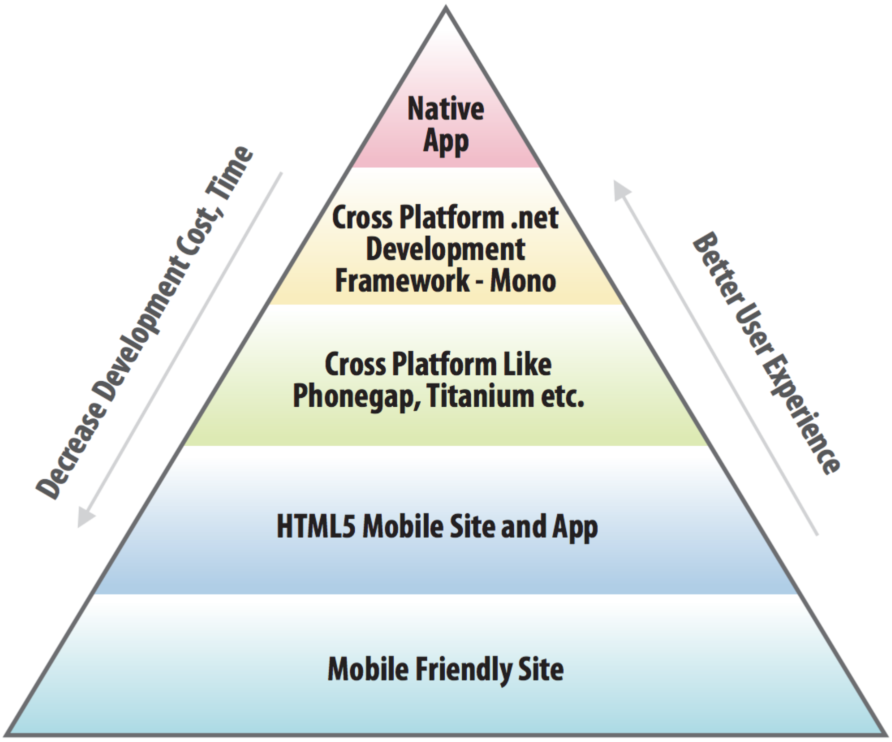


Figure 6.3: Native vs Cross-Platform cost and time factors.[2]

are:

- Reduction of required skills to develop applications due to the use of common programming languages;
- Reduction of coding work, because the source code is written once and it is compiled for each supported OS;
- Reduction of development time and long term maintenance costs;
- Decrement of API knowledge, because with these tools is not needed to know the API's of each OS, but only the API's provided by the selected tool.[10]

6.3 Mobile hardware limitations

As this project started from a previously developed desktop application, we needed to take into account all the limitation of mobile devices hardware. Despite the trend of mobile hardware is a constant increasing in computational power and memory availability, we cannot compare it with the one available nowadays in PC architectures, as the architectures on this two platforms are totally different: most of all pc are based on x86 architectures, whereas mobiles rely on ARM architectures. Furthermore, we could not commit the error of targeting only recent devices as they represent only a small percentage of all devices in the market, and it would have been in contrast to our main goal, which is a wide device compatibility. We will discuss briefly about memory and computational differences in these type of architecture in the next paragraphs.

6.3.1 Memory limitations

The first point to consider is the device memory limitation. In the previous desktop application not so much effort was spent to reduce memory allocation: all the ECG record samples were allocated in memory both during acquisition and record opening. This approach could result convenient considering its low implementation effort, but it could be source of lots of problem when moving to a mobile device, where not only the physical memory are often small, but also the memory reserved to an application is limited (figure 6.4), both due to the coexistent apps running in the system and the power management of the mobile OSs. So we needed to find a good approach in order to manage these problems and not to face with memory leakages.

6.3.2 Performance limitations

Talking about cpu computational power there is a huge gap between the two architectures.. Not considering the evident difference for what regarding the space availability for the CPU package, the main difference is that ARM architectures are focused on reducing the power consumption, a crucial characteristic in mobile devices. A technical report analyzed the execution time of a benchmark using a cross-platform version of WebKit showed that the tested ARM A9 architecture had a ratio of 5.8 (normalized time) compared to the tested x86 architecture.[11]

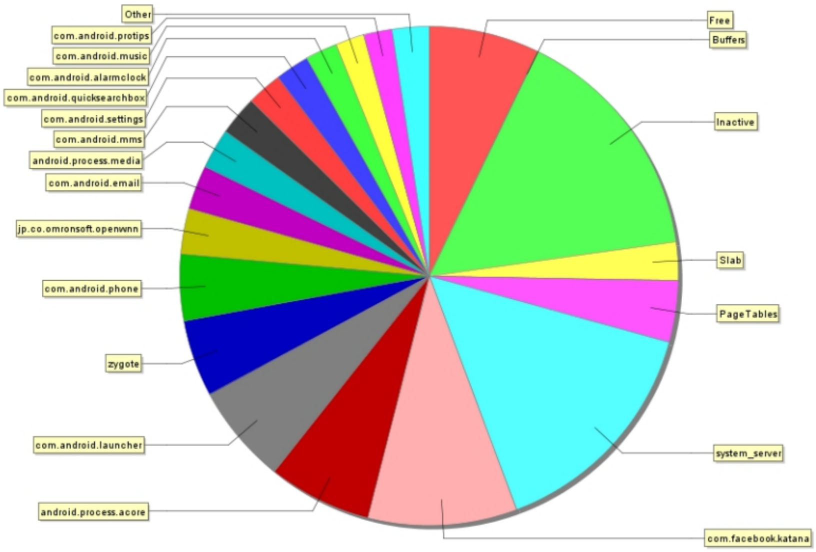


Figure 6.4: Memory reserved to an app in Android.

6.4 ECG baseline wander

For a right interpretation and identification of physiological and pathological phenomena, a low error is required in the ECG signal. But often ECG recordings are distorted by two main artifacts:

- high-frequency noise caused by electromyogram induced noise, power line interferences, or mechanical forces acting on the electrodes;
- baseline wander (BW) that may be due to respiration or the motion of the patients or an instrument fault (Figure 6.5).

These artifacts strongly limit the utility, readability and interpretation of a recorded ECGs. In ECG enhancement, the goal is to separate the valid ECG from the undesired artifacts so that we can extract a signal that allows an easy visual interpretation.[12] The first class of artifact is mostly solved within the acquisition device at hardware low-level though filters and a proper settings. In this project we will deal with the second class of artifact which can be managed at software level.

6.4.1 State of Art

Two methods, in recent years have often been applied to remove BW:

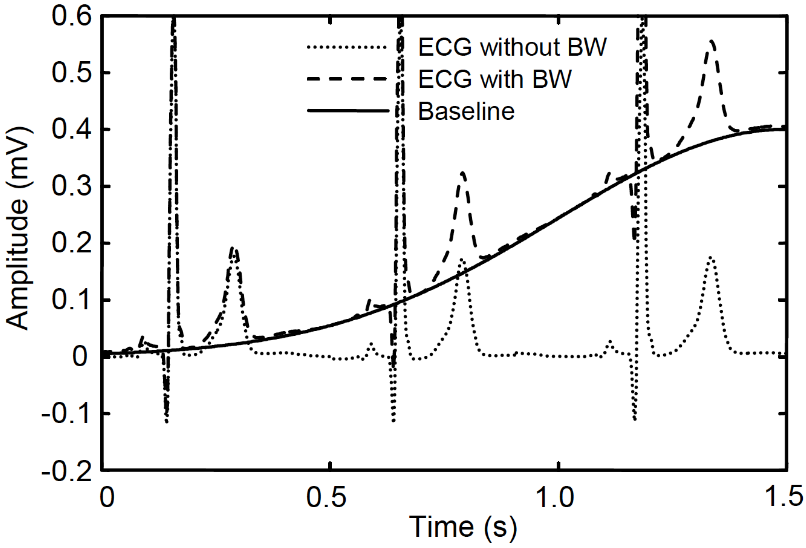


Figure 6.5: The baseline wander artifact in the ECG signal.

- Polynomial fitting: in order to assess the baseline, it uses a polynomial interpolation. The baseline is fit from some fiducial points that are determined from P-R intervals, whereas these fiducial points are difficult to accurately locate before noise is removed from the ECG signal. As result this approach is ineffective if the ECG signal is contaminated by noise.[13]

- High-pass filtering: to implement this type of filtering, usually a moving average filter[14] and wavelet translation[15]. This approach however would unavoidably introduce distortions in various parts of the ECG signal, especially in the ST segment due to the spectra of the ST segment that overlaps the spectra of BW.

Simple Moving Average Filter

This type of filtering is defined as

$$y(n) = \frac{1}{2N+1} \sum_{i=-N}^{N} x(n+i) \tag{6.1}$$

where $x(n)$ and $y(n)$ are input signal and output signal of the moving average respectively, and N specifies the observation window length equal to $2N+1$. So the baseline can be estimated as

$$z(n) = x(n) - y(n) \tag{6.2}$$

where $z(n)$ is the output signal of the high-pass filter.

Distortion using Simple Moving Average Filtering

Obviously the baseline values estimated from the P-R segment (between 0.3 s and 0.6 s) are very close to real baseline values, while the baseline values estimated from segments including QRS complex and T wave are far away from the real baseline. Therefore, if an observation window covers some sample points with extreme amplitudes, an ECG signal would be distorted.

CPU intensive operation

Many methods are known in the literature that perform better and reduce the error in the ECG signal due to the filtering, like wavelet package translation filter[16], or the statistical weighted moving average filter[17]. Defined the normalized root mean square error ($NRMSE$) and maximum error (ME) as

$$NRMSE = \sqrt{\frac{\sum_{i=1}^{L}(ecg_{in}(i) - ecg_{out}(i))^2}{\sum_{i=1}^{L} ecg_{in}(i)}} \tag{6.3}$$

$$ME = \max_{i=1,2,\ldots,L} |ecg_{in}(i) - ecg_{out}(i)| \tag{6.4}$$

The figure 6.6 shows a comparison between the methods. Of course the computational complexity these advanced methods is really high. Considering that this type of filtering is supposed to be used during a real-time acquisition in our project, they are obviously prohibitive.

6.5 Signal visualization

The problem of an efficient ECG signal visualization shows up when the application has to plot samples of an acquisition device with a rate of 500Hz. Because of this the app has to be able to draw $500 - 1$ lines (connecting two samples) for each mapped second in the ECG paper grid. Taking into account the standard size of a second

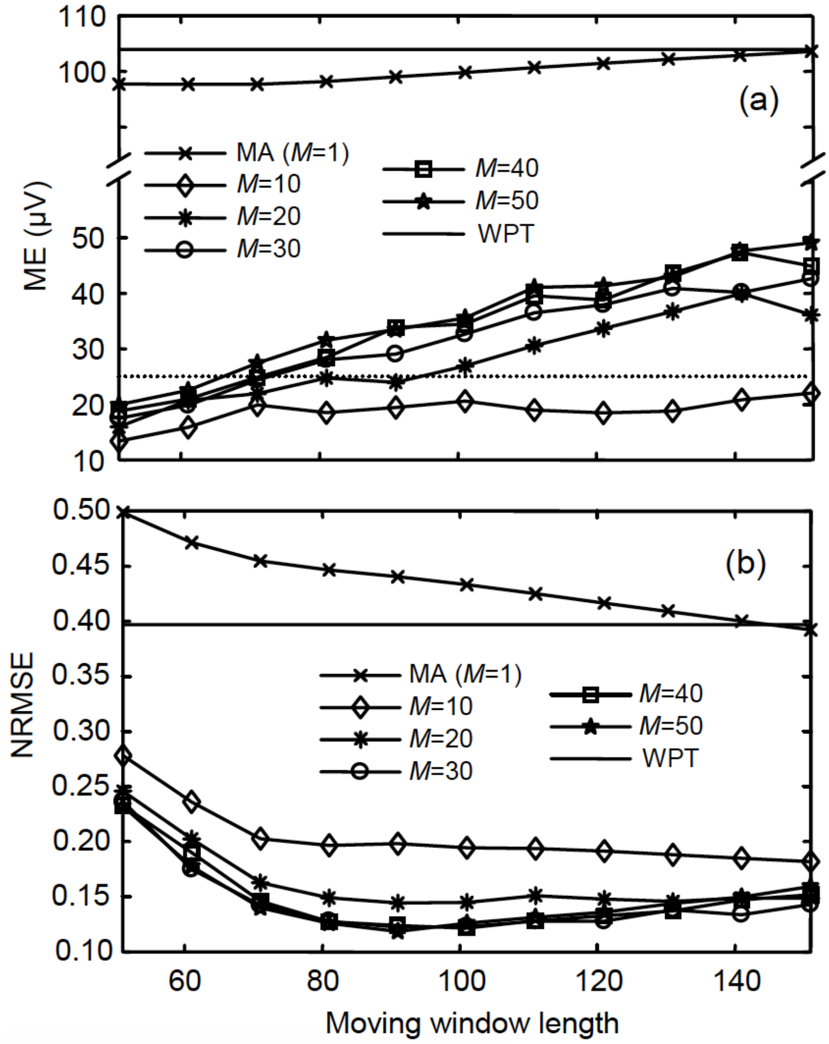


Figure 6.6: Relation between distortion and moving window length for wavelet package translation (WPT), traditionally used MA, and our proposed high-pass filter based on a statistical weighted moving average (SMA).
(a) Maximum error (ME) vs. moving window length;
(b) Normalized root mean square error (NRMSE) vs. moving window length. M is the number of sub-bounds. SMA is the same as MA when M=1.

in the standard ECG paper of 5mm, and the fact that the visualization size has to respect this measure, if we consider a device screen of 4.6 inch in portrait, we have around 11 seconds of samples to visualize. Hence we have $(500 - 1) * 11 = 5489$ lines to plot. If we are supposed to plot ideally all the ECG derivation (12), the number of total lines become $5489 * 12 = 65868$. Now, this is the number of lines we need to plot in order to cover the total visible window screen space. If we want to

have a good display smoothness we must guarantee at least 30FPS (frame rate per seconds), ideally 60 frame/s (FPS). Assuming a minimum of 30 FPS we have a total of $65868 * 30 = 1976040$.
These numbers gives us an idea of the computational effort that is spent only only to plot the samples.

Chapter 7

Solution choices

After a long research period of time and direct experience on developing mobile application using the most known cross-platform (Xamarin) and hybrid (Cordova Phonegap) solution, we decided to go native. This decision was based mostly on the needs and the strict performance requirements related to the project. Thanks to a native implementation we can achieve better results in term of performance and in term of user experience of the application. At the choice of a native platform we picked Android because it is the most spread mobile OS over mobile devices and it is open source even if mostly maintained by Google.

7.1 Android platform

Android is a mobile operating system (OS) currently developed by Google, based on the Linux Kernel and designed primarily for touchscreen mobile devices such as smartphones and tablets. Android has the largest installed base of all operating systems of any kind. Android has been the best selling OS on tablets since 2013, and on smartphones it is dominant by any metric.[18] We have chosen to develop ECG-ira firstly on this platform because of it is widely spread over the world and its nature of being from an open source software made it stable and widely supported. The Android OS programming language is Java which is one of the most known and used OOP language for application and web development.

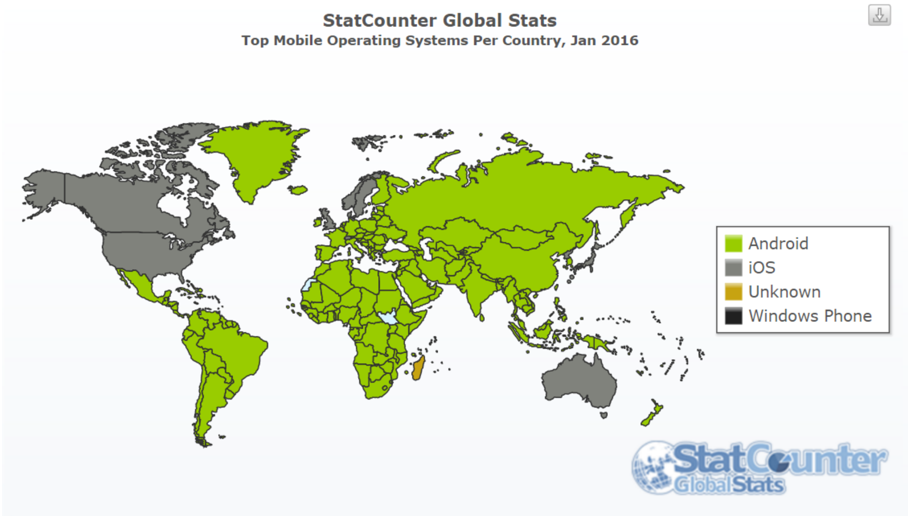


Figure 7.1: Top Mobile Operating Systems Per Country, Jan 2016. Statcounter.com

7.2 Why native?

The advantages of cross-platform (and hybrid) solution is mostly connected to code maintenance and faster development, because most of the written codes stands for all the supported OS at building phase. Business logic and data structure can be easily share among the many OS for example by using Xamarin we can write unique code using C# (c-sharp) and abstract the business logic, the web services and the database management independently from the specific platform we want to target. Most of the time it is just a matter of working on different user interface, one for any supported OS. All of this look awesome, and it is, but when it comes to performance metrics, custom interfaces and user experience, here we meet its weaknesses and limitations. ECG-ira main goal is to build up an usable and stable mobile medical application and to fulfill it we needed to exploit the native platform in order to achieve the best performance and the best user experience. As Android is the most spread mobile OS over smartphones and tablets it results in an obvious pick.

We believe native may not be the best pick for any kind of application. The choices has to be done according to the project requirements and goals. Pitfall for going native is the long development time and a deep (if not full) knowledge of that specific platform.

7.3 Android concurrency exploitation

As we have seen previously, a considerable computational effort is required to the app, especially during record acquisition. For this reason, the only way in order to guarantee a reasonable performance was to exploit the concurrency mechanisms available in the chosen platform. This decision will obviously increase the complexity of the execution: analyzing the execution of a single-threaded application is relatively simple because the order of execution is known. In multi-threaded applications, it is a lot more difficult to analyze how the program is executed and in which order the code is processed.

In the following paragraphs we will start from the basic mechanisms provided by Java language, and we will then analyse the ones, given available by the Android OS, that we have chosen to use in our application.

7.3.1 Thread Overview

Software programming is all about instructing the hardware to perform an action. The instructions are defined by the application code that the CPU processes in an ordered sequence, which is the high-level definition of a thread. From an application perspective, a thread is execution along a code path of Java statements that are performed sequentially. A code path that is sequentially executed on a thread is referred to as a task, a unit of work that coherently executes on one thread. A thread can either execute one or multiple tasks in sequence.

Thread execution

A thread in Java machine is represented by java.lang.Thread. It is the most basic execution environment in Android that executes tasks when it starts and terminates when the task is finished or there are no more tasks to execute; the alive time of the thread is determined by the length of the task. Thread supports execution of tasks that are implementations of the java.lang.Runnable interface. An implementation defines the task in the run method:

```
private class MyTask implements Runnable {
   public void run() {
      int i = 0; // Stored on the thread local stack.
   }
```

```
}
```

All the local variables in the method calls from within a run() method—direct or indirect—will be stored on the local memory stack of the thread. The task's execution is started by instantiating and starting a Thread:

```
Thread myThread = new Thread(new MyTask());
myThread.start();
```

On the operating system level, the thread has both an instruction and a stack pointer. The instruction pointer references the next instruction to be processed, and the stack pointer references a private memory area—not available to other threads—where thread-local data is stored. Thread local data is typically variable literals that are defined in the Java methods of the application.
A CPU can process instructions from one thread at a time, but a system normally has multiple threads that require processing at the same time, such as a system with multiple simultaneously running applications. For the user to perceive that applications can run in parallel, the CPU has to share its processing time between the application threads. The sharing of a CPU's processing time is handled by a scheduler. That determines what thread the CPU should process and for how long. The scheduling strategy can be implemented in various ways, but it is mainly based on the thread priority: a high-priority thread gets the CPU allocation before a low-priority thread and receive more execution time with respect to low-priority threads. The execution of two concurrent threads can be done in java just declaring two Thread objects and then starting them by calling the method Thread .start():

```
Thread T1 = new Thread(new MyTask());
T1.start();
```

7.3.2 Threads in Android

In Android basically there are three thread types:

- **UI thread** (or main thread): it is started on application start and stays alive during the lifetime of the application process. The UI thread is the main thread of the application, used for executing Android components and updating the UI elements on the screen. If the platform detects that UI updates are attempted from any other thread, it will promptly notify the application by throwing a CalledFromWrongThreadException. This harsh platform behavior is required because the Android UI Toolkit is not thread safe, so the runtime allows access to the UI elements from one thread only.

- **Binder threads**: they are used for communicating between threads in different processes. Each process maintains a set of threads, called a thread pool, that is never terminated or recreated, but can run tasks at the request of another thread in the process. These threads handle incoming requests from other processes, including system services, intents, content providers, and services.

- **Background threads**: All the threads that an application explicitly creates are background threads. This means that they have no predefined purpose, but are empty execution environments waiting to execute any task. The background threads are descendants of the UI thread, so they inherit the UI thread properties, such as its priority. By default, a newly created process doesn't contain any background threads. It is always up to the application itself to create them when needed.

The UI thread is the most important thread, but it gets no special scheduling advantage compared to the other threads—the scheduler is unaware of which thread is the UI thread. Instead, it is up to the application to not let the background threads interfere more than necessary with the UI thread.

7.3.3 Thread communication in Android

In multithreaded applications, tasks can run in parallel and collaborate to produce a result. Hence, threads have to be able to communicate to enable true asynchronous processing.
The most common thread communication use case in Android is between the UI thread and worker threads. Hence, the Android platform defines its own message passing mechanism for communication between threads. The UI thread can offload long tasks by sending data messages to be processed on background threads. The message passing mechanism is a nonblocking consumer-producer pattern, where neither the producer thread nor the consumer thread will block during the message handoff.
The message handling mechanism in android is implemented with the following classes:

- **android.os.Looper**: A message dispatcher associated with the one and only consumer thread.
- **android.os.Handler**: Consumer thread message processor, and the interface for a producer thread to insert messages into the queue. A Looper can have many associated handlers, but they all insert messages into the same queue.
- **android.os.MessageQueue**: Unbounded linked list of messages to be processed on the consumer thread. Every Looper—and Thread—has at most one MessageQueue.
- **android.os.Message**: Message to be executed on the consumer thread.

The mechanism is summarized in the figure 7.2.

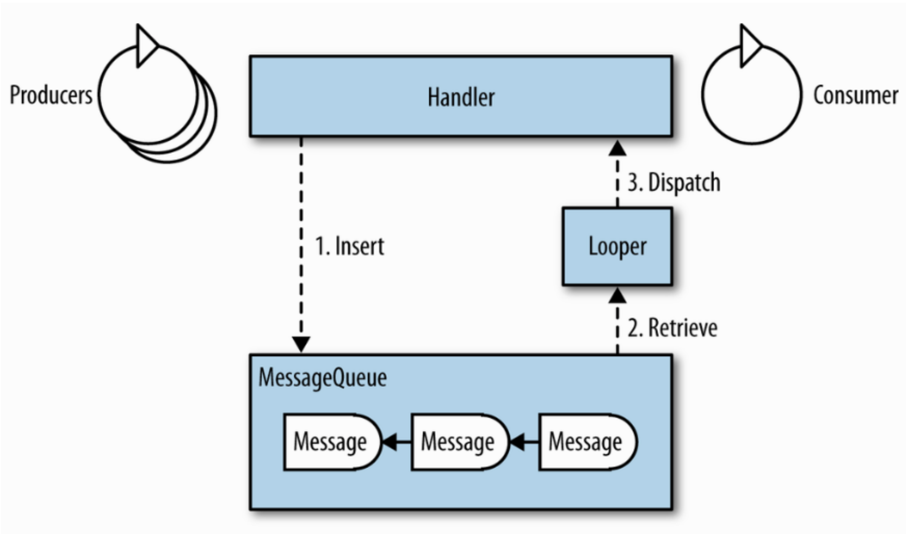


Figure 7.2: Overview of the message-passing mechanism between multiple producer threads and one consumer thread.

- **Insert**: The producer thread inserts messages in the queue by using the Handler connected to the consumer thread.
- **Retrieve**: The Looper, runs in the consumer thread and retrieves messages from the queue in a sequential order.
- **Dispatch**: The handlers are responsible for processing the messages on the consumer thread. A thread may have multiple Handler instances for processing

messages; the Looper ensures that messages are dispatched to the correct Handler.

7.3.4 HandlerThread

Now we will describe a component that we have heavily exploited in our application, and as you will see in implementation details section, it represents the base of two main operations: the ECG signal drawing during both acquisition and record opening, and the ECG signal reading during record opening.
HandlerThread is a thread with a message queue that incorporates a Thread, a Looper, and a MessageQueue. It is constructed and started in the same way as a Thread. Once it is started, HandlerThread sets up queuing through a Looper and MessageQueue and then waits for incoming messages to process:

```
HandlerThread handlerThread = new HandlerThread("HandlerThread");
handlerThread.start();

mHandler = new Handler(handlerThread.getLooper()) {
   @Override
   public void handleMessage(Message msg) {
      super.handleMessage(msg);
      // Process messages here
   }
};
```

There is only one queue to store messages, so execution is guaranteed to be sequential, and therefore thread safe. The HandlerThread sets up the Looper internally and prepares the thread for receiving messages.
Here is an simple example of an implementation:

```
public class MyHandlerThread extends HandlerThread {
   private Handler mHandler;
   public MyHandlerThread() {
      super("MyHandlerThread", Process.THREAD_PRIORITY_BACKGROUND);
   }
   @Override
   protected void onLooperPrepared() {
```

```
        super.onLooperPrepared();
        mHandler = new Handler(getLooper()) {
            @Override
            public void handleMessage(Message msg) {
                switch(msg.what) {
                    case 1:
                        // Handle message
                        break;
                    case 2:
                        // Handle message
                        break;
                }
            }
        };
    }
    public void publishedMethod1() {
        mHandler.sendEmptyMessage(1);
    }
    public void publishedMethod2() {
        mHandler.sendEmptyMessage(2);
    }
}
```

Lifecycle

A running HandlerThread instance processes messages that it receives until it is terminated. A terminated HandlerThread can not be reused. To process more messages after termination, create a new instance of HandlerThread. The lifecycle can be described in a set of states:

- **Creation**: The constructor for HandlerThread takes a mandatory name argument and an optional priority for the thread:

  ```
  HandlerThread(String name)
  HandlerThread(String name, int priority)
  ```

  The name argument simplifies debugging, because the thread can be found more easily in both thread analysis and logging. The priority argument is optional and should be set with the same Linux thread priority values used in Process.setThreadPriority. The default priority is Process.THREAD_PRIORITY_DEFAULT, the same priority as the UI thread, and can be lowered to Process.THREAD_PRIORITY_BACK to execute non critical tasks.

- **Execution**: The HandlerThread is active while it can process messages; i.e., as long as the Looper can dispatch messages to the thread. The dispatch mechanism is set up when the thread is started through HandlerThread.start and it is ready when either HandlerThread.getLooper returns or on the onLooperPrepared callback. A HandlerThread is always ready to receive messages when the Handler can be created, as getLooper blocks until the Looper is prepared.

- **Reset**: The message queue can be reset so that no more of the queued messages will be processed, but the thread remains alive and can process new messages. The reset will remove all pending messages in the queue, but not affect a message that has been dispatched and is executing on the thread:

```java
public void resetHandlerThread() {
    mHandler.removeCallbacksAndMessages(null);
}
```

  The argument to removeCallbacksAndMessages removes the message with that specific identifier. null, shown here, removes all the messages in the queue.

- **Termination**: A HandlerThread is terminated either with quit or quitSafely, which corresponds to the termination of the Looper. With quit, no further messages will be dispatched to the HandlerThread, whereas quitSafely ensures that messages that have passed the dispatch barrier are processed before the thread is terminated. You can also send an interrupt to the HandlerThread to cancel the currently executing message:

```java
public void stopHandlerThread(HandlerThread handlerThread) {
    handlerThread.quit();
    handlerThread.interrupt();
}
```

A terminated HandlerThread instance has reached its final state and cannot be restarted.

7.3.5 Thread Pools

A thread pool is the combination of a task queue and a set of worker threads that forms a producer-consumer setup. Producers add tasks to the queue and worker threads consume them whenever there is an idle thread ready to perform a new background execution. So, the worker thread pool can contain both active threads executing tasks, and idle threads waiting for tasks to execute. There are several advantages with thread pools over executing every task on a new thread (thread-per-task pattern):

- The worker threads can be kept alive to wait for new tasks to execute. This means that threads don't have to be created and destroyed for every task, which compromises performance.
- The thread pool is defined with a maximum number of threads so that the platform isn't overloaded with background threads—that consume application memory—due to many background tasks.
- The lifecycle of all worker threads are controlled by the thread-pool lifecycle.

ThreadPoolExecutor

A thread pool's behavior is based on a set of properties concerning the threads and the task queue, which you can set to control the pool. The properties are used by the ThreadPoolExecutor to define thread creation and termination as well as the queuing of tasks. The configuration is done in the constructor,

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(
int corePoolSize,
int maximumPoolSize,
long keepAliveTime,
TimeUnit unit,
BlockingQueue<Runnable> workQueue);
```

where:

- **corePoolSize**: The lower limit of threads that are contained in the thread pool. Actually, the thread pool starts with zero threads, but once the core pool size is reached, the number of threads does not fall below this lower limit. If a task is added to the queue when the number of worker threads in the pool is lower than the core pool size, a new thread will be created even if there are idle threads waiting for tasks. Once the number of worker threads is equal to or higher than the core pool size, new worker threads are only created if the queue is full.

- **maximumPoolSize**: The maximum number of threads that can be executed concurrently. Tasks that are added to the queue when the maximum pool size is reached will wait in the queue until there is an idle thread available to process the task.

- **keepAliveTime**: Idle threads are kept alive in the thread pool to be prepared for incoming tasks to process, but if the alive time is set, the system can reclaim noncore pool threads. The alive time is configured in TimeUnits, the unit the time is measured in.

- **workQueue**: An implementation of BlockingQueue that holds tasks added by the consumer until they can be processed by a worker thread. Depending on the requirements, the queuing policy can vary.

ScheduledThreadPoolExecutor

This is an extension of the ThreadPoolExecutor, that can schedule commands to run after a given delay, or to execute periodically. This class will be really useful in our application because of its capability of scheduling task at a fixed rate through the method:

```
scheduleAtFixedRate(Runnable command, long initialDelay,
    long period, TimeUnit unit)
```

where:

- **command**: the task to execute
- **initialDelay**: the time to delay first execution
- **period**: the period between successive executions

- **unit**: the time unit of the initialDelay and period parameters

7.3.6 AsyncTask

As the name indicates, an AsyncTask is an asynchronous task that is executed on a background thread. The only method you need to override in the class is doInBackground(). Hence, a minimal implementation of an AsyncTask looks like this:

```
public class MinimalTask extends AsyncTask {
   @Override
   protected Object doInBackground(Object... objects) {
      // Implement task to execute on background thread.
   }
}
```

The task is executed by calling the execute method, which triggers a callback to doInBackground on a background thread:

```
new MinimalTask().execute(Object... objects);
```

When an AsyncTask finishes executing, it cannot be executed again—i.e., execute is a one-shot operation and can be called only once per AsyncTask instance, the same behavior as a Thread.
In addition to background execution, AsyncTask offers a data passing mechanism from execute to doInBackground. Objects of any type can be passed from the initiating thread to the background thread. This is like HandlerThread, but with AsyncTask you do not have to be concerned about sending and processing Message instances with a Handler.
In the common case where you want to execute a task in the background and deliver a result back to the UI thread, AsyncTask shines; it is all about handling the flow of preparing the UI before executing a long task, executing the task, reporting progress of the task, and finally returning the result. All of this is available as optional callbacks to subclasses of the AsyncTask, which look like this:

```
public class FullTask extends AsyncTask<Params, Progress, Result> {
   @Override
   protected void onPreExecute() { ... }
   @Override
```

```
    protected Result doInBackground(Params... params) { ... }
    @Override
    protected void onProgressUpdate(Progress... progress) { ... }
    @Override
    protected void onPostExecute(Result result) { ... }
    @Override
    protected void onCancelled(Result result) { ... }
}
```

This implementation extends the AsyncTask and defines the arguments of the objects that are passed between threads:

- **Params**: Input data to the task executed in the background.
- **Progress**: Progress data reported from the background thread (i.e. from doInBackground) to the UI thread in onProgressUpdate.
- **Result**: The result produced from the background thread and sent to the UI thread.

All callback methods are executed sequentially, except onProgressUpdate, which is initiated by and runs concurrently with doInBackground. Figure 7.3 shows the lifecycle of an AsyncTask and its callback sequence. The different steps are:

1. Create the AsyncTask instance.
2. Start execution of the task.
3. First callback on the UI thread: onPreExecute. This usually prepares the UI for the long operation—e.g., by displaying a progress indicator on the screen.
4. Callback on a background thread: doInBackground. This executes the long-running task.
5. Report progress updates from the publishProgress method on the background thread. These trigger the onProgressUpdate callback on the UI thread, which typically handles the update by changing a progress indicator on the screen. The progress is defined by the Progress parameter.

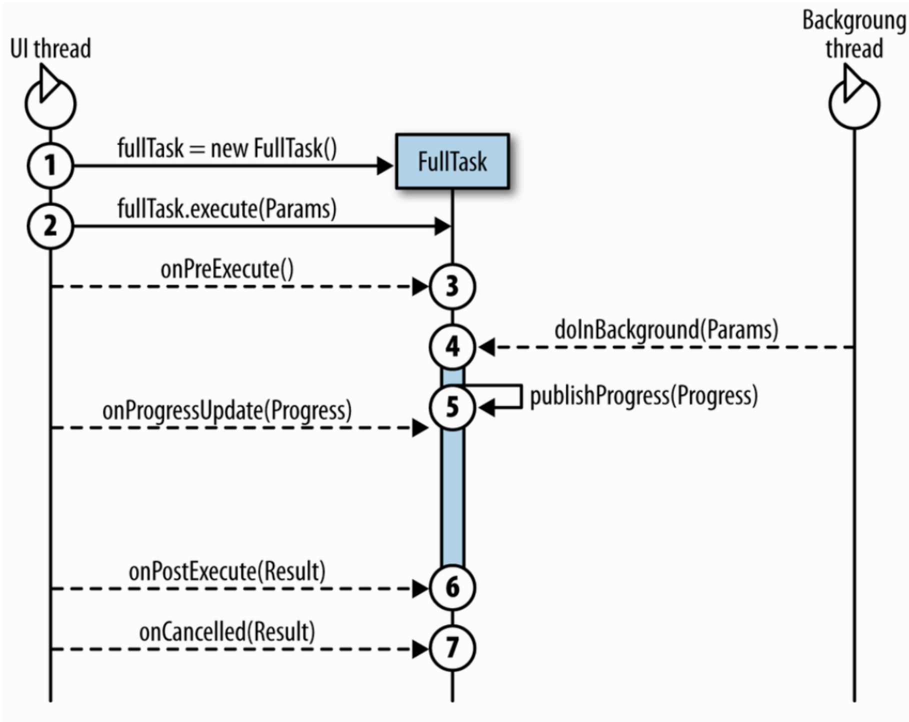


Figure 7.3: The execution lifecycle of AsyncTask.

6. The background execution is done and is followed by running a callback on the UI thread to report the result. There are two possible callbacks: onPostExecute is called by default, but if the AsyncTask has been cancelled, the callback onCancelled gets the result instead. It is guaranteed that only one of the callbacks can occur.

The progress update mechanism solves two use cases:

- Displaying to the user how the long-running operation is progressing, by continuously reporting how many of the total tasks are executed.

- Delivering the result in portions, instead of delivering everything at the end in onPostExecute. For example, if the task downloads multiple images over the network, the AsyncTask does not have to wait and deliver all images to the UI thread when they are all downloaded; it can utilize publishProgress to send one image at the time to the UI thread. In that way, the user gets a continuous update of the UI.[19]

7.4 Baseline wander solutions

In order to solve this problem, taking into account all the related problematic, we have chosen two kinds of approach. Each one will be described afterwards. The first approach will be always active, and will consist in the dynamic calculation of samples vertical axis during drawing iteration. The second is the usage of a simple moving average filter, that could be activated in the app settings for the acquisition phase.

7.4.1 Adaptive vertical displaying

We decided to hold this type of solution active by default in order to maintain a solid and versatile way to overcome the worst scenario caused by a strong baseline wander. The approach works like this: at each frame drawing, we have a visible window of samples, with a length dictated from the space availability of the device screen, that we have to plot. Given that window, we know that we have to fit as many samples as we can, inside the space dedicated to that signal, the signal ECG strip. In a normal scenario, the signal will be aligned to a baseline, and so we can easy plot all the samples inside the relative strip. But in some other scenarios, it could happen that because of movement of the patient, the signal could immediately drop down. For this reason, the signal could easily go out of the available vertical space. To avoid this, we have to provide a mechanism in order to hit this cases and accordingly respond. We do this by not fixing the vertical baseline of the ECG strip, and leave it dynamic. So this baseline will go up and down according to the position of the values inside the samples window. So every time we have to redraw the updated window on the screen, we before compute the baseline of the signal at that moment by computing the mean of all samples. After that, we can shift the signal to plot up or down, trying to include the majority of samples on the screen. An example of the mechanism is showed in the figure 7.4. As you can see, starting from the second 23, a patient movement caused the baseline wander artifact, causing a drop of the D1 signal. The app by applying the dynamic displaying, it computed the new baseline of the signal, represented by the mean of all samples, and shifted all sample window accordingly. In the figure, the variation of the vertical axis is represented by Δy.
In this way we can overcome this type of scenario, avoiding the appliance of any kind of filtering. The latter, depending on the technique, can result in some percentage of error on the filtered ECG signal.

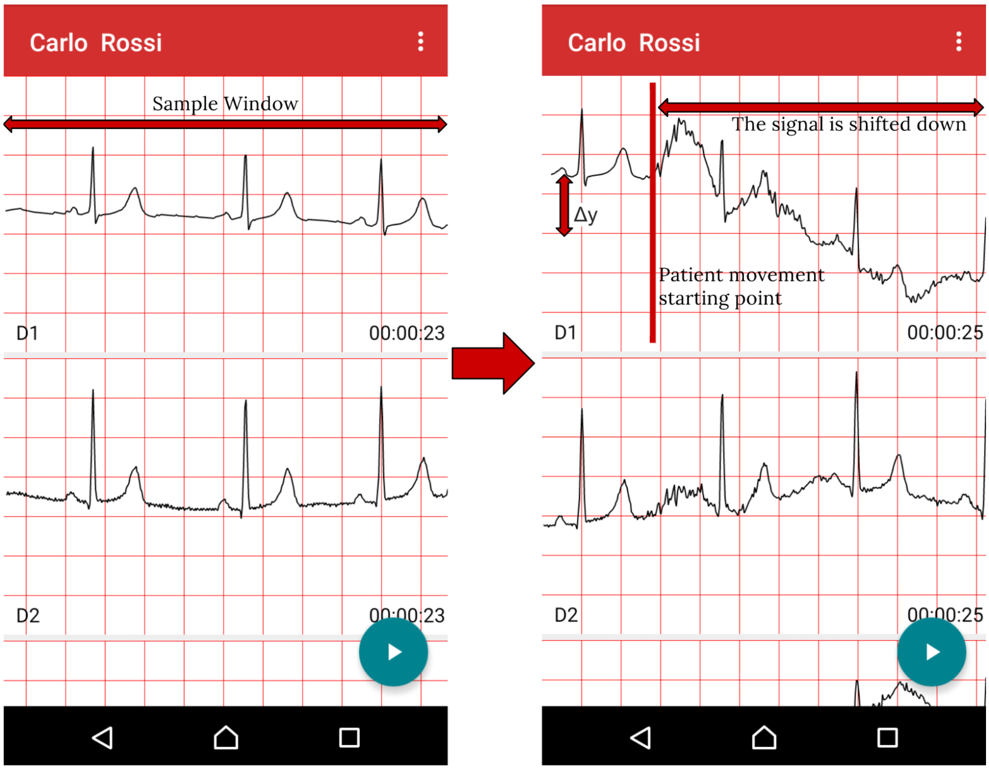


Figure 7.4: The dynamic displaying result after a patient movement, causing the baseline wander artifact.

7.4.2 Moving average filter

As discussed in the requirements section, many types of filtering are known to overcome to the baseline wander artifact. Some of them are capable to reduce at minimum the error on the output signal. But this comes with a cost in computational effort, and so there is the need to mediate between the type of filtering and its related complexity. Unfortunately in our case, putting in all the required operation, especially during real-time acquisition, where the app has to hold the bluetooth channel for transmission, interpret the transmitted signal, write to a file, derive the missing ECG lead, and plot at a reasonable rate the acquired signal, we had very limited computational availability to spend in any type of filtering in order to remove the eventual baseline wander. And so we decided to apply the most basic type of filtering during acquisition, that is the simple moving average filtering. Given that we are all conscious about the possible distortion introduced by this filter, we decided to:

- Take a reasonable size for the moving average window (two seconds at least), inasmuch if it is true that as much as the window size grows, the effectiveness of the filter decreases, by doing this, we can keep the error rate low.

- Keep the filter deactivated by default, so that the doctor will decide when will be opportune to use it.

7.5 Custom View Drawing

Now we will talk about the most costly operation performed about our application. This was the thing on which we have spent lots of work, investigating all the problematic and different possibilities that we had in order to make the best possible implementation choice. We have mentioned earlier the computational effort that needs to be spent on signal drawing, and for this reason we tried all the possible ways in order to discard the bad implementation, always having performance in our mind.

7.5.1 Custom Libraries

This was our first trial: we tried to find some libraries that could have permitted us to avoid an implementation from scratch of our drawing classes. For sure this possibility was the easiest possible. Given that our signal was not so different from other types of signals, as could be interpreted as a generic function plotted on a two dimensional system, we had quite sure that we could have found a nice plotting library and avoid useless implementation. Actually, we were able to find some well realized libraries for handling plottin. But all of them clashed with one characteristic that we was seeking for: the customization of the rendered views, as we have said previously, has to mimic as much as possible the ECG paper on the look and also respect the required standard sizing of the same. So, for this reason, we needed to discard this solution, given that some libraries permitted us to have very good displaying performance.

7.5.2 Hardware Accelerated Drawing (GPU)

Another solution, and potentially the best one, was to exploit the graphic hardware acceleration. Android is possible by using OpenGL ES, a subset of the OpenGL API designed for embedded system. The use of OpenGL can move all the graphics computations to the GPU, and so freeing up precious computing resources on the CPU. But unfortunately, in Android the usage of this libraries is not so integrated:

they are written in hardware native code, which is C. This characteristic, while could of course guarantee the best performance[20], introduce a misalignment with the language used for developing Android application, which is Java. As a result, it's a common belief that the usage of OpenGL ES in Android is quite painful, forcing many developer to switch to better alternative libraries and frameworks, like Unity, LibGDX, Cocos2D or others. Putting aside all the problematic related to its implementational effort, we decided to try OpenGL ES for the drawing part. With much surprise, this led us to an unexpected result: the results in performances were quite beneath the performance achieved using the CPU also for the application drawing. This relies on the fact that, as said earlier, the OpenGL libraries on Android are not so integrated, and developer are required to represent datatypes of the C language in the Java language. This may not seem problematic, but in a situation where all ECG samples need to be represented as classes, holding their coordinates in the ECG paper space in a corresponding matrix, and at each draw update there is the need to reallocate all the samples matrix, causing a remapping to the C data types, the performance improvements are quickly drop out. For this reason, we realized that our best chance was to relying on the CPU also for the drawing, and trying as much as possible to optimize the algorithms in order to achieve the best performances.

7.5.3 Not hardware Accelerated Drawing (CPU)

Having underlined the downsides of the previous solution, we decided to spend all our energies to implement the best possible drawing code, relying on the mechanisms provided by Android for drawing operations using the CPU. This is possible using Canvas.

Android Canvas provides the developer with the ability to create and modify 2D images and shapes. Moreover, the Canvas can be used to create and render our own 2D objects as this class provides various drawing methods to do so. Canvas can also be used to create some basic animations such as frame-by-frame animations or to create certain Drawable objects such as buttons with textures and shapes such as circles, ovals, squares, polygons, and lines.[21]

A we have mentioned in the chapter about Android concurrency exploitation, all applications run on a single thread in Android. All instructions run in a sequence on the UI thread, meaning that the second instruction will not start unless the first one is finished. The UI thread as it is responsible for drawing all the objects or views on the screen and processing all events, such as screen touches and button clicks. Now the problem is that, if we have two operations scheduled to run in the same default thread or UI thread and the first operation takes too long to finish, the system will ask the user to forcibly close the application or wait for the process to complete. This scenario is called ANR (Application Not Responding).

Given that we wanted also to provide scrolling of the differents ECG leads in our app, we could not hold this computational effort on the UI thread, which as result of drawing operations, would be blocked. So we decided to assign all the drawing operations on different threads created ad-hoc. The number of threads dedicated to drawing will be decided at run time by the application, depending on the device availability, and thus allowing device scalability.

The use of Canvas permitted us to achieve both reasonable performances, after a deep code optimization, and high customization of the rendered view.

Chapter 8

System architecture

The entire system is based on an acquisition device named Zecg, and a native mobile application on the Android platform. Our focus and main effort were on the developing of the mobile application fulfilling all the requirements. The acquisition device was instead developed and designed by Crespi Alessandro and Ulisse Pizzagalli during their thesis work at the Politecnico of Milan.

8.1 Acquisition device

Zecg is composed by the following different modules:

1. OVP: used to protect the patient from high voltage or voltage leakage.
2. LFP EMI Filter: anti-aliasing RC filter used to remove noises due to high frequencies.
3. RLD: Active electrode driver for the right leg
4. WCT: Derivator used to compute the precordials (Wilson Central Terminator)
5. PGA: 8 gain amplifier with programmable inputs
6. ADC: 8 analog-digital 16 bit 8KSa/s converter.
7. MCU: microcontroller
8. Bluetooth: bluetooth module for data transmission

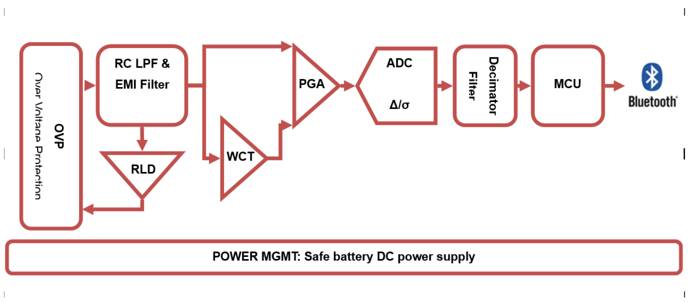


Figure 8.1: ZEcg device block diagram including all the modules components.

9. Power MGMT: to manage battery recharge and stabilizer

The core component is the Texas Instrument system on chip ADS1198. This chip has 8 input bipolar channels, representing the 8 clinical leads.
The channels 1 and 2 produce the lead I and lead II. Channel 1 measures the potential difference between the electrode RA(-) and the electrode LA(+), the channel 2 the difference between RA(-) and LL(+). Lead III and the augmented leads are obtained from a combination of lead I and lead II at software level.
V1, V2,...,V6 are computed as difference between the respective electrode and the signal related to the negative value of the WCT (Wilson Central Terminator).
The WCT signal comes from the average between RA, LA and LL and it's connected to the channels 3,4,5,6,7,8. Each channel is amplified using a programmable gain (PGA) and a CMRR, before being converted into digital. For more details about the entire device architecture we invite you to read the thesis of our colleagues Crespi Alessandro and Ulisse Pizzagalli[22] who designed and developed zecg.

8.1.1 Mobile app

The mobile application we developed for this thesis work was implemented having in mind all the user best interface design principles and the best practice starting from the planning and design phase to the final coding phase. As this application was designed for Android OS, we strictly followed Google specific standards and procedures.
We made use of Android Studio as IDE (strongly suggested by Google as main IDE to develop Android native applications). Starting from 2013 the development of native Android application moved from Eclipse + Android Plugin to Android Studio. The entire project building system has changed and moved to use the Gradle building

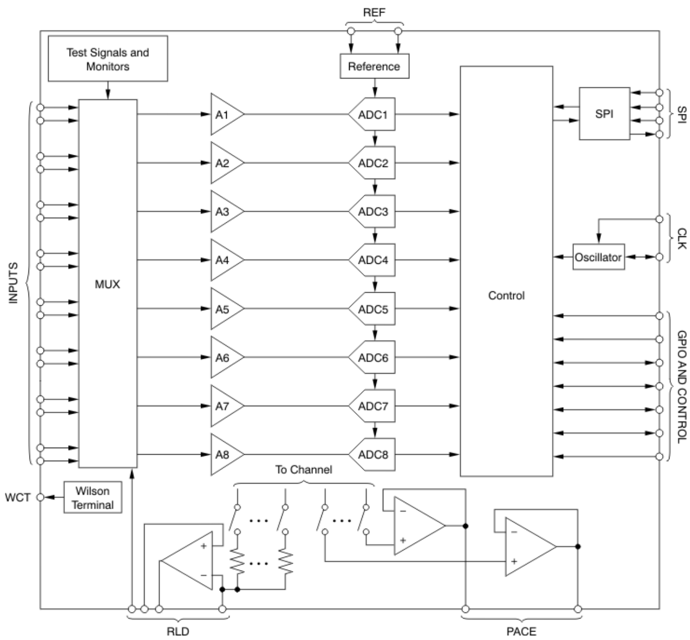


Figure 8.2: ADS1198 functional diagram showing the 8 channels representing the 8 leads used during the ECG record acquisition.

system[23]. The entire process during project building to compilation can be resumed in the following image: The most important steps along the building process are:

1. The Android Asset Packaging Tool (aapt) takes the application resource files, such as AndroidManifest.xml file and the XML files for the Activities, and compiles them. A R.java file is produced so that all the resources can be easly accessed within your application.

2. The aidl tool converts any .aidl interfaces into Java interfaces.

3. The Java compiler will compile the R.java and .aidl files generating the .class files.

4. The Dex tool will convert the .class files to Dalvik bytecode. Any 3rd libraries and .class files included in the project build will be also converted into .dex files so that they can be later packed into the final .apk.

5. All non-compiled resources(such as images), compiled resources, and the .dex files are sent to the apkbuilder tool that will output the .apk file.

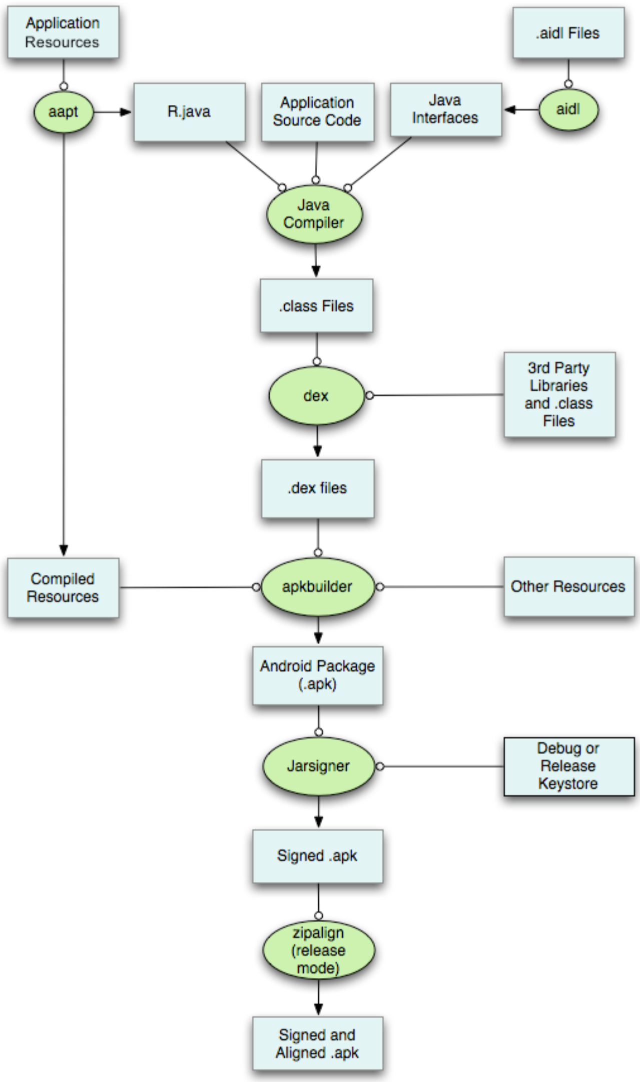


Figure 8.3: Android Build system process. How and which component are involved during an android application build and compilation.

6. Once the .apk is built, it must be signed with either a debug or release key before it can be installed to a device.

7. To reduce the size of the .apk and to decrease the memory usage for releasing mode the zipalign tool is launched.

We split the application functionalities into different packages. The packages content are divided as follow:

- Activity: it contains all the Activity used inside the Application.

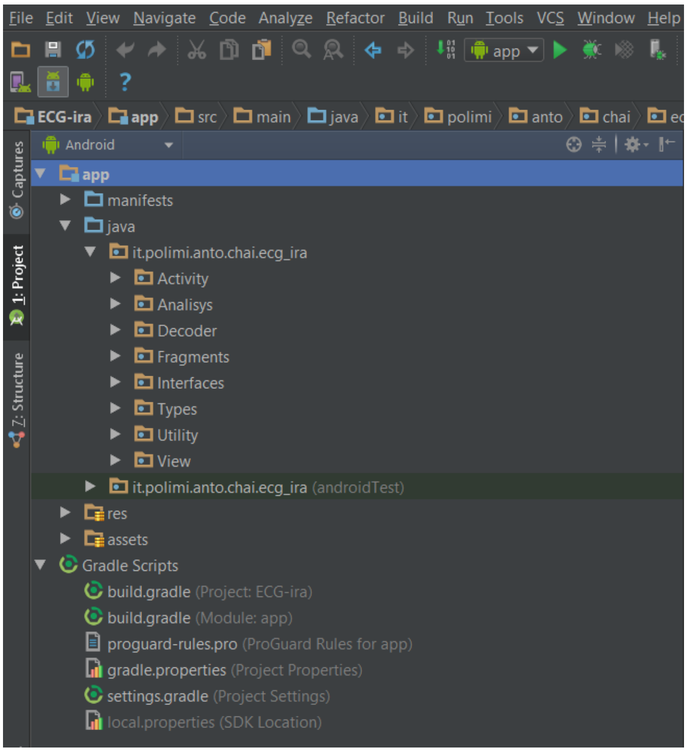


Figure 8.4: ECG-ira project structure and packages inside Android Studio IDE.

- Analysis: All the java classes used for the analysis of the ECG records; they include the Neural Network, the QRSDetector, the PWaveDetector and some signal Filters implementations.
- Decoder: this package contains the java classes in charge to decode and parse the .hea and .dat files.
- Fragments: contains some fragments used inside the application, for example the ones used to show the chronograms (Istogram, Tacogram, ST graphs).
- Interfaces: all interfaces declarations to abstract the specific application behaviour with more general one.
- Utility: some utility class such as the in application FileManager.java class, and the Adapters used within the application (for example to show ListViews or RecycleViews).
- View: it contains all the classes working directly on the View and the custom view themselves such as the View used to show the ECG grid.

The key points of this application are based on three main concepts:

- Code reuse and maintainability
- Maximize performance depending on the device constraints (cpu, memory, storage capacity)
- Usability in term of interface and interaction

Code reuse and maintainability

The most important classes representing the core of the application functionalities are all abstracted through interfaces or in case of the classes related to the View through a set of parameters. This makes the application highly customizable and easily to extend. For example if in a next future there will be a new ECG data format, it can be possible to give the app the capability to read and parse such a format just by implementing the interfaces and writing the specific code to parse such a new data format.
A concrete example is given in our code by observing that both the MITDatReader and the ZEcgDatReader extends the abstract class DatReader, so the next format reader has just to extend it as well. We know that MIT-BIH format is completely different from the ZEcg format, that why specific code to parse the file content to extract the signal is mandatory.
Any new format reader has just to override the method:

```java
public void readAndAddSample(int numOfSample, int addPosition) {
}
```

More concrete details about the effective implementation will be exploited later on in the next chapters.

Maximize performance depending on the device constraints

We do well know about the great number of constraints due to the huge number of different devices with different hardware on. We decide to make our application available starting from Android API 16 (the minimum sdk API recommended by Google to support). To overcome the problem of the difference devices and Android OS version starting from Jelly Bean (API 16) we took maximum advantages from

the device hardware by splitting the thread jobs in between the maximum number of cores available. In the class CpuInfoExtractor we discover the hardware capabilities (number of cores) and according to it we split the calculus between a certain number of threads. More the cores, more the threads we can take advantage of.
From an interface point of view instead, we are forced to depend on the density of pixel of the device screen and its size, but at the same time to accomplish our requirements related to achieve a perfect grid of squares of centimeters. We found a way to always achieve the same dimensions of square independently by the screen size of the devices by retrieving and taking in account the display exact pixels per inch size in the X dimension. Then, we computed the number of pixels needed to achieve the right sizing. In this way, we solved the problem of having same dimensions and so metrics on different devices. On the other hand, if this method is quite functional and device independent, on devices with different screen size (width for example) the number of grid cells may vary from just a few on small screen, to many of them on tablets. This is a direct consequence due to the difference in number of pixels and the pixels size itself (some are square, most of them are rectangle).

Usability in term of interface and interaction

One mobile application is usable if it does what the user expects it to do when interacting with it.
We followed all Google guidelines in term of user experience using native view and patterns. We took advantage of the last API features, such as RecycleViews instead of the old classic ListViews. We made use of the button ripple effects available starting from Lollipop (Android API 21), but at the same time we provided to the older Android OS version the selector effect which still gives a nice response to the user interaction with command and buttons. We made use of the typical android Preference Settings so familiar to Android users and most important we always inform the user about the operations going on so he never feels lost inside the application.

Chapter 9

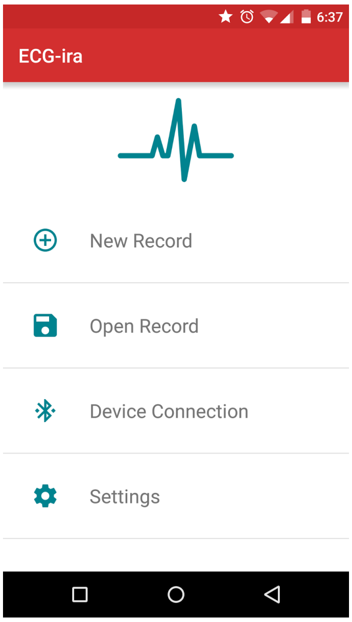
Final result

ECG-ira (ECG instant rapid analyzer) is a fully working mobile software application able to replace and fulfill most of the functionalities of an analog desktop application. The solution was designed according to the best mobile application development principles and it resulted working fine on all the smartphones[1] we tested it on. In this chapter we will show the final results providing significant screenshots of the application and metrics related to the performances evaluated and calculated with the application running on different devices.

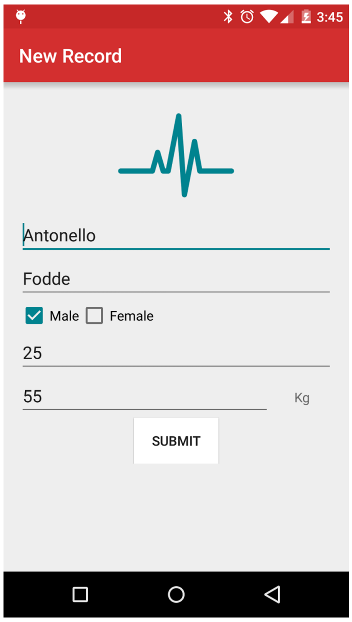
9.1 App Screens

By launching the application the first screen is the home screen. It is the starting point, here the user can create a new record (starting a new acquisition with the ZEcg device), he can open the list of records (previously acquired or already present in the device system folder), he can discover and establish a connection with an acquisition device for future acquisitions and he can access to the application settings section where he can customize the application behaviour and tune some settings like the default folder to store in and open the records from.
By clicking on the “New Record” button option (see figure 9.1a), the user will be asked to fill a form with his data as a patient. This data will be then stored within a header file (with .hea extension) together with the file containing the effective record (.dat file extension). The personal data are only used by the doctor to identify the

[1]Samsung Galaxy SIII Mini, Samsung Galaxy Note N7000, OnePlus X, Sony Xperia Z3 Compact, Sony Tablet Z, Huawei P8 Lite



(a) The home screen of the application ECG-ira.

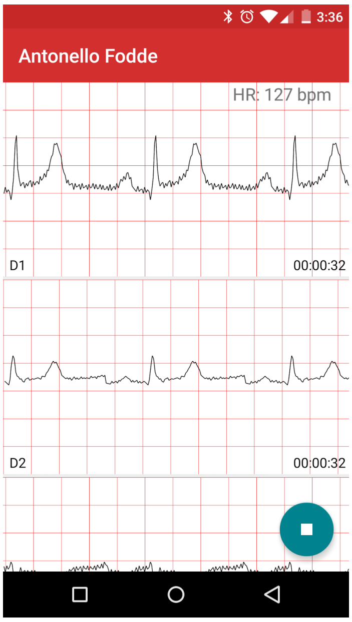


(b) The form to be compiled before a record can start. It is necessary to bind any record to a patient data in order to avoid confusion between ecg records. This data is also useful for a fully understanding of the record itself

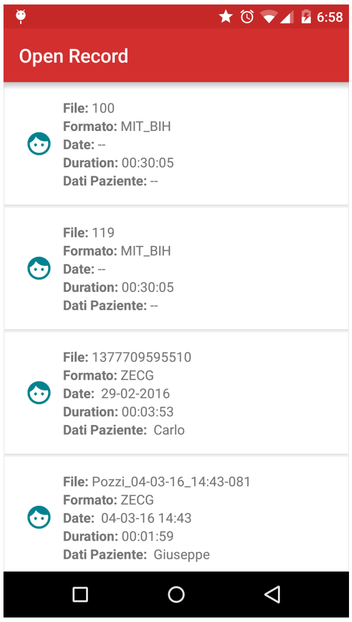
Figure 9.1: Two screens of the application, respectively the home screen and the form screen to be compiled by the patient

patient records.

Over the submission of the form the file .hea is immediately created to store all the information.
If the device is already connected with an acquisition device Zecg, then the application tries immediately to establish a connection to receive data. Otherwise it opens a connection request activity to enable the Bluetooth and start to discover nearby Bluetooth devices.



(a) A real time ECG acquisition from the Zecg acquisition device.



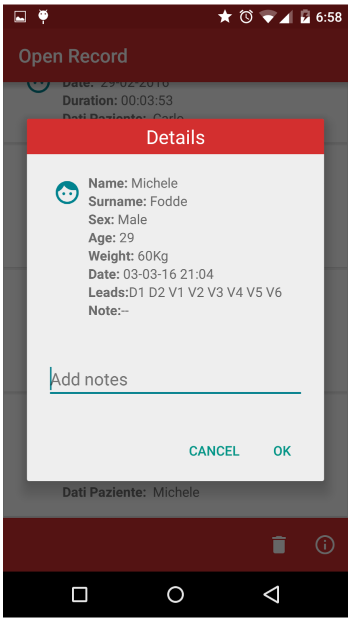
(b) List of records within the device default folder

Figure 9.2: Two screens of the application representing the realtime acquisition and the list of records within the device

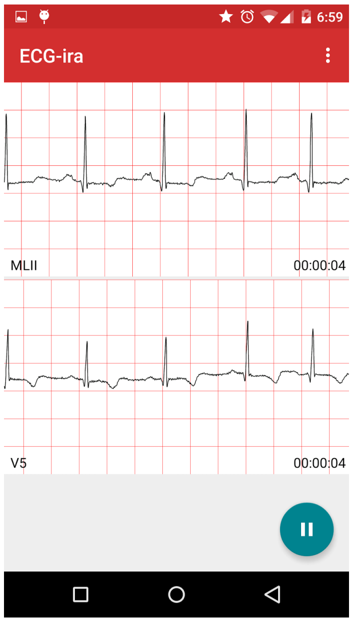
Going back to the home screen (figure 9.1a), by clicking on the "Open record" option button the user will access a list of ECG records saved on its own device on a predefined root folder within the device system memory (the default folder can be changed within the setting section).
For each record the most relevant information are highlighted:

- File Name
- File Format
- Acquisition Date
- Duration in time of the record
- Name of the patient

For more details about each record it is possible to long press on the desired record to show up a ToolBar shown in figure 10.5.



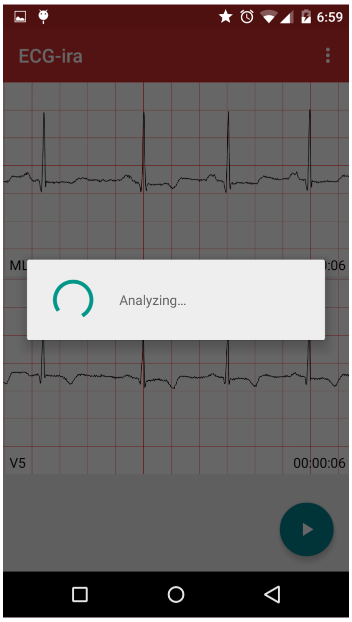
(a) Screen triggered by long pressing on a list item. At the bottom of the screen there is a ToolBar showing the info and the delete options described above.



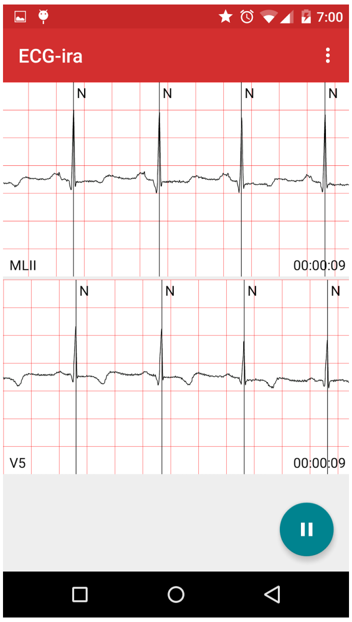
(b) An example of record opening and plotting. In the screen a MIT-BIH record format was opened.

Figure 9.3: Two screens of the application representing the info dialog for a certain record and the visualization of that record

The ToolBar is show at the bottom, it gives the possibility to show more details about the record and the patient information. In the side screenshot the user clicked on the “i” info button so the .hea file containing the user information are shown. Plus the user (in this case the doctor) can add personal notes to the specific record and the specific patient. The notes will be permanently stored within the .hea file.
The other option within the ToolBar is a delete record option in case the user decide to remove that specific record, assuming it was taken with too noises or it was just a test record.
By opening a record the application will fetch the .dat file and will plot its content over the ECG paper view. It is possible to scroll the record strip just by touching and dragging the screen with a finger, otherwise it is possible to reproduce the record as it was recorded with the exact acquisition speed time by clicking on the green round button at the bottom right.



(a) Progress displaying during the analysis process over the record.



(b) Screen of the record plot with annotations results from an analysis over the first lead (MLII).

Figure 9.4: Two screens of the application representing the info dialog for a certain record and the visualization of that record

The top ToolBar option menu is used to trigger the analysis algorithm over the opened record. If the user triggers this command, due to the great amount of operations behind the analysis process we force the user to wait for the result till the process ends.
The result of an analysis is the plot of the annotation letters (N=normal, V=ventricular, A=atrial) related to each detected beats and its diagnosis. In the screenshot we can see four heart beats marked as Normal. The annotations covers all the record length and even if it is done on one lead only (D2 as default analysis signal), it is shown on all the other leads.

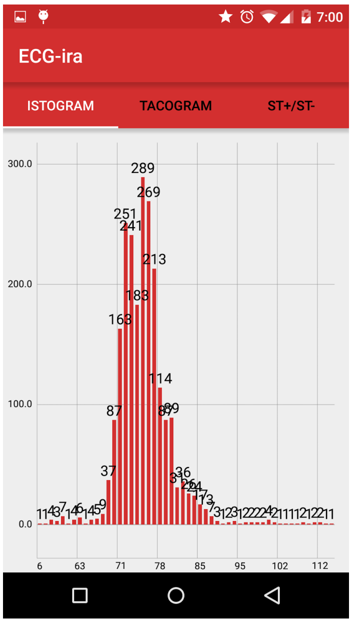
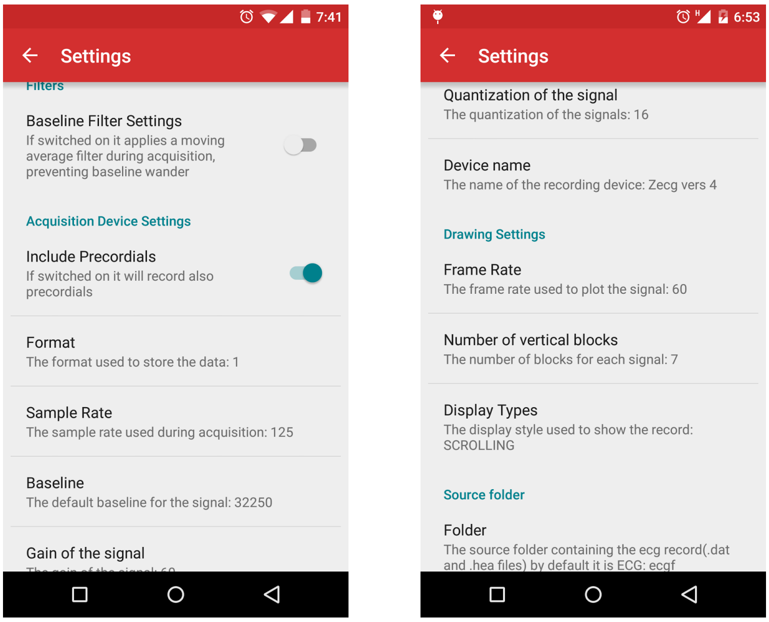


Figure 9.5: Screen of the three graphs generated after an analysis of the records. They are graphs summing up the complete record characteristics. Useful for a general overview of the patient record and health status.

Another result from the analysis is the generation of three different graphs useful to support a better analysis over the heart overall behaviour as they sum up the total record "statistics". They are:

- **Istogram** : on the X-axis we have an ordered values of the heart beats measures in bpm. On the Y-axis the occurrences of beats at that bpm value.
- **Tacogram**: on the X-axis we have the number of heartbeats retrieved by the analysis. On the Y-axis the bpm values of each heart beats.
- **ST+/ST-**: this graph shows instead, for each heartbeat, the amount of area below the ST segment and the area above that segment. This is useful especially to detect some specific arrhythmias.



(a) First parameters option within the Custom Settings

(b) Second parameters option within the Custom Settings

Figure 9.6: The customization section of the application. A list of available settings parameters to tune the application behaviour.d

About the customization setting for the application we made a long setting screen with lots of customizable parameters.
The first is related to the application of a baseline filter during the record visualization in order to reduce the baseline wandering effect. The second category of settings are related to the acquisition phase. It includes the acquisition format, the gain, the sample rate, quantization and the name of the acquisition device (for example it can be: ZEcg vers 2/3/4).
The third category is related to the drawing settings. These settings will affect only the graphical views such as the height of each strips, the way the strip is reproduces (by scrolling or by using the old style oscilloscope effect) and the preferred frame rate to be used.
The last category is to tune the default folder to store the new acquired records and from which open the records within the application (figure 9.2b).

9.1.1 Performance

In this section we reported some performance results from different aspects such as Memory usage, CPU usage and response time during a run of the application performed on different devices with installed different Android OS and different hardwares. The tests were performed on the following devices dated respectibely on 2011 and on 2015:

- Samsung Galaxy Note N7000
- OnePlus X

The table 9.1 shows the specification details of these 2 smartphones.

The results are not to be considered absolutely valid for all the above device families.

Model	OnePlus X	Samsung Galaxy N7000
Brand	OnePlus	Samsung
Year	November 2015	October 2011
Display Type	AMOLED capacitive touchscreen, 16M colors	Super AMOLED capacitive touchscreen, 16M colors
Display Size	5.0 inches	5.3 inches
Resolution	1080 x 1920 pixels (441 ppi pixel density)	800 x 1280 pixels (285 ppi pixel density)
OS	Android OS, v5.1.1 (Lollipop)	Android OS, v4.1.2 (Jelly Bean) (upgraded from v2.3.5)
CPU	Quad-core 2.3 GHz	Dual-core 1.4 GHz Cortex-A9
Memory(internal)	16 GB	16 GB
RAM	3 GB	1 GB

Table 9.1: The two test devices specification details.

The performance can be affected by many different factors that may not be related with the applications itself; hence the results may vary from a test to another. Since Android OS is in charge to manage the system memory and split the available resources between all the installed applications, having our application on foreground doesn't necessary mean we can reserve all the power and memory resource for our business. Part of the system resources are allocated to run the Garbage Collector which itself has its own timing when cleaning and deallocating other application resources. Then

all the applications having background tasks also need to be kept alive, for example notification services or messaging services.
It is not easy to measure the application performance in a clean absolute way, harder is to compare the results coming from different devices, that is why we will only report the results related to each devices and we will analyze how the application performs on each.

9.1.2 Evaluation

We evaluated the performance considering three application states:

- **Idle state**: the application is open on a simple basic screen such as the home screen or the record list screen.

- **Plotting state**: the application is performing a plotting of a record

- **Analysis state**: the analysis algorithm is performed over the record

Memory

Before going in details about the memory usage result for the application performing on the different devices, there is some basics knowledge about the analysis tools used and the way to read them that the reader should be aware of.
Android Studio provides a Memory Monitor Tool allowing the user to track how his application performs in term of memory usage. Apart of a generic view of memory usage there is a more detailed view that let you observe how your app's memory is divided between different types of RAM allocation. The most important and relevant values to care of are:

- Private (Clean and Dirty) RAM: The memory being used by ONLY your process. This is the bulk of the RAM that the system can reclaim when your app's process is destroyed. The most important is the private dirty RAM, which is the most expensive because it is used by only your process and its content exists only in RAM so it can't be paged to storage. (Android does not use SWAP). All Dalvik and native heap allocations you make will be private dirty RAM; Dalvik and native allocations you share with the Zygote process are shared dirty RAM.

- Proportional Set Size (PSS): This is a measurement of your app's RAM use that takes into account sharing pages across processes. Any RAM pages that are unique to your process directly contribute to its PSS value, while pages that are shared with other processes contribute to the PSS value only in proportion to the amount of sharing. For example, a page that is shared between two processes will contribute half of its size to the PSS of each process.

- Dalvik Heap: The RAM used by Dalvik allocations in your application.

- .so mmap and .dex mmap: The RAM used for mapped .so (native) and .dex (Dalvik or ART) code.

- .art mmap: Amount of RAM used by the heap image which is based off the preloaded classes which are commonly used by multiple apps. It does not count towards your app heap size.

- .Heap: The amount of heap memory for your application.

```
Applications Memory Usage (kB):
Uptime: 135182 Realtime: 135180

** MEMINFO in pid 4578 [it.polimi.anto.chai.ecg_ira] **
                  Shared  Private    Heap     Heap     Heap
             Pss   Dirty    Dirty    Size    Alloc     Free
          ------  ------   ------  ------   ------   ------
  Native      40      32       40    3480     3451       24
  Dalvik    3409   15896     3108   13575    13008      567
  Cursor       0       0        0
  Ashmem       0       0        0
Other dev      4      28        0
 .so mmap   1009    2388      592
.jar mmap      0       0        0
.apk mmap    137       0        0
.ttf mmap      2       0        0
.dex mmap   1948       0       12
Other mmap   454     320       36
 Unknown    1373     692     1364
   TOTAL    8376   19356     5152   17055    16459      591

Objects
               Views:      36         ViewRootImpl:        1
         AppContexts:       2           Activities:        1
              Assets:       3        AssetManagers:        3
       Local Binders:       6        Proxy Binders:       15
    Death Recipients:       0
     OpenSSL Sockets:       0

SQL
         MEMORY_USED:       0
  PAGECACHE_OVERFLOW:       0          MALLOC_SIZE:        0
```

Figure 9.7: An example of detailed view of an application memory usage.

The memory tests were executed running the application on the same screens and opening and plotting always the same record. It was used the 100.dat from MIT/BIH format because it is the longest and biggest record.

Samsung Galaxy N7000 This is the oldest device according to its release date. To make this device compatible with the minimum API requirement (API 16) the device was upgraded to android Jelly Bean (API 16).
The hardware instead was the same from the manufacturer since its release on 2011. During the test the device performed quite well. It never crashed due to memory leakage or Out of Memory Exception.
Let's start observing the memory usage during the three application states:

Idle state The memory usage looked constant over the time as expected. The

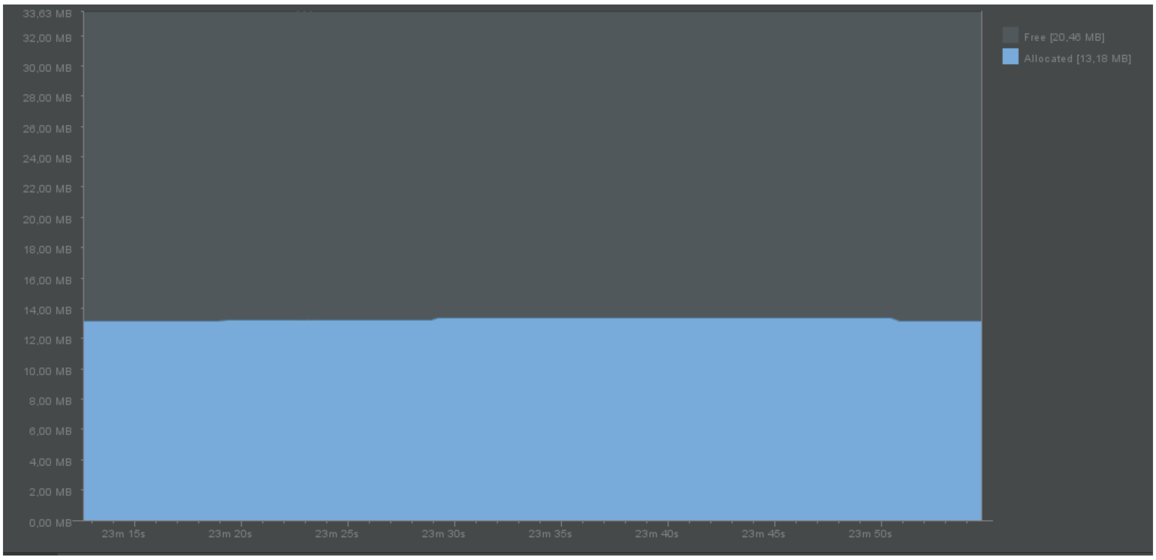


Figure 9.8: Memory usage during application idle state. It is quite constant (due to user interaction it can slightly increase or decrease if no interaction at all for a while).

memory usage graph above shows the actual memory stack allocated for the application and how many of that is used by the application. As it can be seen on the top right the application is using 13.18 MB and 20.46 are still free. The total device memory is not shown. In case the application hits the upper bound, let's say it occupies the other free 20.46 MB, then the OS will allocate more memory for the actual application by garbage collecting resources from other background applications or actually killing other low priority processes (all the processes not in foreground are considered with lower priority).

Here are more detailed information about the memory usage and resources allocations during idle state:

```
Applications Memory Usage (kB):
Uptime: 135182 Realtime: 135180

** MEMINFO in pid 4578 [it.polimi.anto.chai.ecg_ira] **
                    Shared  Private     Heap     Heap     Heap
              Pss    Dirty    Dirty     Size    Alloc     Free
           ------   ------   ------   ------   ------   ------
   Native      40       32       40     3480     3451       24
   Dalvik    3409    15896     3108    13575    13008      567
   Cursor       0        0        0
   Ashmem       0        0        0
Other dev       4       28        0
 .so mmap    1009     2388      592
.jar mmap       0        0        0
.apk mmap     137        0        0
.ttf mmap       2        0        0
.dex mmap    1948        0       12
Other mmap    454      320       36
  Unknown    1373      692     1364
    TOTAL    8376    19356     5152    17055    16459      591

 Objects
               Views:       36         ViewRootImpl:        1
         AppContexts:        2           Activities:        1
              Assets:        3        AssetManagers:        3
       Local Binders:        6        Proxy Binders:       15
    Death Recipients:        0
     OpenSSL Sockets:        0

 SQL
         MEMORY_USED:        0
  PAGECACHE_OVERFLOW:        0          MALLOC_SIZE:        0
```

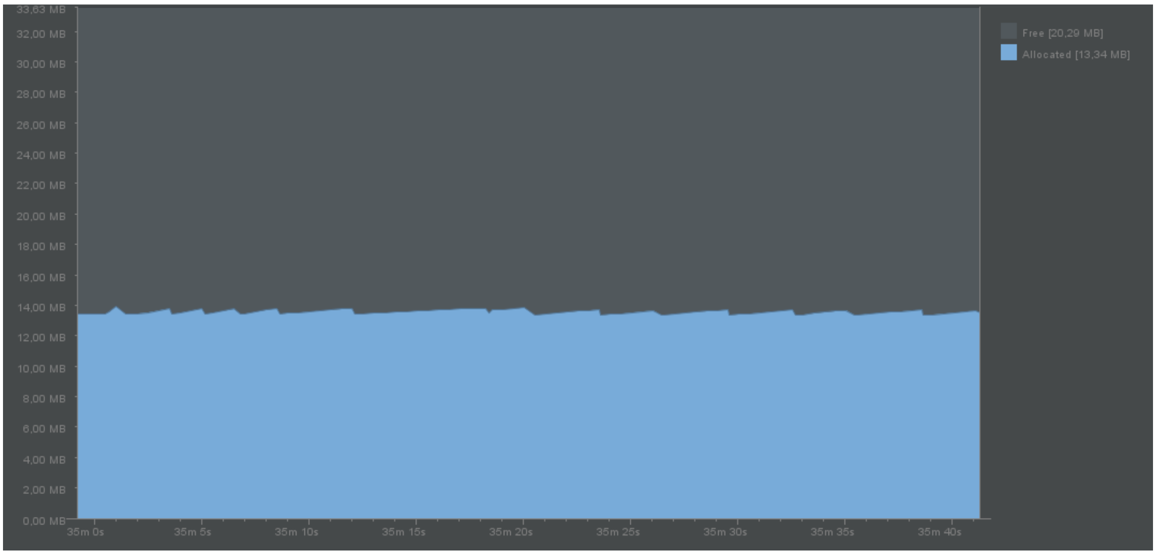
Figure 9.9: Details about memory usage and allocation over the different RAM section for the application EC-ira.

Plotting state During a record plotting the memory usage increases since the drawing process itself consume memory by allocating new resources and by the resource manipulations and usage.

The waves in the memory usage graph (figure 9.10a are due to the garbage collector calls which free unused resources. This is really an interesting Garbage Collecting behaviour, because we can observe it collecting resources with a certain regularity rate (somehow very often).

The detailed information about memory usage divided by types of RAM used are shown in the figure 9.10b.

The Private Dirty Dalvik in increased from Idle state from 3108 kB to 5224 kB. The TOTAL Pss increased from 8376 kB in Idle state to 12343 kB during a plot.



(a) Memory usage during a record plotting. The waves are due to Garbage collector calls to free resources. The final resource usage in this phase is rather constant as well.

```
Applications Memory Usage (kB):
Uptime: 282419 Realtime: 282416

** MEMINFO in pid 4578 [it.polimi.anto.chai.ecg_ira] **
                         Shared  Private     Heap     Heap     Heap
                   Pss    Dirty    Dirty     Size    Alloc     Free
                ------   ------   ------   ------   ------   ------
       Native       44       28       44     4444     4382       61
       Dalvik     5518    15576     5224    14599    13584     1015
       Cursor        0        0        0
       Ashmem        0        0        0
    Other dev        4       28        0
     .so mmap     1259     2488      784
    .jar mmap        0        0        0
    .apk mmap      149        0        0
    .ttf mmap       75        0        0
    .dex mmap     2448        0       12
   Other mmap      569      320       36
      Unknown     2277      516     2272
        TOTAL    12343    18956     8372    19043    17966     1076

 Objects
               Views:      205         ViewRootImpl:        3
         AppContexts:        4           Activities:        3
              Assets:        3        AssetManagers:        3
       Local Binders:       12        Proxy Binders:       19
    Death Recipients:        0
     OpenSSL Sockets:        0

 SQL
         MEMORY_USED:        0
  PAGECACHE_OVERFLOW:        0          MALLOC_SIZE:        0
```

(b) Memory usage by RAM types during a record plot.

Figure 9.10: Two memory usage views, relatively the realtime view with the application running and the per process memory view and allocation.

Analysis state This is the most interesting part of the memory usage results. We can clearly distinguish the application behaviour just by looking at the memory stack. The analysis is the more complex and intensive memory usage step.

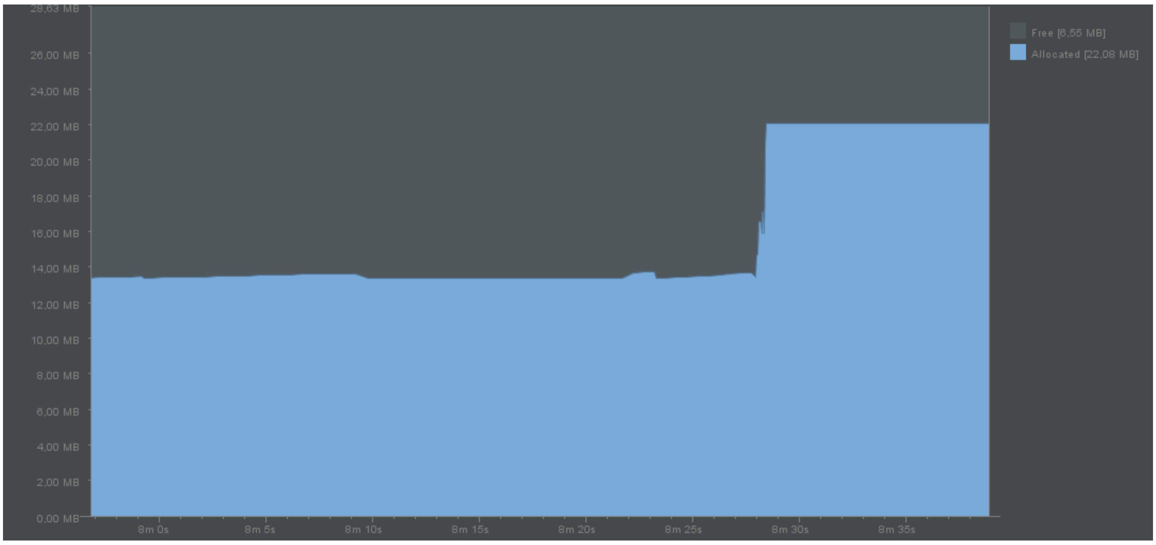


Figure 9.11: Memory usage during an analysis launched over a record. The step is due to the record allocation in a memory buffer.

At seconds 0m 28s the user clicked on the analysis button. The peak on the memory usage graph is due to the creation of the buffer to store the entire record in order to perform the analysis algorithm on it. At 10m34s the analysis algorithms finish to

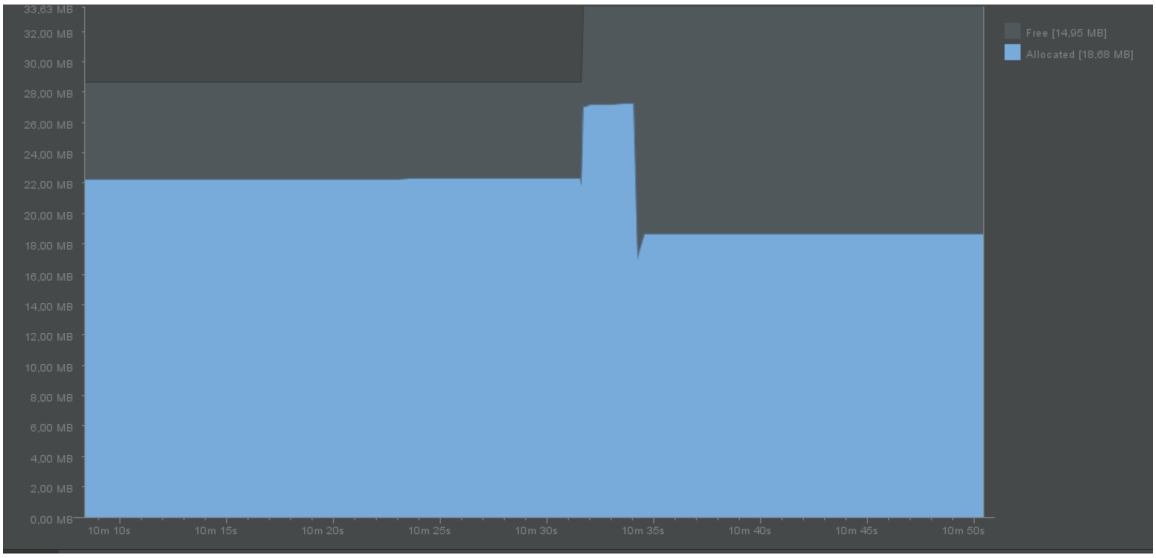


Figure 9.12: Memory usage at the exact time the algorithm performs the final steps by computing the ST+/St- areas (execution finish at 10m34s).

perform. More resources are instantiated for computational purpose of the algorithms.

As at 10m34s the algorithms terminate, all the resources can be freed and only results are returned.
As you can see in the memory usage graph, the OS kind of predicted or expected the application to require and allocate more resources, so it allocates more spaces for it even if the upper bound of 28.40 MB wasn't reached. But as soon as the resources allocated were of no use anymore, it triggered the Garbage Collector to free them.
In figure 9.13 is the detailed information about the overall memory usage which is divided by RAM types. We can easily observe that the Private Dirty Dalvik became

```
Applications Memory Usage (kB):
Uptime: 818527 Realtime: 818524

** MEMINFO in pid 4578 [it.polimi.anto.chai.ecg_ira] **
                   Shared  Private     Heap     Heap     Heap
             Pss    Dirty    Dirty     Size    Alloc     Free
          ------   ------   ------   ------   ------   ------
  Native      48       24       48     5604     5108      299
  Dalvik   15941    15160    15664    34439    22620    11819
  Cursor       0        0        0
  Ashmem       0        0        0
Other dev      4       28        0
 .so mmap   1218     2476      796
.jar mmap      0        0        0
.apk mmap    231        0        0
.ttf mmap     76        0        0
.dex mmap   2580        0       20
Other mmap   880      320      256
 Unknown    3255      428     3252
   TOTAL   24233    18436    20036    40043    27728    12118

 Objects
               Views:      230         ViewRootImpl:        4
         AppContexts:        4           Activities:        3
              Assets:        3        AssetManagers:        3
       Local Binders:       23        Proxy Binders:       20
    Death Recipients:        0
     OpenSSL Sockets:        0

 SQL
         MEMORY_USED:        0
  PAGECACHE_OVERFLOW:        0          MALLOC_SIZE:        0
```

Figure 9.13: Memory usage details by RAM types.

the triple with respect to its size during the plotting phase (5224 kB –¿ 15664 kB) and the TOTAL Pss doubled. This behaviour is due to the record copy into memory and all the data structures used to compute the analysis.
Much of the other field didn't changed their values nor allocation size, for example the .so mmap was the same both in the plotting state both in the analysis state.

As final result we can conclude that during the entire process and states of the application ECG-ira on the Samsung Galaxy N7000, the application performed fairly well consuming resources as it needed and releasing them as soon as possible. We observed also that the OS Garbage Collector overall behaviour was to trigger as often as possible as soon as the resources in no more used.

OnePlus X This device is the most advanced one we tested the application on. With it's 3GB RAM memory and its powerful quad-core processor we really expect no issues, but let's dig into the results.
For a better understand of some terms, you are invited to read the introduction of the Memory Chapter.

Idle state The idle state is as well predictable, the memory usage is stable. It

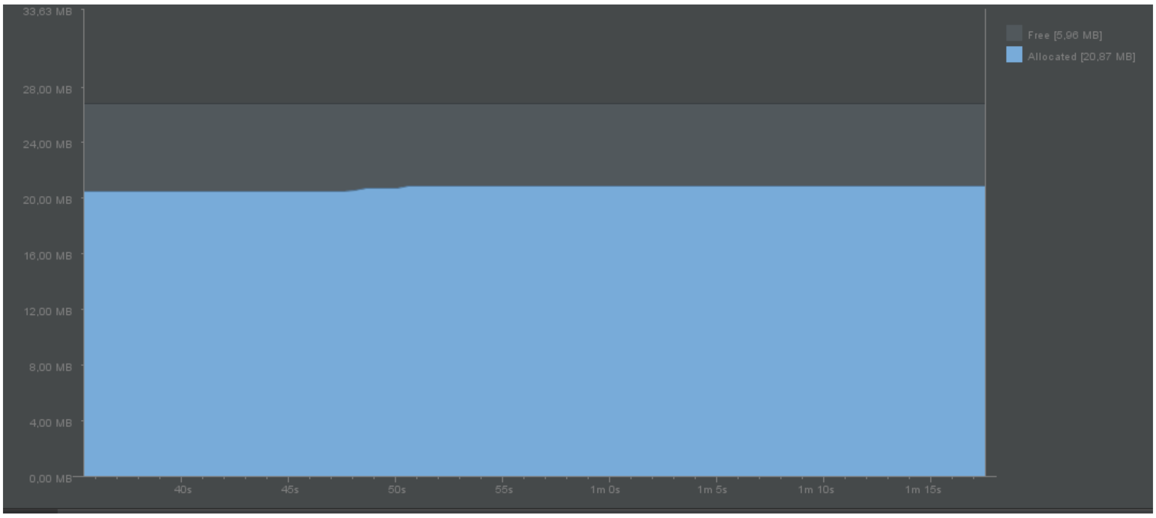


Figure 9.14: Memory usage of ECG-ira in Idle state.

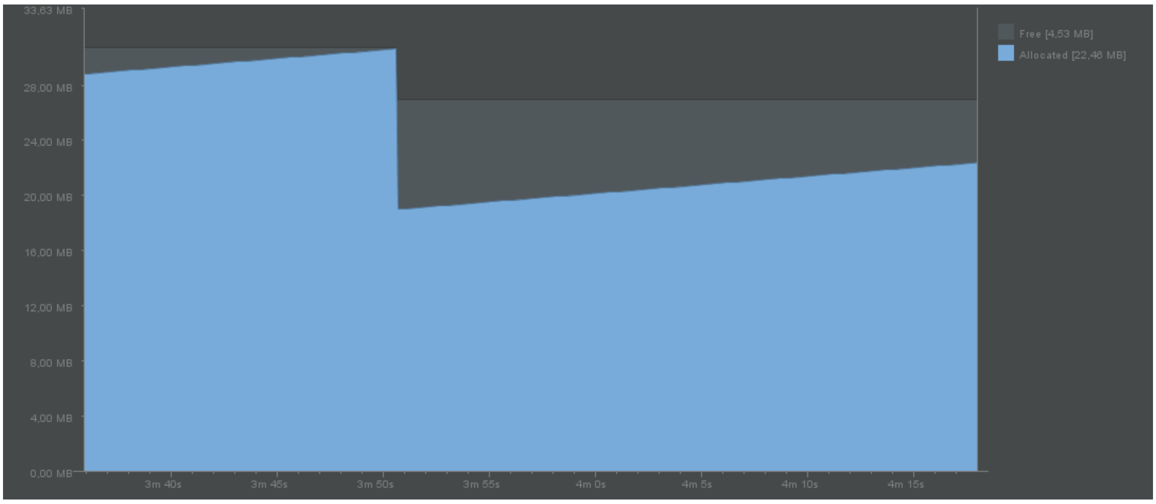
is relevant to observe that the device consume much more memory (20.87 MB) with respect to the idle state values measured on the Samsung Galaxy N7000 (13.18 MB). This can be explained with the fact that firstly the two OS are firstly completely different and rely on different API. The OnePlus X device relies on the API 21 while the Samsung N7000 on API 16. Also the screen size and pixels density is not comparable since the OnePlus X doubles the old Samsung device.
Apart using more resources the final result is the same since the idle state doesn't show any memory leakage. More information are provided in the figure 9.15. An interesting data from the above table is that the Private Dirty Native Heap and the Private Dirty Dalvik Heap are really close to have the same value. This means that some views used inside the application makes use of the Graphical layer offered by

```
Applications Memory Usage (kB):
Uptime: 9629164 Realtime: 12331506

** MEMINFO in pid 7388 [it.polimi.anto.chai.ecg_ira] **
                   Pss  Private  Private  Swapped     Heap     Heap     Heap
                 Total    Dirty    Clean    Dirty     Size    Alloc     Free
                ------   ------   ------   ------   ------   ------   ------
  Native Heap     4281     4240        0        0    12288     6599     5688
  Dalvik Heap     5552     5164        0        0    27470    21411     6059
 Dalvik Other      576      576        0        0
        Stack      112      112        0        0
      Gfx dev     3182     2772        0        0
    Other dev        5        0        4        0
     .so mmap      857      336      120        0
    .apk mmap      136        0       24        0
    .ttf mmap       49        0        4        0
    .dex mmap     3156        0     3152        0
    .oat mmap      681        0       72        0
    .art mmap      802      584       20        0
   Other mmap        4        4        0        0
   EGL mtrack    39808    39808        0        0
      Unknown      156      156        0        0
        TOTAL    59357    53752     3396        0    39758    28010    11747

 Objects
               Views:      158         ViewRootImpl:        2
         AppContexts:        4           Activities:        2
              Assets:        3        AssetManagers:        3
       Local Binders:       10        Proxy Binders:       19
       Parcel memory:        3         Parcel count:       14
    Death Recipients:        0      OpenSSL Sockets:        0

 SQL
         MEMORY_USED:        0
  PAGECACHE_OVERFLOW:        0          MALLOC_SIZE:        0
```

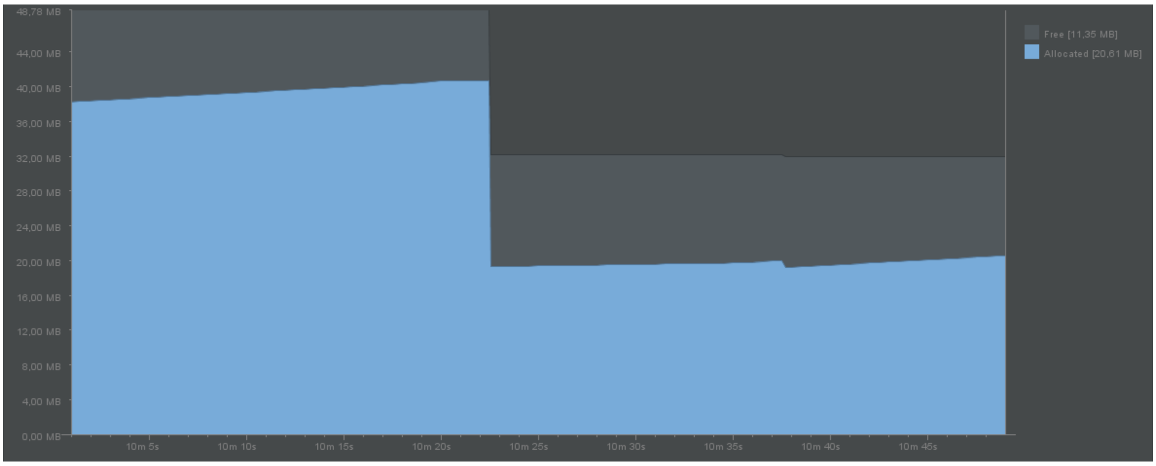
Figure 9.15: Memory usage of ECG-ira in Idle state by RAM types.

OpenGL (it means they are hardware accelerated). Actually most of the Total Pss and Private Dirty allocations comes from the EGL mtrack, in other words it refers to the memory consumed by Graphics layer.

Plotting state During the plotting phase we can observe a different Garbage Collector behaviour with respect to the one observed on the Samsung N7000. In the figure 9.16b we can see a system GC call triggered when the application reaches the upper bound of the memory heap allocated for the application process. The OS waits to trigger a GC call until the process fills and hit the memory heap. Lollipop GC policy is then completely different from the Jelly Bean (Samsung N7000). It allows application to fill the heap and only after that it triggers a GC and estimates the new heap size. For a test purpose We choose to trigger a GC call before the process can fill the heap. You can see in figure 9.16b that the forced GC call cleared the heap (the first big step) and resized the heap according to the resource need. After that the heap keeps being filled with new allocations and also old allocations, but at 10m37s we triggered another forced GC call (second small step). We can now realize for real how many resources are really used during plotting phase and how many are just put in the GC queue. If we keep calling forced GC, probably we would achieve the same



(a) Memory usage during a record plotting. We can see that the step is due to a GC call.



(b) Memory usage during a record plotting. We force two GC calls as they can be seen considering the the first big step and the second small one.

Figure 9.16: Two memory usage views during a normal app execution, on the second figure a GC is called before the memory reach the heap

behaviour as the one observed on the Samsung N7000 (a series of memory waves).

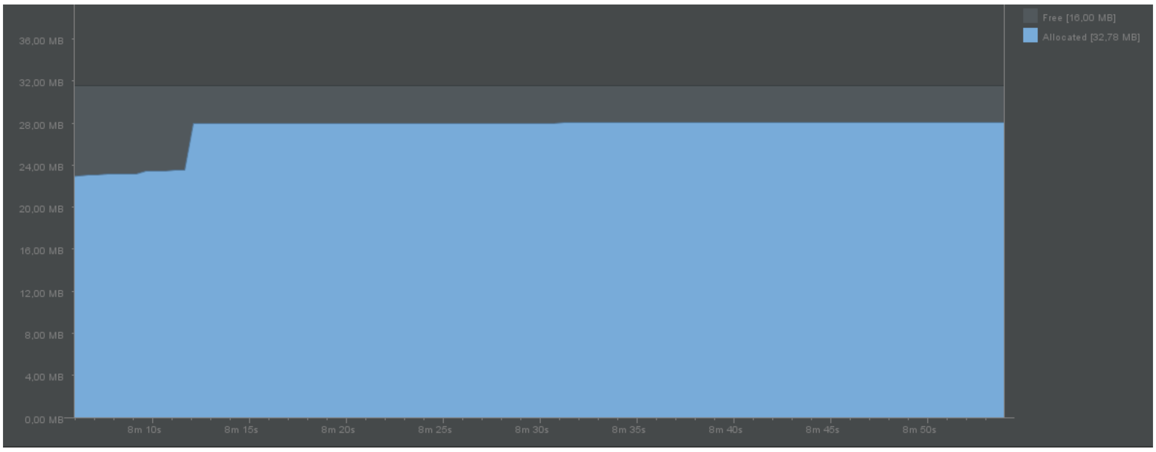
A detailed view of memory allocation by RAM types is shown below: As the

```
Applications Memory Usage (kB):
Uptime: 10316126 Realtime: 13018467

** MEMINFO in pid 7388 [it.polimi.anto.chai.ecg_ira] **
                   Pss  Private  Private  Swapped     Heap     Heap     Heap
                 Total    Dirty    Clean    Dirty     Size    Alloc     Free
                ------   ------   ------   ------   ------   ------   ------
  Native Heap     6158     6120        0        0    16384     7761     8622
  Dalvik Heap     6899     6512        0        0    33527    21963    11564
 Dalvik Other      868      868        0        0
        Stack      184      184        0        0
      Gfx dev     4162     3752        0        0
    Other dev        5        0        4        0
     .so mmap      887      332      124        0
    .apk mmap      150        0       28        0
    .ttf mmap       55        0        4        0
    .dex mmap     3692        0     3688        0
    .oat mmap     1162        0      260        0
    .art mmap     1582      836      488        0
   Other mmap       24        4        0        0
   EGL mtrack    39808    39808        0        0
      Unknown      156      156        0        0
        TOTAL    65792    58572     4596        0    49911    29724    20186

 Objects
               Views:      221         ViewRootImpl:        3
         AppContexts:        7           Activities:        5
              Assets:        3        AssetManagers:        3
       Local Binders:       26        Proxy Binders:       22
       Parcel memory:        4         Parcel count:       16
    Death Recipients:        0      OpenSSL Sockets:        0

 SQL
         MEMORY_USED:        0
  PAGECACHE_OVERFLOW:        0          MALLOC_SIZE:        0
```

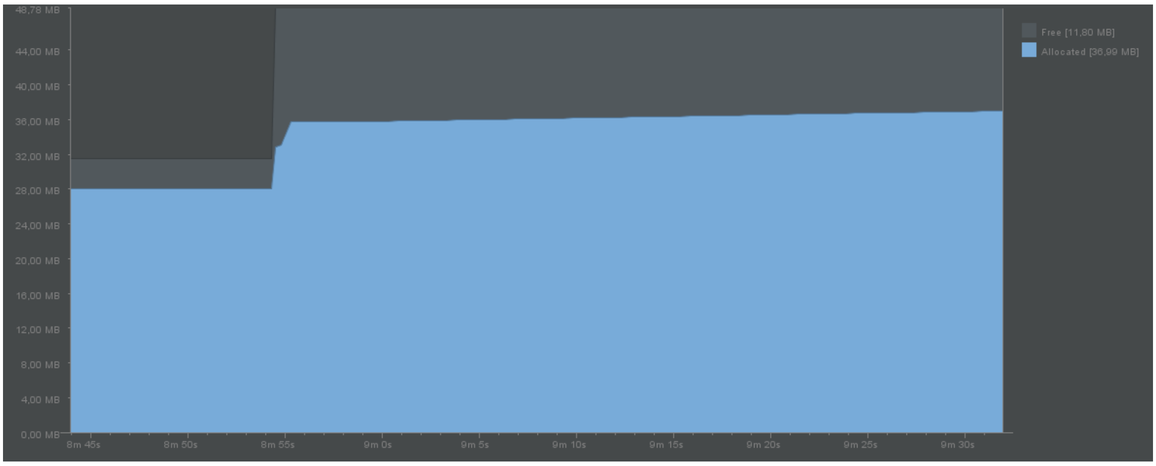
Figure 9.17: Memory usage details by RAM types.

plotting state requires more drawings and dynamic data usage (updates of data structure and views updates) there is an obvious increase of resources needs.

Analysis state Considering how the GC behaves the analysis state is kind of predictable. Instead of calling a GC after the buffer becomes useless as the analysis algorithm finishes, the OS instantiates more memory heap for the process as it needs and sets up a new and higher upper bound. The GC call will be triggered only when the new bound is reached. We already know that most of the allocated resources are to be garbage collected but since the limit of the heap size will not be hit no GC calls will be triggered. We can observe that all the resources used during the analysis are kept in memory.



(a) Memory usage when a record analysis is called. The step in the memory usage graph is due to the buffer allocation for the record.



(b) Memory usage when a record analysis is called. The step in the memory usage graph is due to the analysis algorithm running on the record.

Figure 9.18: Two memory usage views during the execution of the analysis algorithms, on the first figure 9.18a the step sign the execution satrting point, in figure 9.18b the step is due to the end of the execution

Due to the creation of the copy record buffer the Dalvik Private Dirty is doubled (from 6512 kB in plotting state to 14268 kB in analysis state) and since we pause all the views the EGL mTrack is reduced (less views need memory).
The overall memory usage keeps being the same also for the OnePlus X device, even if on this device the GC behaviour is completely different, the resource used are limited and deallocated when there is no more need of them. Having less GC calls improved the application performance since any GC call affects and keeps the CPU busy for while. Also the use of an additional graphical layer improves the application

```
Applications Memory Usage (kB):
Uptime: 10242259 Realtime: 12944600

** MEMINFO in pid 7388 [it.polimi.anto.chai.ecg_ira] **
                   Pss  Private  Private  Swapped     Heap     Heap     Heap
                 Total    Dirty    Clean    Dirty     Size    Alloc     Free
                ------   ------   ------   ------   ------   ------   ------
  Native Heap     6910     6872        0        0    16384     8708     7675
  Dalvik Heap    14655    14268        0        0    32596    28939     3657
 Dalvik Other      816      816        0        0
        Stack      184      184        0        0
      Gfx dev     3990     3580        0        0
    Other dev        5        0        4        0
     .so mmap      895      340      124        0
    .apk mmap      150        0       28        0
    .ttf mmap       55        0        4        0
    .dex mmap     3692        0     3688        0
    .oat mmap     1159        0      260        0
    .art mmap     1582      836      488        0
   Other mmap       24        4        0        0
   EGL mtrack    30688    30688        0        0
      Unknown      156      156        0        0
        TOTAL    64961    57744     4596        0    48980    37647    11332

 Objects
               Views:      232         ViewRootImpl:        4
         AppContexts:        6           Activities:        4
              Assets:        3        AssetManagers:        3
       Local Binders:       23        Proxy Binders:       21
       Parcel memory:        4         Parcel count:       16
    Death Recipients:        0      OpenSSL Sockets:        0

 SQL
         MEMORY_USED:        0
  PAGECACHE_OVERFLOW:        0          MALLOC_SIZE:        0
```

Figure 9.19: Memory usage when a record analysis is called. The memory usage is divide by RAM types.

responsiveness and performance, but it also comes with a more intensive memory usage that not all devices can afford. In case of the OnePlus X hardware memory limitation is not a great deal with its 3GB RAM memory, nor the performance limitation is really a limit, but since the application has to be run on other devices as well, these principles should always be considered.

CPU

The CPU performance tests were done by executing the same actions on the same views. We decide to provide CPU usage during an application run on different devices. As in the memory tests we decide to track CPU usage with the application on the three different states:
Idle state, Plotting state and Analysis state. The CPU usage is reported into percentages and is related to the amount of CPU power used by the Kernel and by the User (in this case the User is the application process).

Samsung Galaxy N7000 This device has a Dual-core 1.4 GHz Cortex-A9, performed fairly well even with its large screen 5.3 inches and its 800x1280 pixels (285 ppi pixel density).

Idle state During Idle state, when the application is simply kept on foreground on a static screen, we can observe there are no cpu usage leakage. The peaks in the

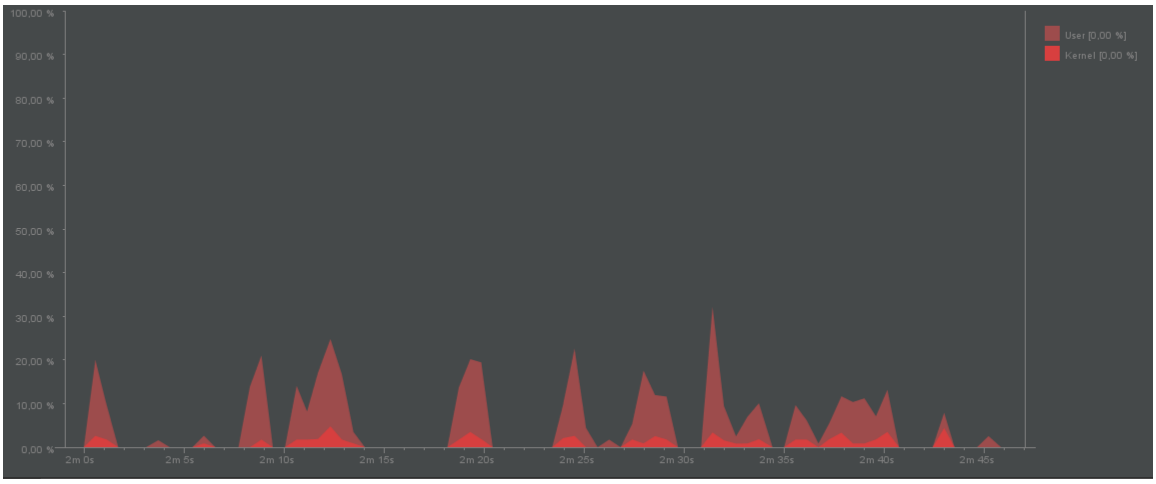


Figure 9.20: Cpu usage during idle state, the application is on foreground on a static page.

figure 10.28 are due to the user's interaction with the listview, otherwise in case of no interaction at all the graph is plain. At each peak the ListAdapter (in this case the RecycleViewAdapter) instantiates and so inflates the next item list represented by a record in the list so that the user can visualize it on the scrolling action.

Plotting state During plotting state the CPU usage results to be quite constant. In the figure 9.21, we can observe that it never exceed 60%. At minute 7m10s we selected the record 100.dat and opened it. The plotting screen keeps the cpu quite busy at a constant rate (between 40% and 58%) . This value doesn't change and it is independent from the user's interaction (not like the listView which requires cpu at each user's interaction as scrolling). Also during the automatic scrolling of the ECG strip, the cpu usage keeps being the same. This can be explained by observing that this screen and all the views were completely customized and implemented by us. As the views on screen are plotting canvases and to get the animation features we are forced to use a second thread to manage the drawings. By this way the screen content

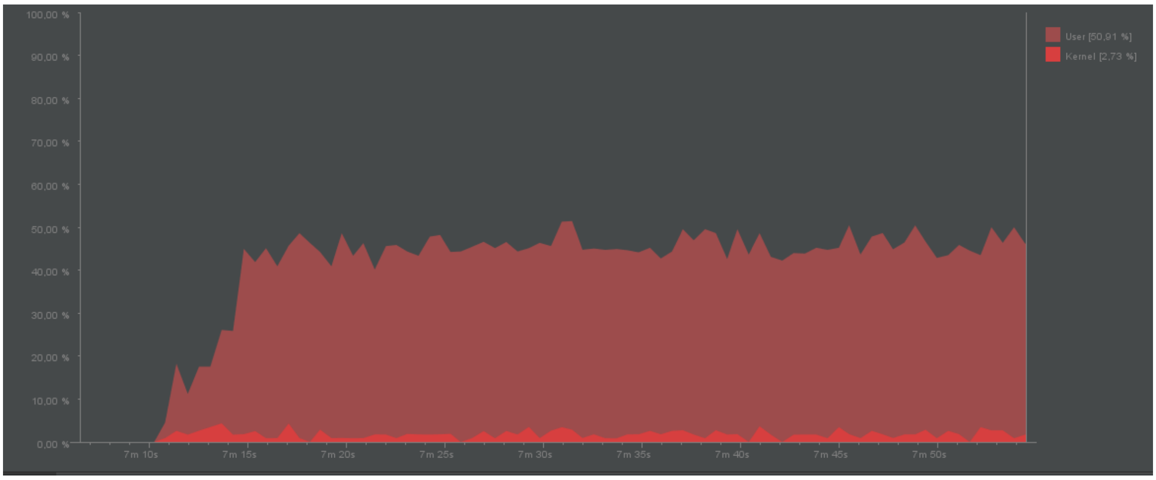


Figure 9.21: Cpu usage when a record is plot on screen.

is redrawn as often as it is possible according to the maximum frame rate set by the user but compatible the frame rate the thread can get.
As soon as the user closes the Plotting screen, the cpu usage dropped and resources are kept free. In figure 10.30 at minute 13m25s we closed the Activity with the ECG

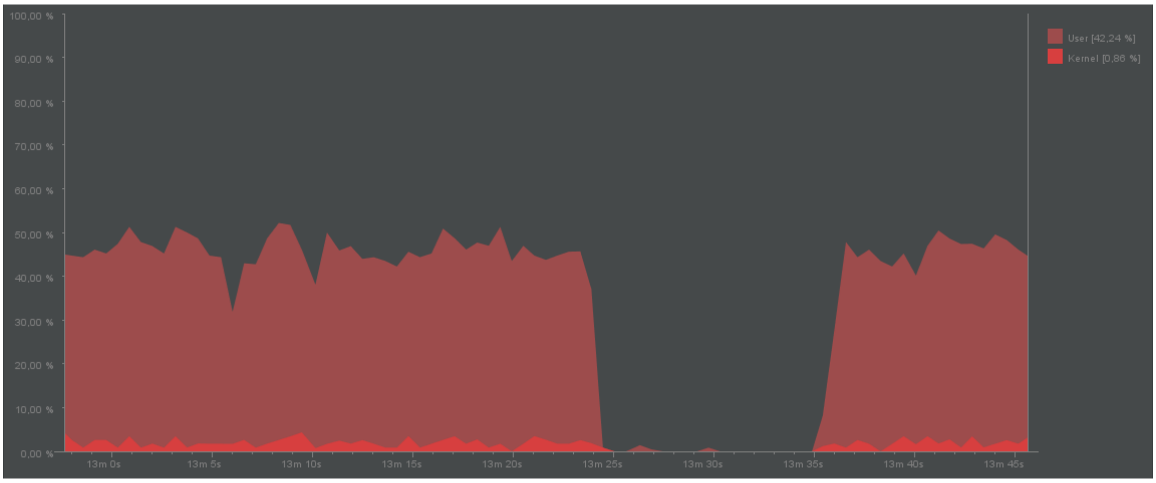


Figure 9.22: Cpu usage when we close a plotting screen and reopen it.

strip plot. We can observe that the application released immediately the cpu resources as they are no more needed. Also all the drawing threads are killed and streams closed. But as at minute 13m35s we reopened another record, the resources are reallocated and the new views show ups. New drawing threads are instantiated and a new data stream is opened to read the new record data.

Analysis state Interesting results comes from the Analysis state. As you can see in the figure 10.31, at minute 8m30s we launched an analysis command. The down peak is due to the drawing thread freezing as we stop all drawing for a while. If during the memory test we observed an increasing of memory allocation, here there is no significant changes into the cpu usage. We just pause the drawing thread and launch an AsyncTask to compute the analysis over the ECG record.

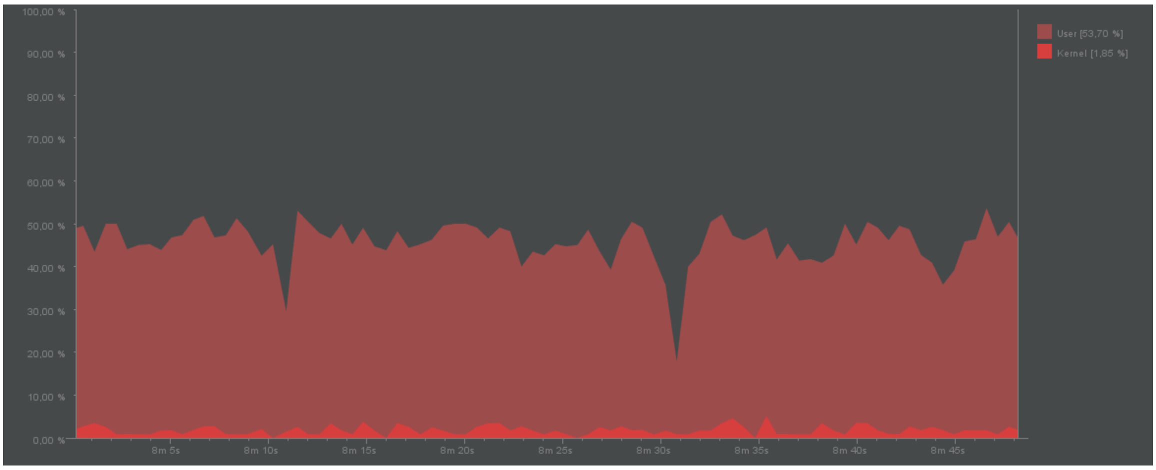


Figure 9.23: CPU usage during analysis state. An analysis command is launched over an ECG record strip.

OnePlus X CPU usage on this device didn't differ that much on the previous results on the Samsung N7000. The application achieved same performance and never exceeded 60% of the CPU capabilities. The graph suggests that in the Quad-core the average CPU usage is also smaller than the one found on the Samsung N7000's Dual-core.

Idle state On idle state the cpu usage is reduced at minimum. Less than 2% of CPU power usage is registered on static screen pages.

Plotting state During the plotting of a record the application CPU usage is constant in a range between 25% to 45%. The slope down is due to a change in activity view. The Record Activity is closed at minute 8m5s. The process (and all the threads) in charge to draw the ECG signals is killed so also resources are released as well. At minute 8m15s another record is opened and new resources and threads

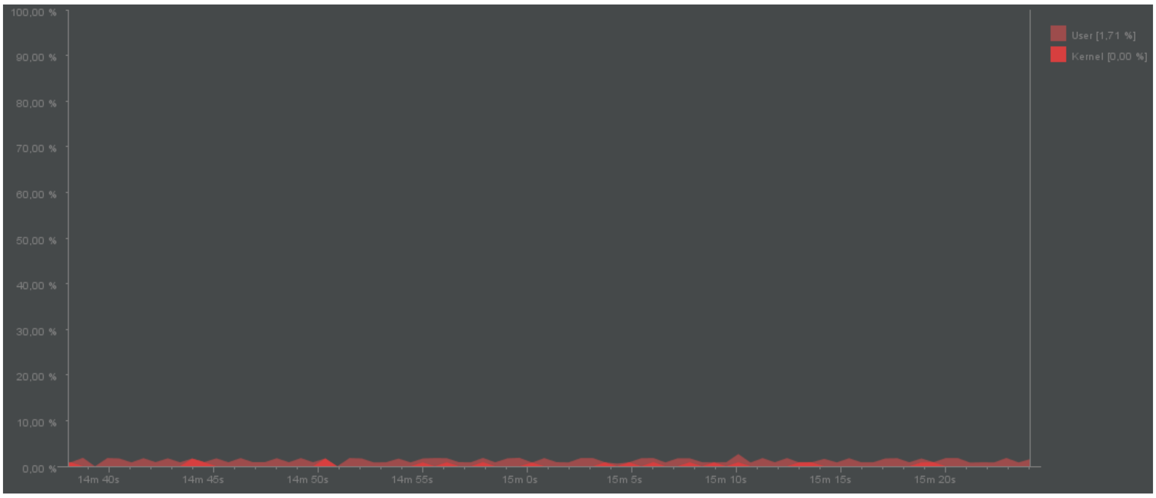


Figure 9.24: CPU usage on idle pages(app open on static screen page)

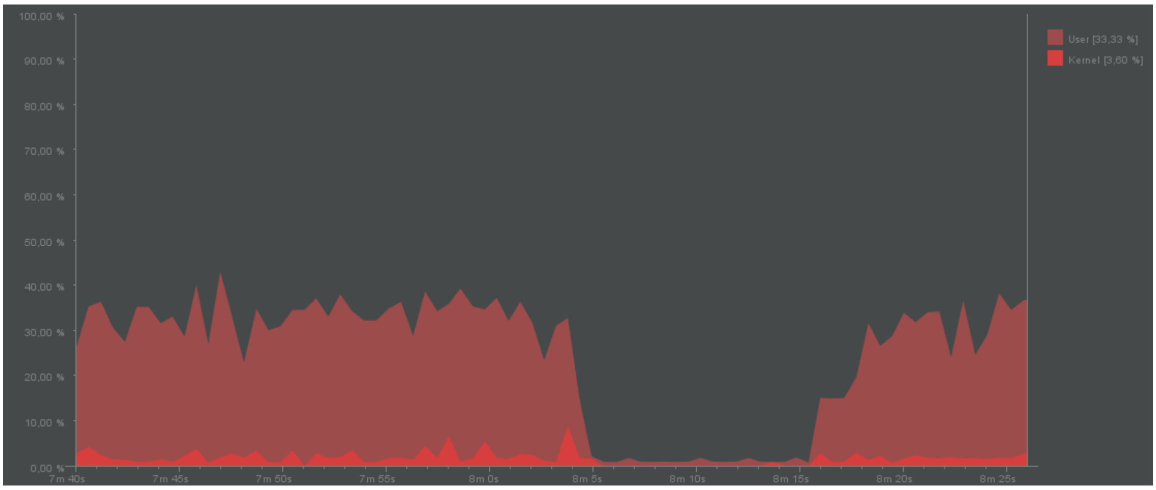


Figure 9.25: CPU usage during a record plotting)

are allocated.
We can observe that the overall cpu usage on the OnePlus X device is on average less than the average cpu usage on the Samsung N7000, even if the application overall performance results to have the same behaviour.

Analysis state During the analysis state the maximum cpu usage is not changed. It worth mentioning that even if there is no significant change in the % of cpu usage, on the OnePlus X the execution time of analysis is much shorter. In the figure 9.26 during a plotting state the analysis is launched at minute 3m30s (first white bar). The process terminates at minute 4m20s as shown by the second white bar.

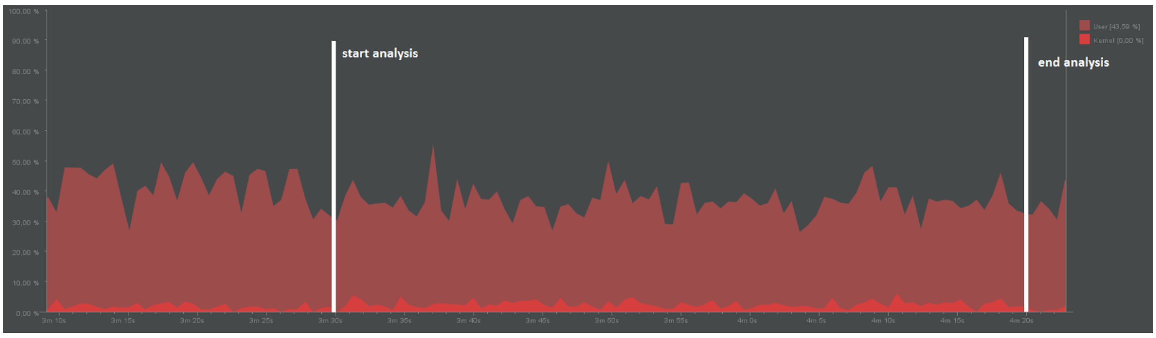


Figure 9.26: usage when the analysis algorithm is launched over a record)

9.1.3 Response Time

We performed a test about the response time and the time spent by the processor to load the plotting page and the time spent on executing the analysis algorithm.
To have accurate measurements we put inside the application additional lines of code as checkpoints.
Android OS offers an utility class to measure execution time of methods that is the TimingLogger class.
As documented into the Android references guide the class is used as follows:

```
    TimingLogger timings = new TimingLogger(TAG,"methodA");
        //do some work A
        timings.addSplit("work A");
        // do some work
        timings.addSplit("work B");
        //do some work C
        timings.addSplit("work C");
        timings.dumpToLog();
```

After a class declaration we split the time between the methods we want to check the execution time. We used such a class to measure the longest operations within ECG-ira, that is when we load the record into memory and when we execute the analysis on the record.

```
//A timingLogger implementation to measure the execution time of the two
   //longest processes.
@Override
protected Boolean doInBackground(Void params) {
```

```
TimingLogger timeLogger = new
    TimingLogger("it.polimi.anto.chai.ecg_ira.Activity.AsyncTask", "RECORD
    ANALYSIS");
short[] result = parser.getSamplesForAnalysis(index);
timeLogger.addSplit("sample extraction from file to memory");
Analysis.execute(result, rate);
timeLogger.addSplit("analysis execution");
timeLogger.dumoToLog();
}
```

The same code was executed on different devices and as expected the OnePlus X performed much better in term of execution time.
Below is reported the execution time in milliseconds for the sample extraction from the .dat file to the memory and the time spent to execute the analysis on the record. The memory load on the OnePlus X is extremely fast taking on average about 0.1

#of executions	**Memory load (ms)**	**Analysis execution (ms)**
run1	115	55038
run2	101	46429
run3	98	47775
run4	87	47910
run5	92	47087
average	98.6	48847.8

Table 9.2: Execution time for a record load in memory and its analysis on the OnePlus X device

second (98.6 milliseconds) on a record which lasts 30 minutes and dimension 1.9 MB. The analysis method is the longest in term of execution time. On average on the OnePlus X it takes about 48 seconds to execute (48.847 milliseconds). The results are not precise as the execution time takes in consideration many other independent external factors. The overall time can be less than the average or even much higher due to the number of processes running on background. Five runs are not exhaustive at all for a detailed performance analysis but it gives a general view of the performance we can get during a normal usage of the phone. The execution time on the Samsung N7000 are three time slower on average with a memory load time of 0.3 seconds and an Analysis execution time of 2.30 minutes.

#of executions	**Memory load (ms)**	**Analysis execution (ms)**
run1	275	121483
run2	320	128352
run3	358	136027
run4	242	144923
run5	334	156327
average	305.8	137422.4

Table 9.3: Execution time for a record load in memory and its analysis on the Samsung N7000 device

We observed that during other tests the application overall behaviour with respect to memory usage and amount of CPU usage is regular and the application doesn't show any kind of leakages nor crashes. The huge difference in the execution time is mostly due to the difference in the hardware and software strategy. The OnePlus X quad-core is obviously performing much faster than the Dual-core of the Samsung N7000. Also the availability of 3 GB of RAM can allow the OS not to call the GC as often as the OS on the Samsung device which has limited 1GB RAM to be shared across all the applications.

Acknowledgement

Desidero innanzitutto ringraziare...

Bibliography

[1] Blausen Medical Communications. Heart illustrations.

[2] AllianceTek. Comparison chart for mobile app development methods. Norwegian University of Science and Technology, 2011.

[3] Mediatek helio x20 - the world's first mobile processor with tri-cluster cpu architecture and ten processing cores (deca-core) and 30consumption: http://mediatek-helio.com/x20/.

[4] Simone Battaglia Diego Ulisse Pizzagalli. Algoritmi per il riconoscimento automatico di eventi aritmici ed ischemici in segnali ecg. Master's thesis, Politecnico di Milano.

[5] Nicos et al Maglaveras. An adaptive backpropagation neural network for real-time ischemia episodes detection: development and performance analysis using the european st-t database. *Biomedical Engineering, IEEE Transactions*, 45(7):805–813, 1998.

[6] GSMA Intelligence. Mobile platform wars, Feb. 2014.

[7] Gartner. Global market share held by smartphone operating systems, Dec. 2015.

[8] Lionbridge. Mobile web apps vs. mobile native apps, 2012.

[9] Redda Yonathan Aklilu. Cross platform mobile applications development. Norwegian University of Science and Technology, 2012.

[10] Palmieri Manuel, Inderjeet Singh, and Antonio Cicchetti. Comparison of cross-platform mobile development tools. In *Intelligence in Next Generation Networks (ICIN)*, pages 179–186, Oct. 2012.

[11] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. A detailed analysis of contemporary arm and x86 architectures. *UW-Madison Technical Report*, 2013.

[12] Blanco-Velasco, Binwei Weng Manuel, and Kenneth E Barner. Ecg signal denoising and baseline wander correction based on the empirical mode decomposition. *Computers in biology and medicine*, 38(1):1–13, Jan. 2008.

[13] Burattini L, W Zareba, and R Burattini. The effect of baseline wandering in automatic t-wave alternans detection from holter recordings. *Computers in Cardiology*, pages 257–260, Sep. 2006.

[14] Leski, Jacek M, and Norbert Henzel. Ecg baseline wander and powerline interference reduction using nonlinear filter bank. *Signal processing*, 85(4):781–793, 2005.

[15] Xu and Lisheng et al. Baseline wander correction in pulse waveforms using wavelet-based cascaded adaptive filter. *Computers in Biology and Medicine*, 37(5):716–731, 2007.

[16] Hu, Xiao, Ying Gao, and Wai-Xi Liu. Pattern recognition of surface electromyography signal based on wavelet coefficient entropy. *Health*, 1(02):121, 2009.

[17] Hu, Xiao, Zhong Xiao, and Ni Zhang. Removal of baseline wander from ecg signal based on a statistical weighted moving average filter. *Journal of Zhejiang University SCIENCE C*, 12(5):397–403, 2011.

[18] Manjoo and Farhad. A murky road ahead for android, despite market dominance. *The New York Times*, May 2015.

[19] Goransson Anders. *Efficient Android Threading: Asynchronous Processing Techniques for Android Applications*. O'Reilly Media, 1 edition, May 2014.

[20] Lee Sangchul and Jae Wook Jeon. Evaluating performance of android platform using native c for embedded systems. In *Control Automation and Systems (ICCAS)*, pages 1160–1163, Oct. 2012.

[21] Tahir Mir Nauman. *Learning Android Canvas*. Packt Publishing Ltd, 2013.

[22] Crespi Alessandro and Ulisse Pizzagalli. Zecg: Acquisizione, trasmissione e memorizzazione del segnale ecg. Master's thesis, Politecnico di Milano.

[23] Muschko and Benjamin. *Gradle in Action*. Manning, 2014.