Hellenic Mediterranean University

Department of Electrical and Computer Engineering

Bachelor's thesis

# Real-time processing of healthcare data from medical (electrocardiogram) pulse sensor devices

Ανδρουλάκης Μιχάλης (ΑΜ: 1945)

Supervisor: Prof. Grammatikakis Miltiadis

Committee members: Prof. Panagiotakis Spyros, and Prof. Vasilakis Kostas

Date of presentation: 10 September, 2021

# Abstract

Heart arrhythmia is a condition affecting a person's heart rhythm. In some cases, it can be life-threatening. Ambulatory electrocardiogram (ECG) devices play an important part in diagnosing arrhythmias. In recent years there is a proliferation of consumer- and medical-grade ambulatory ECG devices.

In this thesis, we focus on an open-source consumer-grade ambulatory ECG device (Protocentral's HeartyPatch). By analyzing its firmware source code, we examine how to communicate with it via TCP/IP, and develop a program to record and visualize the ECG data it transmits.

We also compare the performance and power consumption of HeartyPatch, with that of a medical-grade pulse sensor device, the STMicroelectronics' Body Gateway, by considering a soft real-time heart arrhythmia identification system.

# Περίληψη

Οι καρδιακές αρρυθμίες είναι ασθένειες που επηρεάζουν τον καρδιακό ρυθμό ενός ατόμου. Σε κάποιες περιπτώσεις οδηγούν σε επικίνδυνες καταστάσεις. Πρόσφατα, πιστοποιημένες ιατρικές συσκευές αλλά ακόμα και κινητά τηλέφωνα χρησιμοποιούνται για τη μέτρηση του καρδιακού ρυθμού (π.χ. μέσω ηλεκτροκαρδιογραφήματος) και τη διάγνωση διαφόρων μορφών αρρυθμίας.

Σε αυτή την πρακτική εργασία, επικεντρωνόμαστε στη χρήση μη πιστοποιημένης φορετής συσκευής ανάκτησης ηλεκτροκαρδιογραφήματος ανοικτού υλικού/λογισμικού (Protocentral's HeartyPatch). Αναλύοντας τον κώδικα firmware και την επικοινωνία της συσκευής μέσω TCP/IP, αναπτύσσουμε εφαρμογή ανάκτησης και οπτικοποίησης των καρδιολογικών δεδομένων.

Τέλος συγκρίνουμε την επίδοση και ενεργειακή κατανάλωση του HeartyPatch σε σχέση με πιστοποιημένο αισθητήρα (STMicroelectronics' Body Gateway), χρησιμοποιώντας εφαρμογή ανίχνευσης και κατηγοριοποίησης αρρυθμίας σε πραγματικό χρόνο (soft real time).

*This page intentionally left blank*

# Table of Contents

# 1 Introduction

## 1.1 Electrocardiography

An *electrocardiogram* (ECG) (aka. EKG) is one of the non-invasive tests used to check heart activity. It is based on the phenomenon that the heart muscle – each time it beats – produces electrical signals that can be picked up by sensors. These sensors, commonly referred to as *electrodes*, are attached to the skin at various locations, like the chest, arms, and legs. A graphical representation of the signals can then be looked up by a doctor to spot any abnormalities. [5]

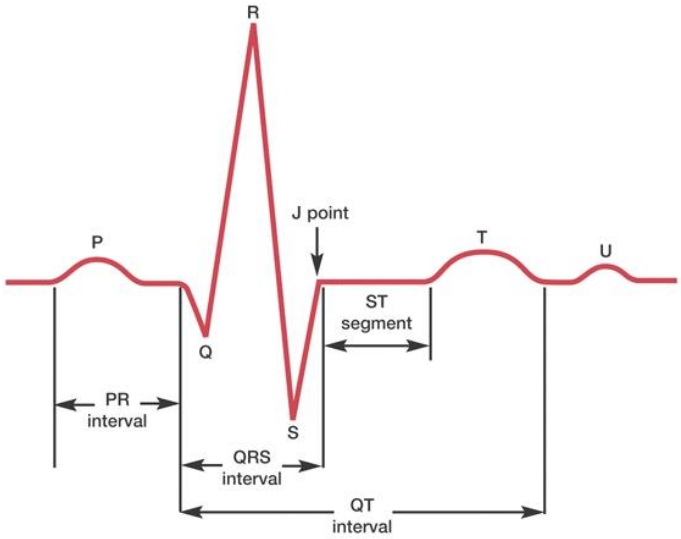


*Figure 1.1.1:* The basic pattern of electrical activity across the heart. [5]

Figure 1.1.1 depicts a typical heartbeat. It is comprised of 4 waves, named P, QRS (a wave-complex), T, and U. The duration, amplitude, and morphology of the QRS complex are useful in diagnosing various heart diseases states including, but not limited to, ventricular hypertrophy, myocardial infarction (heart attack), and heart arrhythmias. [5]

## 1.2 Heart arrhythmia

The heart has four chambers — two upper chambers (*atria*) and two lower chambers (*ventricles*). Heart rhythm is normally controlled by the so-called sinus node located in the right atrium. The sinus node produces electrical impulses that start each heartbeat. These impulses cause the atria muscles to contract and pump blood into the ventricles. The impulses then arrive at a cluster of cells called the *atrioventricular* (AV) node. The AV node sends the electrical signal to the ventricles after a small delay, during which they fill with blood. When electrical impulses reach the muscles of the ventricles, they contract, causing them to pump blood to the rest of the body. [7]

Heart rhythm problems, commonly referred to as *heart arrhythmias*, occur when the electrical impulses that coordinate the heartbeats don't work properly, causing the heart to beat irregularly. Arrhythmias are classified by the speed of heart rate that they cause (*tachycardia* (fast) or *bradycardia* (slow)) and by their origin (atria or ventricles). [7]

Some arrhythmias may be harmless, or cause bothersome signs/symptoms, while others can be deadly. Types of possibly life-threatening arrhythmias include: [7]

- Atrial flutter: may lead to complications such as stroke.
- Ventricular fibrillation: it is fatal if the heart rhythm is not restored in time.
- Long QT syndrome: may lead to fainting and in some cases sudden death.

## 1.3 Heart Rate Variability

Derivable from ECG data is the so-called *Heart Rate Variability* (HRV). It is the variation of the time interval between heartbeats. This variation is controlled by a part of the nervous system called the *Autonomic Nervous System* (ANS). It works regardless of our desire and regulates, among other things, our heart rate, blood pressure, breathing, and digestion. The ANS is subdivided into two components, the sympathetic (fight-or-flight mechanism) and the parasympathetic (relaxation response) nervous system. [6]

The brain, through the ANS, instructs the body either to stimulate or to relax different functions. It responds not only to a poor night of sleep but also to that of a delicious lunch meal. The body handles all kinds of stimuli but, persistent instigators such as stress, poor sleep, unhealthy diet, dysfunctional relationships, and lack of exercise, can disrupt this balance, and shift the fight-or-flight response into overdrive. [6]

HRV is an interesting way to identify these ANS imbalances. If a person's system is in more of a fight-or-flight mode, the HRV is low. If one is in a more relaxed state, the HRV is high. People who have a high HRV may have greater cardiovascular fitness and be more resilient to stress. HRV may also provide personal feedback about one's lifestyle and help motivate those who are considering taking steps toward a healthier lifestyle. On the other hand, research has shown a link between low HRV and depression or anxiety and is even associated with an increased risk of death and cardiovascular disease. [6]

## 1.4 Ambulatory ECG

An often-cited feature in the datasheet of an ECG machine is that of a *lead*. A lead is a "view" of the heart that a system generates by processing the input of multiple electrodes. [5]

A standard 12-lead ECG employs ten electrodes: one attached to each limb, and six across the chest. [5] Einthoven's triangle explains why there are 6 frontal leads when there are just 4 limb electrodes (left arm, right arm, left leg, right leg). In 1895, Willem Einthoven used four measuring points, by immersing the hands and foot in saltwater, as the contacts for his string galvanometer, the first practical ECG machine. Measurements are based on an imaginary inverted equilateral triangle centered on the chest with the points being the standard leads (e.g., left/right arm, left leg).

An ECG test typically lasts for a few minutes and involves a patient wearing electrodes and lying on a bed, running on a treadmill, or pedaling on a stationary bike. The test can reveal a lot of information about the patient's heart, however, when it comes to cardiac arrhythmias, it is often not sufficient to make a full diagnosis. This is because heart rate disturbances often occur only infrequently and may last for very brief time intervals. A single or even several standard ECGs taken at different times are likely to miss these events. [8]

Ambulatory ECG monitors are wearable devices that record the heart rate for prolonged time intervals. Their use increases the odds of detecting an intermittent arrhythmia. Several different types of ambulatory ECG monitoring have been developed to suit different purposes. These include: [8]

- Holter monitor: Consists of several electrodes attached to the skin and plugged into a small device that is worn around the neck. The Holter monitor is worn continuously for a relatively short amount of time (usually 1 or 2 days) during which it records every heartbeat. The device needs to be returned before any analysis is done; the doctor plays back the recordings using a system that produces a sophisticated analysis of every heartbeat recorded. Holter monitors provide the most detailed information of all ambulatory ECG monitors, but they do so only for a limited amount of time. [8]
- Event monitor: This type of device doesn't record every heartbeat, but rather, attempts to capture specific episodes of heart arrhythmia. The main advantage over Holter is that it can be employed for several weeks or even months. Many event monitors can transmit recordings of arrhythmia events wirelessly to a base station for analysis. [8]

Several consumer devices that can record an ECG also exist. Their functionality is usually an amalgam of a standard and an ambulatory ECG device. For some of them, factors such as radiation exposure, product size, form factor, ease of use, battery autonomy, electronics quality, etc., may limit their diagnostic accuracy and features, and thus do not meet the standards of ambulatory ECG devices prescribed by doctors. [9]

A few examples of ECG devices in various form factors:

KardiaMobile by AliveCor
https://store.kardia.com/products/kardiamobile

It is a handheld single-lead ECG device, cleared by the FDA, that detects atrial fibrillation, bradycardia, and tachycardia. The patient needs to place his fingers on the device for the ECG to be recorded.

Apple Watch Series 6 by Apple
https://www.apple.com/apple-watch-series-6

It is capable of generating an ECG similar to a single-lead ECG. It can detect signs of atrial fibrillation, but it is not intended for use by those who have been previously diagnosed with the condition. Apple Watch Series 6 has been tested thoroughly in a case study by Stanford University, sponsored by Apple.


Zio XT by iRHYTHM
This is is a prescription-use only, single-use, ECG monitor patch, that continuously records data for up to 14 days. It is cleared by the US FDA for use as a medical device.

D-heart
https://www.d-heartcare.com

Portable, medical-grade 8/12 lead ECG for use with a smartphone or a tablet. It is CE 1370 certified per Directive 2007/47/EC.

## 1.5 Scope & Objectives

The purpose of this thesis is to develop a software application to read the ECG recorded by a wearable ECG patch called “HeartyPatch”. This device is of particular interest because it is open-source, i.e., its hardware blueprints and software source code are both available for free. Being open-source is important because it allows for arbitrary customization of its behavior to application needs.

The HeartyPatch adheres to the idea of continuously streaming the ECG wirelessly, rather than storing it in the device (much like a Holter monitor). It supports Bluetooth and Wi-Fi for its wireless communication needs, and out-of-the-box, it utilizes Bluetooth. In this thesis, we want to communicate with the HeartyPatch via TCP/IP & Wi-Fi. This will prompt us to flash a custom, slightly-altered official version of firmware that relays ECG data using TCP/IP via Wi-Fi.

In addition, we evaluate the sensor by measuring the performance of a soft real-time arrhythmia classification software application. There are two series of tests. In the first one, the system will analyze the ECG produced by HeartyPatch, and in the second one, by the Body Gateway, which is a medical-grade wearable ECG patch. It will be interesting to see how the HeartyPatch, which is not certified for medical use, fares very well compared to a similar, but higher-cost, medical-grade ECG device. And while we are at it, we also assess the battery autonomy of both devices.

In summary, the objectives of our study are as follows,

**Primary objectives**

- Make the HeartyPatch relay its data via Wi-Fi.
- Analyze HeartyPatch’s firmware to figure out how to communicate with it.
- Develop an application to read the ECG recorded from the HeartyPatch in soft real-time.

**Secondary objectives**

- Measure and compare the performance of a soft real-time arrhythmia classification application in conjunction with HeartyPatch and Body Gateway.
- Assess battery autonomy of both the HeartyPatch and the Body Gateway.

# 2 HeartyPatch

*HeartyPatch* is a complete open-source (both software and hardware), single-lead ECG wearable patch, developed by the Indian company Protocentral[1]. At the time of writing, it was neither available for sale, nor had been certified for medical use. Recently, a Protocentral team in collaboration with MIT and Mayo Clinic is working towards a new medical-grade pulse sensor for ECG and R-R interval monitoring based on HeartyPatch.


*Figure 2.1:* The HeartyPatch PCB. [1]

## 2.1 Overview

The PCB (Fig. 2.1) roughly measures 65mm x 42mm x 4mm.


*Figure 2.1.1:* HeartyPatch's board. [1]

[1] https://protocentral.com/

On the backside (Fig. 2.1.1, left), there are two standard snap-style receptacles for ECG electrodes.

On the front-side (Fig 2.1.1, right), there is *Espressif's ESP32 SoC* (System-on-a-Chip). It is a low-power chip designed for mobile/wearable electronics with Wi-Fi and Bluetooth/Bluetooth-low-energy (BLE) capabilities. HeartyPatch firmware is built with *Espressif's IoT Development Framework* (ESP-IDF) which integrates a modified version of the FreeRTOS operating system.

To the right of the Espressif SoC lies the *Maxim Integrated MAX30003 ECG AFE* (Analog Front End) chip which is responsible for the core low-level functionality: ECG recording and R-to-R detection.

On the far-right end of the board, a micro-USB port can be used to flash new software, log debug information, recharge the battery, or even provide power without being connected to a battery. In normal operation, it is advised to use a battery lest, since the input to the device is noisy[2].

A battery can be connected as shown in the next picture (the HeartyPatch kit includes a 450 mAh LiPo battery), similar to most other ECG sensor devices.


*Figure 2.1.2:* The HeartyPatch with a battery attached. [2]

To use the HeartyPatch, it must first be turned on. Use the small power switch that lies on the bottom left side in Fig. 2.1.2. A LED just to the left should turn red, once powered on. Then, one must place it correctly near the heart as shown next,

[2] There are other factors that may also affect the quality of the data. See https://heartypatch.protocentral.com/#frequently-asked-questions.

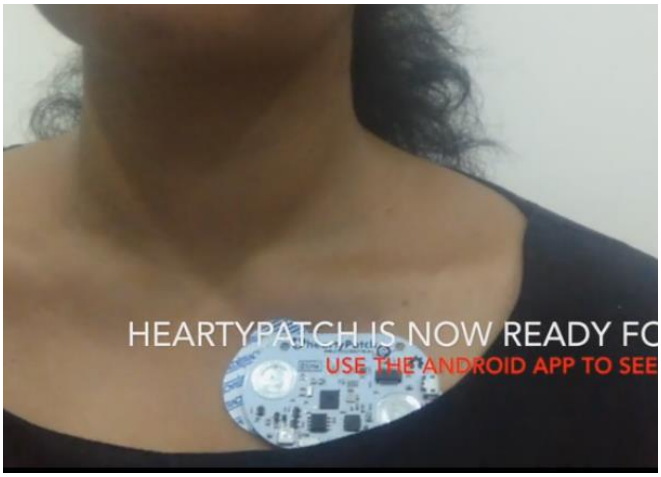


*Figure 2.1.3:* Wearing the HeartyPatch correctly. [1]

The device's default (preloaded) firmware performs heart rate and R-to-R measurements and relays the data over BLE adhering to the Heart Rate Service[3] Bluetooth standard. Moreover, HRV-related data are sent using a non-standard protocol. An Android app that displays in real-time the output signals of this firmware version is available on Google Play. For more information, refer to: https://play.google.com/store/apps/details?id=com.protocentral.heartypatch. (Alternative versions, such as EliteHRV can also be used.)

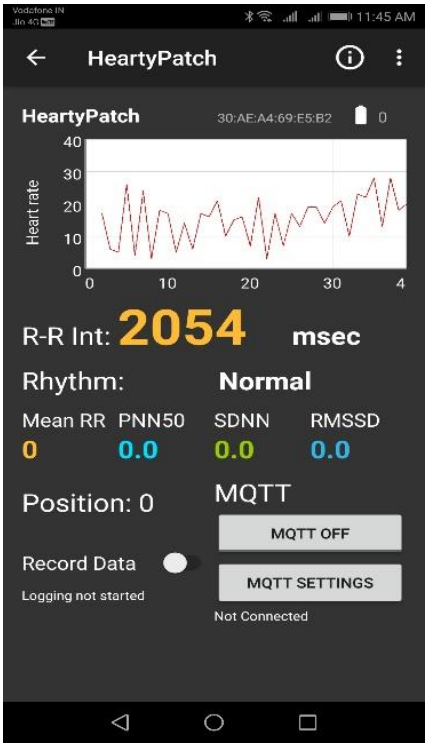


*Figure 2.1.4:* The official Protocentral Android app. [1]

[3] https://www.bluetooth.com/specifications/specs/heart-rate-service-1-0/

## 2.2 Setting up for Wi-Fi/TCP mode

For this thesis, we utilize the Wi-Fi/TCP-capable version of the official firmware. We initially flash its official precompiled binaries found at https://github.com/patchinc/heartypatch/releases. We opt to flash the device from a Windows PC since the procedure is simpler.

The board schematics found at https://github.com/patchinc/heartypatch/tree/master/hardware reveal that the micro-B USB port is connected to a UART chip (Fig. 2.2.1). A *Universal Asynchronous Receiver-Transmitter* (UART) is a hardware device for asynchronous serial communication commonly used in embedded systems.

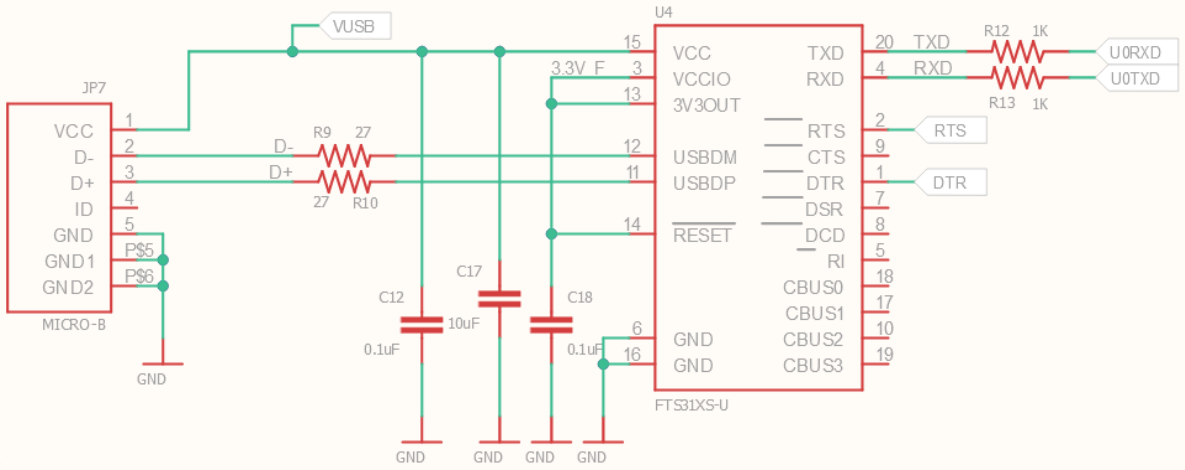


*Figure 2.2.1:* HeartyPatch's (version 2.3) FT231XS-U USB to UART schema.

Therefore, before flashing the firmware, we must be able to communicate with the HeartyPatch via its UART. This is achieved by installing the driver for the UART chip from FTDI's website: https://ftdichip.com/drivers/vcp-drivers/.

Next, we download Espressif's secure flash download tool from https://www.espressif.com/en/support/download/other-tools. Running the tool prompts for the ESP chip version; we select ESP32. The tool needs to be configured as shown in the following image.

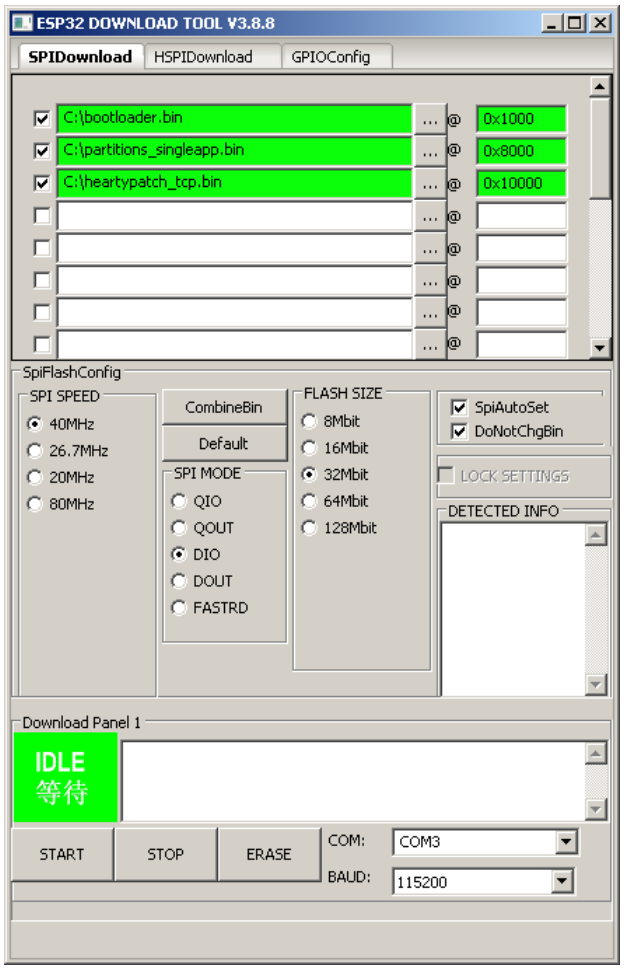


*Figure 2.2.2:* Example configuration for flashing using the Flash Download Tool.

Pressing the “START” button initiates the flashing procedure. The console window (not visible in figure 2.2.2) logs the progress of the whole operation.

To flash the HeartyPatch, we follow these steps,

1. Turn the device off.
2. Press the “START” button on the tool.
3. Provided no errors occurred, a sequence of underscore characters (‘_’) followed by dot characters (‘.’), will be printed character-by-character periodically to the console window. Turn on the device just before the sequence begins, in wit, just before the first underscore in a sequence is printed. If the deadline is missed, turn off the device and try again.
4. Wait for the flash to complete.

## 2.3 Firmware analysis

The C source code for the Wi-Fi and TCP/IP version of the firmware is available at https://github.com/patchinc/heartypatch/tree/master/firmware/heartypatch-stream-tcp.

The core files are:

- **kalam32_tcp_server.c** TCP/IP server code.
- **main.c** general initialization code.
- **max30003.c** interface to the MAX30003 chip.
- **packet_format.h** various constants related to the packet structure.

By convention, we refer to a procedure within a specific file by using: `<file>::<procedure>`, and where a procedure is already introduced or its location obvious by the context we use `::<procedure>`. Similarly, we use `<file>::<line>` to refer to a line within a file. Where it facilitates reading, code listings will be explained in a bullet list where each item references a single or a range of lines, e.g.

- 45: <commentary for line 45>
- 78 – 96: <commentary for lines 78 to 96>

## 2.3.1 Initialization

The entry point is `main.c::app_main`,

```
 99  void app_main(void)
100  {
101      nvs_flash_init();
102
103      max30003_initchip(PIN_SPI_MISO,PIN_SPI_MOSI,PIN_SPI_SCK,PIN_SPI_CS);
104      vTaskDelay(2/ portTICK_PERIOD_MS);
105
106      kalam_wifi_init();
107
108      /* Wait for WiFI to show as connected */
109      xEventGroupWaitBits(wifi_event_group, CONNECTED_BIT_kalam,false, true, portMAX_DELAY);
110
111  #ifdef CONFIG_TCP_ENABLE                    //enable it from makemenuconfig
112      kalam_tcp_start();
113  #endif
114
115  }
```

First, ESP's *Non-Volatile Storage* (NVS) subsystem is initialized. It provides a mechanism for storing/accessing key-value pairs in flash memory. Even though not directly used by the firmware, it is used behind the scenes by the Wi-Fi driver and PHY subsystem to store configuration data.

Next, the MAX30003 chip is initialized. We examine details of how this is done at a later time.

Interspersed throughout the code are various `vTaskDelay` calls which cause the calling task to yield its execution for a given amount of time. This is typically used after interacting with the MAX30003 chip, presumably, to allow for some extra time to process the request. On other occasions, it is used just to stall the execution, for example, if polling for some value. Use of a delay is first encountered at line 104.

Afterward, the Wi-Fi driver is initialized by a call to `::kalam_wifi_init`,

```
72  void kalam_wifi_init(void)
73  {
74        tcpip_adapter_init();
75        wifi_event_group = xEventGroupCreate();
76        ESP_ERROR_CHECK( esp_event_loop_init(event_handler, NULL) );
77        wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
78        ESP_ERROR_CHECK( esp_wifi_init(&cfg) );
79        ESP_ERROR_CHECK( esp_wifi_set_storage(WIFI_STORAGE_RAM) );
80        wifi_config_t wifi_config = {
81            .sta = {
82                .ssid = CONFIG_WIFI_SSID,
83                .password = CONFIG_WIFI_PASSWORD,
84            },
85        };
86        ESP_LOGI(TAG, "Setting WiFi configuration SSID %s...", wifi_config.sta.ssid);
87        ESP_ERROR_CHECK( esp_wifi_set_mode(WIFI_MODE_STA) );
88        ESP_ERROR_CHECK( esp_wifi_set_config(WIFI_IF_STA, &wifi_config) );
89        ESP_ERROR_CHECK( esp_wifi_start() );
90
91  #ifdef CONFIG_MDNS_ENABLE
92        esp_err_t err = mdns_init(TCPIP_ADAPTER_IF_STA, &mdns);
93        ESP_ERROR_CHECK( mdns_set_hostname(mdns, KALAM32_MDNS_NAME) );
94        ESP_ERROR_CHECK( mdns_set_instance(mdns, KALAM32_MDNS_NAME) );
95  #endif
96
97  }
```

- 74: ESP's TCP/IP network stack is initialized.
- 75: A FreeRTOS event group is created that will be used to notify `::app_main` whether the network adapter was assigned an IP address.
- 76: A network event handler is registered. In this case, a networking event may be posted from the Wi-Fi driver or TCP/IP stack (ESP-IDF uses the lwIP[4] lightweight implementation).
- 77 – 89: The Wi-Fi system is initialized, configured to connect to the SSID defined in the firmware build script, and started.
- 92 – 94: Optionally, the mDNS service is initialized. This service associates a name with the device's IP for ease of access[5].

An important step in this code is the registration of the event handler at line 76,

[4] https://www.nongnu.org/lwip

[5] https://en.wikipedia.org/wiki/Multicast_DNS

```
53  static esp_err_t event_handler(void *ctx, system_event_t *event)
54  {
55      switch(event->event_id) {
56      case SYSTEM_EVENT_STA_START:
57          esp_wifi_connect();
58          break;
59      case SYSTEM_EVENT_STA_GOT_IP:
60          xEventGroupSetBits(wifi_event_group, CONNECTED_BIT_kalam);
61          break;
62      case SYSTEM_EVENT_STA_DISCONNECTED:
63          esp_wifi_connect();
64          xEventGroupClearBits(wifi_event_group, CONNECTED_BIT_kalam);
65          break;
66      default:
67          break;
68      }
69      return ESP_OK;
70  }
```

Its purpose is a) to connect the Wi-Fi station to the configured SSID once the Wi-Fi system is initialized, and b) to automatically attempt to reconnect if disconnected. Moreover, the connected event used by `::app_main` is set/reset accordingly.

Back at `main.c::109`, the program suspends execution indefinitely waiting for the connected event, before starting the TCP/IP server by calling `kalam32_tcp_server.c::kalam_tcp_start`,

```
224  void kalam_tcp_start(void)
225  {
226   xTaskCreate(&tcp_conn, "tcp_conn", 4096, NULL, 5, NULL);
227  }
```

This will spawn a new task with entry point `kalam32_tcp_server.c::tcp_conn`. `4096` is the stack space allocated for the task in words and `5` is the priority of the task. The lower the priority value, the lower the priority of the task, with `0` being the lowest and `configMAX_PRIORITIES - 1` being the highest (`configMAX_PRIORITIES` is defined in FreeRTOSConfig.h).

Note that `::app_main` runs in the context of a FreeRTOS task (that was spawned by the ESP-IDF), but unlike other tasks, it is allowed to return. Thus, after calling `::kalam_tcp_start` at line 112 and returning, the system will delete the task that ran `::app_main` and continue running any other tasks normally.

## 2.3.2 The TCP/IP server

The server's task entry point is `kalam32_tcp_server.c::tcp_conn`,

```c
void tcp_conn(void *pvParameters)
{
    ESP_LOGI(TAG, "task tcp_conn start.");

    /*create tcp socket*/
    int socket_ret;
    TaskHandle_t tx_rx_task;
    vTaskDelay(2000 / portTICK_RATE_MS);
    while (1) {
        ESP_LOGI(TAG, "create_tcp_server.");
        socket_ret=create_tcp_server();
        if(ESP_FAIL == socket_ret)
        {
            ESP_LOGI(TAG, "create tcp socket error,stop.");
            vTaskDelete(NULL);
        }
        /*create a task to tx/rx data*/
        xTaskCreate(&send_data, "send_data", 4096, NULL, 4, &tx_rx_task);
        int flag = true;
        while (flag)
        {
            vTaskDelay(3000 / portTICK_RATE_MS);//every 3s
            int err_ret = check_socket_error_code();
            if (err_ret == ECONNRESET)
            {
                ESP_LOGI(TAG, "disconnected... stop.");
              close_socket();
              flag = false;
            }
        }
        vTaskDelete(&tx_rx_task);
        max30003_sw_reset();
        ESP_LOGI(TAG, "restart");
        flag = true;
    }

    close_socket();
    vTaskDelete(tx_rx_task);
    vTaskDelete(NULL);
}
```

This is the “control” loop of the server, where it services one client at a time. A call to `kalam32_tcp_server.c::create_tcp_server` will create a TCP/IP socket to listen for incoming connections and once a connection is accepted, the MAX30003 is prepared for ECG recording. Next, the task responsible for sending the data is created (with priority lower than that of `::tcp_conn`), and the program enters a control loop for managing the task: it polls the status of the connected socket every 3 seconds, and if the connection was reset, deletes the `send_data` task, resets the MAX30003 chip, and goes back to listening for new connections.

The `send_data` task is a simple loop where a packet is repeatedly constructed and sent,

```
55  static void send_data(void *pvParameters)
56  {
57      uint8_t* db;
58      uint8_t databuff[DEFAULT_PKTSIZE];
59      memset(databuff, PACK_BYTE_IS, DEFAULT_PKTSIZE);
60      db=databuff;
61      vTaskDelay(100/portTICK_RATE_MS);
62      ESP_LOGI(TAG, "start sending...");
63
64      while(1)
65      {
66          db = max30003_read_send_data();
67           //send function
68          if (db != NULL)
69              send(connect_socket, db, PACKET_SIZE, 0);
70          vTaskDelay(2/portTICK_RATE_MS);
71      }
72  }
```

Lines 58 – 61 are completely useless and can be ignored. The observant reader will have no doubt already noticed that the source code is of subpar quality, and we will refrain from pointing out such code again.

The call to `max30003.c::max30003_read_send_data` at line 66, constructs a packet and returns a pointer to it. The data are sent via wifi at line 69 provided that the construction of the packet succeeded. Reasons for failure are a) the ECG queue was empty, and b) the ECG queue was not read fast enough and therefore, has overflowed. We postpone details on `::max30003_read_send_data` until a later time.

## 2.3.3 Interfacing with MAX30003

Back at `main.c::103`,

```
103     max30003_initchip(PIN_SPI_MISO,PIN_SPI_MOSI,PIN_SPI_SCK,PIN_SPI_CS);
```

there is a call to `max30003.c::max30003_initchip` which lays the groundwork for communicating with the MAX30003 chip and initializes it for recording ECG and R-to-R delay intervals.

```
200  void max30003_initchip(int pin_miso, int pin_mosi, int pin_sck, int pin_cs )
201  {
202      esp_err_t ret;
203
204      spi_bus_config_t buscfg=
205      {
206          .miso_io_num=pin_miso,
207          .mosi_io_num=pin_mosi,
208          .sclk_io_num=pin_sck,
209          .quadwp_io_num=-1,
210          .quadhd_io_num=-1
211      };
212
213      spi_device_interface_config_t devcfg=
214      {
215          .clock_speed_hz=4000000,                 //Clock out at 10 MHz
216          .mode=0,                                 //SPI mode 0
217          .spics_io_num=pin_cs,                    //CS pin
218          .queue_size=7,                          //We want to be able to queue 7 transactions at a time
219          .pre_cb=max30003_spi_pre_transfer_callback,  //Specify pre-transfer callback to handle D/C line
220      };
221      //Initialize the SPI bus
222      ret=spi_bus_initialize(HSPI_HOST, &buscfg, 0);             //use 1 instead of 0 to enable dma
223      assert(ret==ESP_OK);
224      //Attach the LCD to the SPI bus
225      ret=spi_bus_add_device(HSPI_HOST, &devcfg, &spi);
226      assert(ret==ESP_OK);
227      max30003_start_timer();
228      vTaskDelay(100 / portTICK_PERIOD_MS);
229
```

The first part of this procedure initializes the SPI bus and gets a handle that will be used to communicate with the chip. The *Serial Peripheral Interface* (SPI) is a synchronous serial communication interface specification used for efficient short-distance communications, primarily in embedded systems.

Also, ESP's *LED Control* (LEDC) peripheral is initialized and configured with a call to `::max30003_start_timer`,

```
52   void max30003_start_timer(void)
53   {
54       ledc_timer_config_t ledc_timer =
55       {
56         .bit_num = LEDC_TIMER_10_BIT, //set timer counter bit number
57         .freq_hz = 32768 ,              //set frequency of pwm
58         .speed_mode = LEDC_HIGH_SPEED_MODE,   //timer mode,
59         .timer_num = 0    //timer index
60       };
61
62       ledc_timer_config(&ledc_timer);
63
64       ledc_channel_config_t ledc_channel =
65       {
66           .channel = LEDC_CHANNEL_0,
67           .duty = 512,
68           .gpio_num = PIN_NUM_FCLK,
69           .intr_type = LEDC_INTR_DISABLE,
70           .speed_mode = LEDC_HIGH_SPEED_MODE,
71           .timer_sel = LEDC_TIMER_0
72       };
73
74       ledc_channel_config(&ledc_channel);
75   }
```

The LEDC is primarily designed to control the intensity of Light Emitting Diodes (LEDs), although it can also be used to generate *Pulse-Width Modulated*[6] (PWM) signals. In this case, it is used to generate a 32.768kHz PWM signal that is required by MAX30003.

Back at `max30003.c::230` with the SPI handle at hand, the next step is to initialize the chip.

```
230     max30003_sw_reset();
231     vTaskDelay(100 / portTICK_PERIOD_MS);
232
233     MAX30003_Reg_Write(CNFG_GEN, 0x080004);
234     vTaskDelay(100 / portTICK_PERIOD_MS);
235
236     MAX30003_Reg_Write(CNFG_CAL, 0x720000);  // 0x700000
237     vTaskDelay(100 / portTICK_PERIOD_MS);
238
239     MAX30003_Reg_Write(CNFG_EMUX,0x0B0000);
240     vTaskDelay(100 / portTICK_PERIOD_MS);
241
```

We will peer into the details of communicating with MAX30003 later on, but for the time being it suffices to know that data is sent using `::MAX30003_Reg_Write`. The first argument is the register address and the second the data to send.

- 230: The chip is reset; equivalent to power cycling it. `::max30003_sw_reset` internally just calls `::MAX30003_Reg_Write`.
- 233: Alters the general configuration register to enable the ECG channel. Also keeps the default master clock frequency setting of 32768Hz.
- 236: Configures the internal calibration voltage sources.
- 239: Configures the ECG channel's input multiplexer.

Next comes the settings for recording ECG and R-to-R delay intervals,

[6] https://en.wikipedia.org/wiki/Pulse-width_modulation

```
242     unsigned long ecg_config = 0x001000;    // was 0x005000
243 #ifdef CONFIG_SPS_128
244     ecg_config = ecg_config | 0x800000;   //  d[23:22] -- RATE[0:1]: 10 for 128sps
245         ESP_LOGI(TAG, "max30003_initchip setting SPS to 128");
246 #endif
247 #ifdef CONFIG_SPS_256
248     ecg_config = ecg_config | 0x400000;   //  d[23:22] -- RATE[0:1]: 01 for 256 sps
249     ESP_LOGI(TAG, "max30003_initchip setting SPS to 256");
250 #endif
251 #ifdef CONFIG_SPS_512
252     //ecg_config = ecg_config | 0x000000;   //  d[23:22] -- RATE[0:1]: 00 for 512sps
253     ESP_LOGI(TAG, "max30003_initchip setting SPS to 512");
254 #endif
255
256 #ifdef CONFIG_DHPF_ENABLE
257     ecg_config = ecg_config | 0x004000;   //  d[14] = enable 0.5Hz filter
258     ESP_LOGI(TAG, "max30003_initchip DHPF Enabled");
259 #else
260     ESP_LOGI(TAG, "max30003_initchip DHPF Disabled");
261 #endif
262
263
264 //ecg_config = 0x001000;    // TODO: Delete me -- for debug purposes only
265     MAX30003_Reg_Write(CNFG_ECG, ecg_config);
266     vTaskDelay(100 / portTICK_PERIOD_MS);
267
268     MAX30003_Reg_Write(CNFG_RTOR1,0x3fc600);
269     vTaskDelay(100 / portTICK_PERIOD_MS);
270
```

- 242 – 265: The ECG sampling rate and DHPF (Digital High-Pass Filter) settings are set based on the build script choices the user made. Also, the DLPF (Digital Low-Pass Filter) is enabled.
- 268: R-to-R detection is enabled, and some parameters relevant to R-to-R measurements are changed.

And finally,

```
271     unsigned mngr_int = 0x4 | (SAMPLES_PER_PACKET - 1) << 19;
272     MAX30003_Reg_Write(MNGR_INT, mngr_int);
273     vTaskDelay(100 / portTICK_PERIOD_MS);
274
275     MAX30003_Reg_Write(EN_INT,0x000400);
276     vTaskDelay(100 / portTICK_PERIOD_MS);
277
278     max30003_synch();
279     vTaskDelay(100 / portTICK_PERIOD_MS);
280
281     MAX30003_init_sequence();
282 }
```

- 271 – 272: Sets the threshold for unread ECG samples in the ECG queue. If met or surpassed the system will set the STATUS register's EINT field, to indicate that there are samples available. In this case, it is set to 8.
- 275: Instructs the R-to-R detector to set the STATUS register's RRINT field to 1 each time it identifies a new R event.

- 278: Equivalent to a call to `::MAX30003_Reg_Write(SYNCH,0x000000)`. Prepares the chip for ECG recording by resetting the ECG and R-to-R circuitry, clearing the ECG memories and DSP filters.
- 281: Begins a new recording session by clearing the ECG queue, resetting various debugging-related variables, and setting the current packet sequence number to zero.

Now let's have a look at the details of relaying data back and forth to MAX30003. The content of an SPI operation consists of 4 bytes: 1 byte for the command and 3 bytes for the data. The command is further decomposed to 7 bits for the register address and 1 bit to denote a read/write operation.

| Register[31:25] | R/W[24] | Data[23:0] |
|---|---|---|

*Figure 2.3.3.1:* The data of an SPI operation.

Keep in mind that, even though ESP32 is a little-endian chip, all SPI operation content will be transferred (both reads and writes) in big-endian since that is how MAX30003 operates.

`max30003.c::MAX30003_Reg_Write` is used to send data,

```
78  void MAX30003_Reg_Write (unsigned char WRITE_ADDRESS, unsigned long data)
79  {
80    uint8_t wRegName = (WRITE_ADDRESS<<1) | WREG;
81
82    uint8_t txData[4];
83
84    txData[0]=wRegName;
85    txData[1]=(data>>16);
86    txData[2]=(data>>8);
87    txData[3]=(data);
88
89    esp_err_t ret;
90    spi_transaction_t t;
91
92    memset(&t, 0, sizeof(t));       //Zero out the transaction
93
94    t.length=32;                 //Len is in bytes, transaction length is in bits.
95    t.tx_buffer=&txData;               //Data
96    ret=spi_device_transmit(spi, &t);  //Transmit!
97    assert(ret==ESP_OK);            //Should have had no issues.
98  }
```

`txData` is the buffer that contains the SPI content data. Note that it is built in a big-endian fashion. The rest is a straightforward call to `::spi_device_transmit`, which queues the transaction and waits for it to complete.

Data is read using `max30003.c::max30003_reg_read`,

```
128  void max30003_reg_read(unsigned char WRITE_ADDRESS)
129  {
130      uint8_t Reg_address=WRITE_ADDRESS;
131
132      SPI_TX_Buff[0] = (Reg_address<<1 ) | RREG;
133      SPI_TX_Buff[1]=0x00;
134      SPI_TX_Buff[2]=0x00;
135      SPI_TX_Buff[3]=0x00;
136
137      esp_err_t ret;
138      spi_transaction_t t;
139      memset(&t, 0, sizeof(t));       //Zero out the transaction
140
141      t.length=32;
142      t.rxlength=32;
143      t.tx_buffer=&SPI_TX_Buff;
144      t.rx_buffer=&SPI_RX_Buff;
145
146      t.user=(void*)0;
147      ret=spi_device_transmit(spi, &t);
148      assert(ret==ESP_OK);            //Should have had no issues.
149
150      SPI_temp_32b[0] = SPI_RX_Buff[1];
151      SPI_temp_32b[1] = SPI_RX_Buff[2];
152      SPI_temp_32b[2] = SPI_RX_Buff[3];
153
154  }
```

This is similar to `::MAX30003_Reg_Write`, except this time there is no data to send, and an extra buffer (`SPI_TX_Buff`) is specified in the transaction to hold the result. After the transaction completes, the 3 last bytes of the result buffer, are the received register data. The function outputs its result by copying those 3 bytes to a static buffer (still in big-endian).

## 2.3.4 Assembling the packet

Let's see now how `max30003.c::max30003_read_send_data` constructs a packet. The function is logically divided into two parts. The first one decides if it is possible to construct the packet.

```
369  uint8_t* max30003_read_send_data(void)
370  {
371      int size;
372      int ptr;
373
374      max30003_reg_read(STATUS);
375      uint8_t status_bits = (SPI_temp_32b[0] >> 4) & 0xF;
376
377      if ((status_bits & 0x4) == 0x4) {
378          // Reset EOVF condition
379          ESP_LOGI(TAG, "FIFO Reset");
380          max30003_fifo_reset();
381          packet_sequence_id = 0;
382          return NULL;
383      }
384
385      if ((status_bits & 0x8) == 0) {
386          // No data present in FIFO
387          return NULL;
388      }
389
390  #if CONFIG_MAX30003_STATS_ENABLE
391      if (stats_read_count > STATS_INTERVAL) {
392          print_counters();
393          stats_read_count = 0;
394      }
395  #endif
396
```

The STATUS register's EOVF and EINT fields are checked. If EOFV was set (line 377), the ECG queue has overflowed and its data may be corrupted (meaning that the write pointer of the ECG ring buffer caught up with the read pointer). If EINT was not set (line 385), there was no data in the queue. In either case the packet cannot be constructed, and the function returns NULL. If neither was the case, and if conditionally compiled, debugging information related to the data already received is logged.

The second part deals with the actual construction,

```
397         DataPacketHeader[0] = PACKET_SOF1;
398         DataPacketHeader[1] = PACKET_SOF2;
399         DataPacketHeader[2] = PAYLOAD_SIZE_LSB;
400         DataPacketHeader[3] = PAYLOAD_SIZE_MSB;
401         DataPacketHeader[4] = PROTOCOL_VERSION;
402         ptr = HEADER_SIZE;
403
404         size = max30003_include_packet_sequence_id(ptr);
405         ptr += size;
406
407         size = max30003_include_timestamp(ptr);
408         ptr += size;
409
410
411         // Fetch RR interval data
412         size = max30003_read_rtor_data(ptr);
413         ptr += size;
414
415
416         // Fetch ECG data
417         int i;
418         for (i=0; i <SAMPLES_PER_PACKET; i++) {
419             size = max30003_read_ecg_data(ptr);
420             ptr += size;
421         }
422
423         DataPacketHeader[ptr] = 0xF0;   // 0x00;
424         DataPacketHeader[ptr+1] = PACKET_EOF;
425
426         return DataPacketHeader;
427     }
```

The packet is constructed byte-by-byte and placed in `DataPacketHeader` in a little-endian fashion.

- 397 – 398: Start of frame; a constant to denote the start of the packet.
- 399 – 400: The size of the payload in bytes. The payload consists of the sequence number, the timestamp, the R-to-R value, and the ECG sample data.
- 401: Protocol version number.
- 404: Packet sequence number. This is just a serial number increased with each new packet.
- 407: Timestamp. This is the system uptime.
- 412: R-to-R value. The measured R-to-R value in milliseconds.
- 417 – 421: ECG sample data.
- 423 – 424: End of frame; a constant to indicate the end of the packet.

This amounts to `PACKET_SIZE` bytes in total (as defined in packet_format.h) which equals 55 bytes.

The R-to-R value is included in the packet using `::max30003_read_rtor_data`,

```
312  int max30003_read_rtor_data(int ptr)
313  {
314      max30003_reg_read(RTOR);
315      unsigned long RTOR_msb = (unsigned long) (SPI_temp_32b[0]);
316      unsigned char RTOR_lsb = (unsigned char) (SPI_temp_32b[1]);
317      unsigned long rtor = (RTOR_msb<<8 | RTOR_lsb);
318      rtor = ((rtor >>2) & 0x3fff) ;
319      unsigned int RR = (unsigned int)rtor*8 ;  //8ms
320
321      DataPacketHeader[ptr] = RR;
322      DataPacketHeader[ptr+1] = RR>>8;
323      DataPacketHeader[ptr+2] = 0x00;
324      DataPacketHeader[ptr+3] = 0x00;
325
326      /*
327      float hr =  60 /((float)rtor*0.008);
328      unsigned int HR = (unsigned int)hr;  // type cast to int
329      DataPacketHeader[ptr+4] = HR;
330      DataPacketHeader[ptr+5] = HR>>8;
331      DataPacketHeader[ptr+6] = 0x00;
332      DataPacketHeader[ptr+7] = 0x00;
333      */
334
335      return RR_SIZE;
336  }
```

First, the R-to-R register (Fig. 2.3.4.1) is read from MAX30003 with a call to `::max30003_reg_read`, the result of which is stored in `SPI_temp_32b`. The actual value is contained in the upper 14 bits.

| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RTOR Interval Timing Data [13:0] | | | | | | | | | | | | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

*Figure 2.3.4.1:* The R-to-R register. [3]

The code between lines 315 and 318 extracts those bits from the big-endian buffer. Next, the bits need to be multiplied by the resolution of the R-to-R detector which depends on the master clock frequency, as can be seen in the next table,

| FMSTR [1:0] | MASTER FREQUENCY ($f_{MSTR}$) (Hz) | ECG DATA RATES & RELATED TIMING (RATE SELECTIONS) | | CALIBRATION TIMING RESOLUTION (CAL_RES) (µs) |
|---|---|---|---|---|
| | | ECG | RTOR | |
| | | DATA RATE (sps) | TIMING RESOLUTION (RTOR_RES) (ms) | |
| 00 | 32768 | 00 = 512<br>01 = 256<br>10 = 128 | 7.8125 | 30.52 |
| 01 | 32000 | 00 = 500<br>01 = 250<br>10 = 125 | 8.000 | 31.25 |
| 10 | 32000 | 10 = 200 | 8.000 | 31.25 |
| 11 | 31968.78 | 10 =199.8049 | 8.008 | 31.28 |

*Figure 2.3.4.2:* "Master Frequency Summary Table". [3]

In this case, the correct resolution is 7.8125 ms, even though the code at line 319 uses 8 ms.

An ECG sample is included in the packet with `::max30003_read_ecg_data`,

```
285  int max30003_read_ecg_data(int ptr)
286  {
287        max30003_reg_read(ECG_FIFO);
288
289        unsigned long data0 = (unsigned long) (SPI_temp_32b[0]);
290        data0 = data0 <<24;
291        unsigned long data1 = (unsigned long) (SPI_temp_32b[1]);
292        data1 = data1 <<16;
293        unsigned long data2 = (unsigned long) (SPI_temp_32b[2]);
294        data2 = data2 & 0xc0;
295        data2 = data2 << 8;
296        data = (unsigned long) (data0 | data1 | data2);
297        ecgdata = (signed long) (data);
298
299        DataPacketHeader[ptr] = ecgdata;
300        DataPacketHeader[ptr+1] = ecgdata>>8;
301        DataPacketHeader[ptr+2] = ecgdata>>16;
302        DataPacketHeader[ptr+3] = ecgdata>>24;
303
304        unsigned char ecg_etag = (SPI_temp_32b[2] >> 3) & 0x7;
305        tally_etag[ecg_etag]++;
306        stats_read_count++;
307
308        return SAMPLE_SIZE;
309  }
```

In a similar vein to R-to-R data, the ECG register is read, and relevant bits extracted.

| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ECG Sample Voltage Data [17:0] | | | | | | | | | | | | | | | | | | ETAG [2:0] | | | PTAG [2:0] | | |

*Figure 2.3.4.3:* The ECG register. [3]

One important thing to notice here is that the result will be a 32-bit word that contains the 18 bits of information justified to the left. Consequently, the client parsing the data will have to perform a 14-bit right-shift as an additional step.

The function also extracts the ETAG field of the register and stores it for debugging purposes. This field indicates the status of the read operation, e.g., the queue was empty, the sample read was the last one, etc.

### 2.3.5 Summary

A sketch of the firmware is illustrated in the next figure,

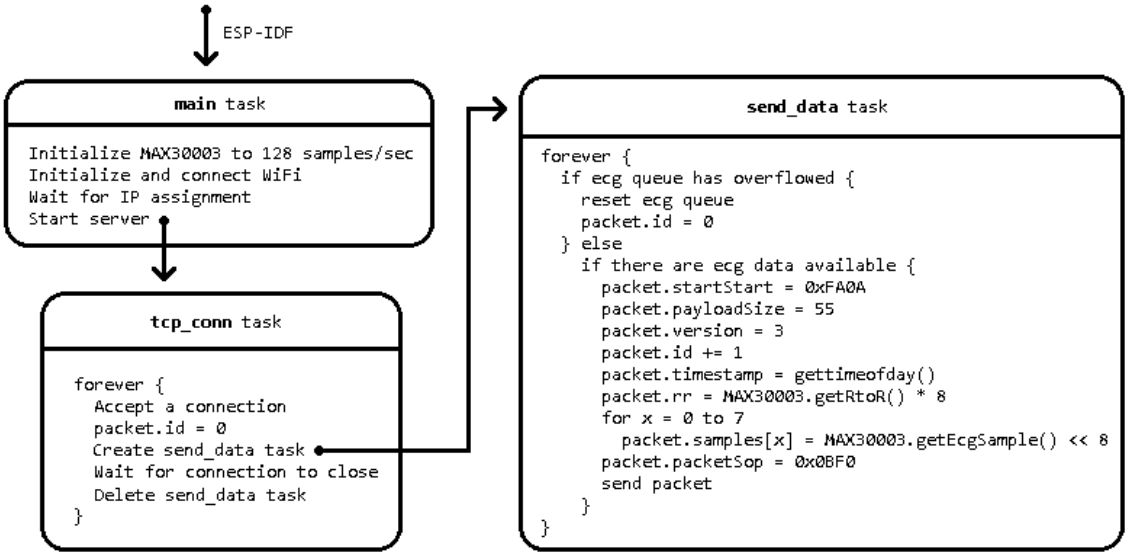


*Figure 2.3.5.1:* Bird's-eye view of the firmware in pseudo-code.

## 2.4 Packet format

Analysis of the firmware revealed the following packet structure, coded in C++,

```
#pragma pack(push, 1)

struct RawPacket {

    static constexpr unsigned SamplesCount = 8; //amount of ECG samples contained in the payload

    uint16_t packetStart, //should be 0xFA0A
             payloadSize; //size of the Payload structure
    uint8_t version; //should be 3

    struct Payload {
        uint32_t id; //serial\sequence number
        timeval timestamp; //timestamp taken while assembling the packet
        uint32_t rr; //computed R-to-R value in milliseconds
        int32_t samples[SamplesCount]; //ECG samples; upper 18bits is the actual sample
    } payload;

    uint16_t packetStop; //should be 0x0BF0

};
```

*Listing 2.4.1:* The packet structure of HeartyPatch on Wi-Fi mode.

Fields do not require further processing except for `samples`, where its elements must be shifted 14 bits to the right.

## 2.5 A sample client application

As part of this thesis, a Windows application was developed to get acquainted with the device. It's called Hearty Patch Stats® and was built in Visual C++ and MFC.

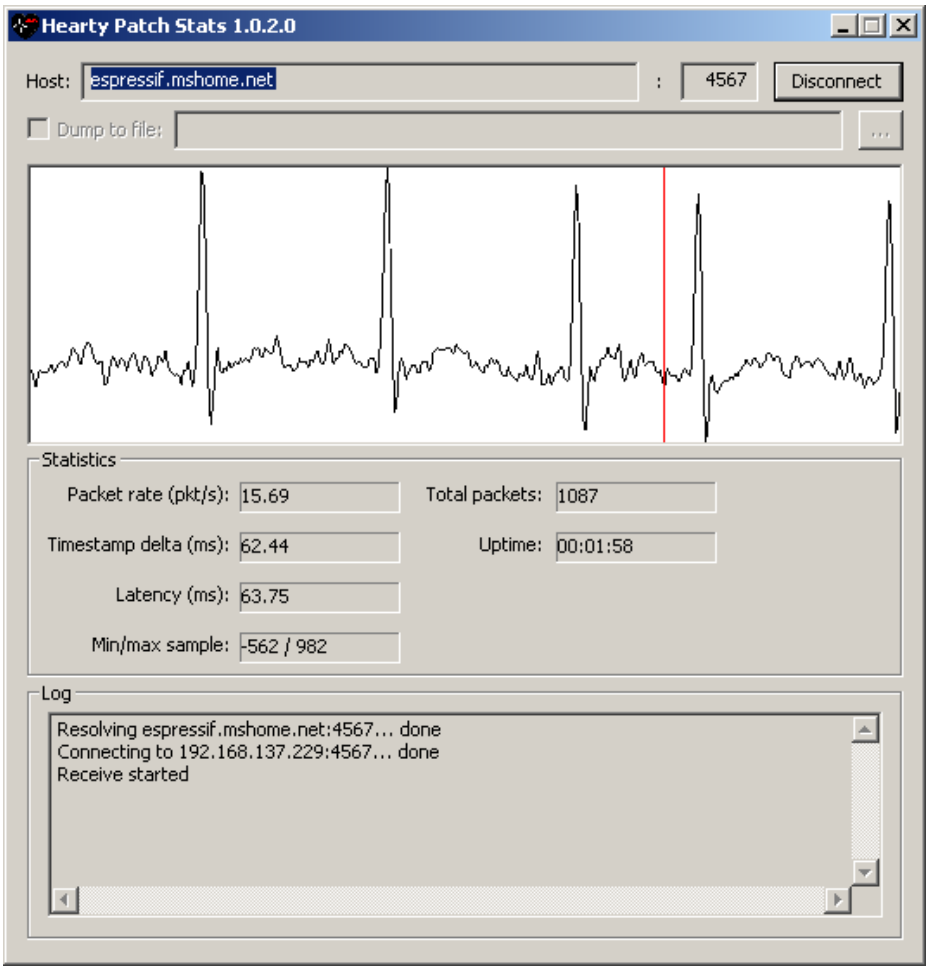


*Figure 2.5.1:* A screenshot of Hearty Patch Stats® in action.

Its primary function is to connect to a HeartyPatch running the Wi-Fi version of the official firmware, a precompiled version that records at 128 samples/sec) and plot incoming ECG samples. It can optionally save incoming packets to a file.

## 2.6 Issues

The use of the program developed in Section 2.5 to test a HeartyPatch device, revealed an unexpected behavior of the firmware, which may prove unacceptable in certain scenarios. To understand the issue, it is imperative to first explain how the firmware reads ECG samples and sends packets. The firmware under question was compiled to record an ECG at a rate of 128 samples/sec.

Before the firmware can read an ECG sample from MAX30003's queue, it must first configure MAX30003 to somehow notify it when there are any samples available. This is done by setting the EFIT

field of MNGR_INT register. The EINT field of the STATUS register will be set every time there are at least an EFIT-number of unread samples in the queue. In this case it is set to 8. Since the sampling rate is set to 128 samples/sec, a new set of 8 samples will be available every $128^{-1} \cdot 8$ sec = 62.5 ms. MAX30003's ECG queue can only hold up to 32 samples, and since the queue is implemented as a ring buffer, if the firmware does not read the queue in a timely fashion, it is possible for the write pointer to go past the read pointer, effectively overwriting older samples. This event is called an overflow event, and the firmware can check for it by testing the EOVF field of the STATUS register.

To send a packet, the firmware does the following in a loop (see Fig. 2.3.5.1): First, it checks for an overflow event. If there is one, clears the ECG queue, and sets the packet serial number back to 0. In this case no packet will be sent as the queue contained invalid data. If no overflow event occurred and there are unread samples available, it proceeds to construct and send a packet. To construct the packet, it increments its serial number by one, and reads 8 ECG samples into it from the queue, among other things. Then the packet is eagerly sent using a blocking send call, before starting all over again. This whole procedure ideally results in a packet rate of $.0625^{-1}$ packets/sec = 16 packets/sec.

The observed issue is that, at seemingly random time intervals, there may be a pause in the incoming packet stream. That is, there may be a delay in receiving the next packet, that can be greater than 62.5 ms. The length of these delays varies, and the exact cause is unknown, but it is known to originate from the send call in the firmware code. That said, because the packet is both constructed and sent from the same task, it may cause an overflow event, which will result in the loss of all recorded samples. The overflow event manifests itself in the packet stream, as a zeroing of the packet serial number.

Had the firmware been compiled to record faster than 128 samples/sec (at 256 or 512 samples/sec), the incidence of such events would be higher, as the tolerance for delaying the reading of the queue would be smaller.

Small modifications that can mitigate the problem (somewhat), include,

- Increasing the EFIT value and the number of samples contained in a packet. This will also slow down the packet rate.
- Disabling Nagle's algorithm[7], since lots of small packets are sent repeatedly.

[7] https://en.wikipedia.org/wiki/Nagle%27s_algorithm

# 3 Testing performance and power usage

We will now test the performance of a software application that analyzes an ECG recording for heart arrhythmias in soft real-time. There will be two runs of tests where the application will receive an ECG, wirelessly via Wi-Fi, analyze it, and display the results on a screen. On the one test, the ECG will be provided by the HeartyPatch and on the other by the Body Gateway. Also, the power usage of both devices will be measured to assess their battery autonomy.

## 3.1 Body Gateway

The *Body Gateway* (BGW) is a medical-grade wearable ECG patch, that is worn on the chest for the acquisition, recording, and transmission of physiological parameters to external devices which can analyze or forward the data to additional storage elements or systems.

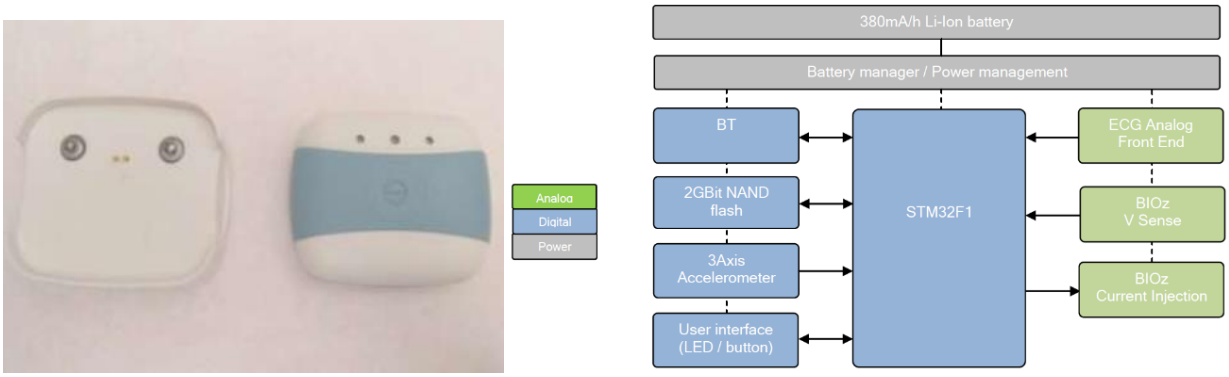


*Figure 3.1.1:* The Body Gateway device and its hardware architecture. [4]

The Body Gateway has several functions, including:

- 1-lead ECG
- HRV
- Breathing rate detection
- Body posture detection
- Physical activity level index

The device is part of a multi-parameter analysis system – the Body Gateway System – and communicates via a Bluetooth radio link with the external device. It is built around a 32-bit STM32F1 series Cortex® microcontroller.

The Body Gateway, developed by STMicroelectronics, is an earlier iteration of a device now called BodyGuardian® Heart[8] by Preventice Solutions. The Body Gateway was chosen to participate in the following tests, solely because it was the only medically certified device of a similar form factor available at the time.

[8] https://www.preventicesolutions.com/healthcare-professionals/body-guardian-heart

Unlike the HeartyPatch, the Body Gateway doesn't have a Wi-Fi radio. Hence, for testing, an intermediate device acting as a Bluetooth-to-Wi-Fi/TCP adapter will be used. That device will be an Odroid XU4 running the "BGW driver" developed by Grammatikakis M. et al. [13].

## 3.2 Server

The server will run the soft real-time arrhythmia classification and visualization application [14], on an Odroid XU3, which is a single-board computer based on ARM's big.LITTLE multicore architecture. The big cluster is the powerful quad-core ARM Cortex A15 clocked from 200 MHz to 2000MHz at intervals of 100 MHz, while the LITTLE cluster is the low-power quad-core Cortex-A7 capable of operating at a cluster-wise frequency of 200 MHz to 1400 MHz at discrete intervals of 100 MHz. Our application runs on two big Cortex A15 cores: 1) Core 0: collecting BGW/HeartyPatch data, and 2) Core 1: analyzing and visualizing the data.

The application relies on two open-source software libraries:

1. *Harvard's Physionet Waveform Database* (WFDB)[9] is used to smooth and standardize BGW's ECG signal to 200 samples/sec according to ANSI/AAMI EC-13.
2. *EP Limited's Open Source ECG Analysis* (OSEA)[10] to perform low/high-pass QRS filtering (via easytest script) for heartbeat detection and classification to normal or abnormal beats. [10, 11] To manage ECG annotation in soft real-time, we extend easytest functionality to avoid re-computation by applying a training signal only on the latest data; our framework theoretically achieves a positive predictivity close to 99.8% when using the MIT/BIH and AHA arrhythmia databases. Other ECG analysis methods result in smaller predictivity rates. [12]

For viewing, annotation, and interactive analysis of ECG waveform in soft real-time (with asynchronous display) we use the Harvard Physionet WAVE[11] software package. It is based on a 32-bit XView open-source toolkit (a low-level XWindows client).

Using our timing infrastructure, we collect measurements each time new ECG data is uploaded to the WAVE tool for visualization (via wave-remote). This includes the number of ECG samples processed, current time, and distribution of ECG analysis latencies to different subprocesses.

For evaluating soft real-time, we share performance statistics, such as the number of samples, latency, throughput, and packet loss, across different application processes using a dynamic shared memory timing infrastructure that supports fast atomic shared memory read/write operations.

[9] https://archive.physionet.org/physiotools/wfdb.shtml
[10] https://www.eplimited.com/confirmation.htm
[11] https://archive.physionet.org/physiotools/wug/wug.pdf

## 3.3 Device configuration & network topology

For ease of reference, a summary of the main features of the ECG devices participating in the tests follows,

| | HeartyPatch | Body Gateway |
|---|---|---|
| **Open-source** | Yes | No |
| **Medical-grade certification** | No | Yes |
| **ECG leads** | 1 | 1 |
| **ECG sampling rate (Hz)** | 125/128/199.8/250/256/500/512 | 128/256 |
| **ECG sample size (bit)** | 18 | 12 |
| **ECG AFE** | MAX30003 | *N/A*[*] |
| **SoC** | ESP32 | STM32F1 |
| **Battery** | 3.7 V, 450 mAh, LiPo | 3.7 V, 380 mAh, LiPo |
| **Communication** | Wi-Fi<br>Bluetooth<br>UART | Bluetooth<br>USART |
| **Sensors** | ECG | ECG<br>Accelerometer<br>Bioimpedance |
| **Functions** | ECG<br>R-to-R<br>HRV[†] | ECG<br>R-to-R<br>HRV<br>Breathing rate<br>Body posture<br>Physical activity index |

(*) Not disclosed.

(†) Non-standard feature; firmware implementation-dependent.

*Table 3.3.1:* Summary of important features of both devices. [2, 3, 4]

For the test, both devices are set to transmit ECG sample data at a sampling rate of 128 samples/sec.

A TP-Link Archer C5400 router will function as the access point for the wireless network required by the tests.

The HeartyPatch will connect to the server as shown in Fig. 3.3.2.


*Figure 3.3.2*: Network topology for the HeartyPatch.

As mentioned in section 3.1, the Body Gateway will require an intermediate device acting as a Bluetooth-to-Wi-Fi/TCP adapter, as shown in Fig, 3.3.3.

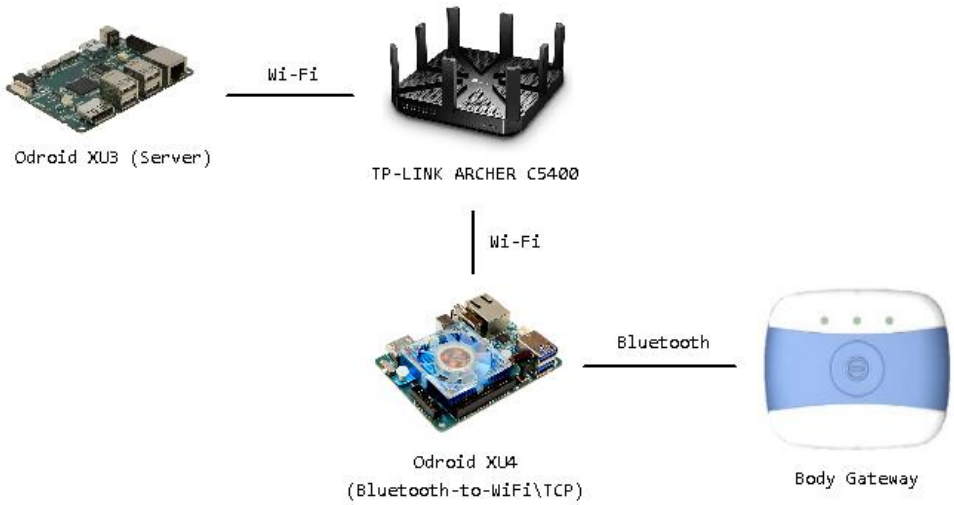


*Figure 3.3.3:* Network topology for the Body Gateway.

## 3.4 Results

Figure 3.4.1 examines the average rate during visualization. Based on the number of samples visualized, and the current timestamp that the data is loaded to WAVE via wave-remote, we compute the average processing rate of ECG data. From this graph, we observe that we can sustain soft real-time for both devices.

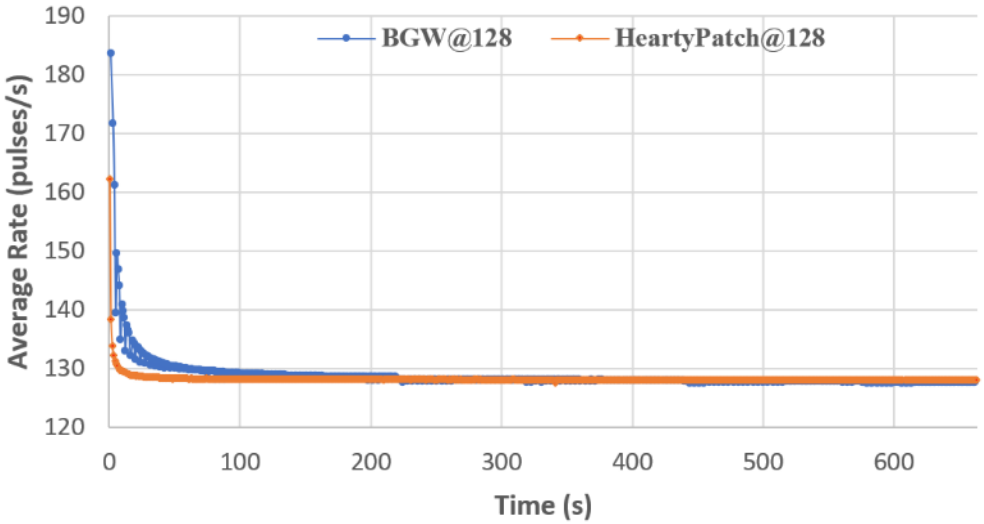


*Figure 3.4.1:* Average processing rate during server visualization.

Focusing on the instant variation of the rate during visualization, Figure 3.4.2 shows a large fluctuation around the average value for BGW, but not for HeartyPatch. This may occur, since BGW transmits ECG samples in bursts (up to 128 samples in a single burst), while the ECG data flow is more regular for HeartyPatch (with a maximum of 8 samples per burst).

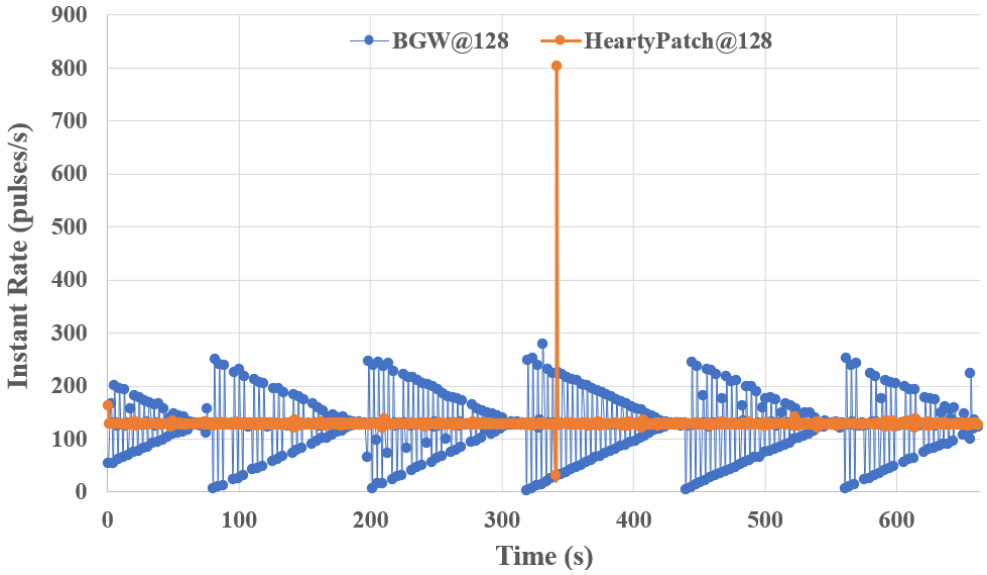


*Figure 3.4.2:* Instant processing rate during server visualization.

Notice that for BGW, missing packet information, e.g., due to missing an RTOS deadline, is subsequently transmitted in the next interval with a special (incomplete) packet. Incomplete packets are ≈ 15% of the complete ones. For HeartyPatch, incomplete packets do not occur. However, it is possible that during some intervals, packets are momentarily delayed, but this is rectified at the subsequent interval; in our experiments, we find that this occurs 2.4% of the time. The peak for HeartyPatch at 340.1 sec is rectified immediately; this rise in the instant rate can be due to a prior system call or interrupt occurring in the Linux kernel at the server.

In Figures 3.4.3 and 3.4.4, we examine the distribution of the different processing delays during animation. The delays are normalized, i.e., they all assume the processing of 128 samples.

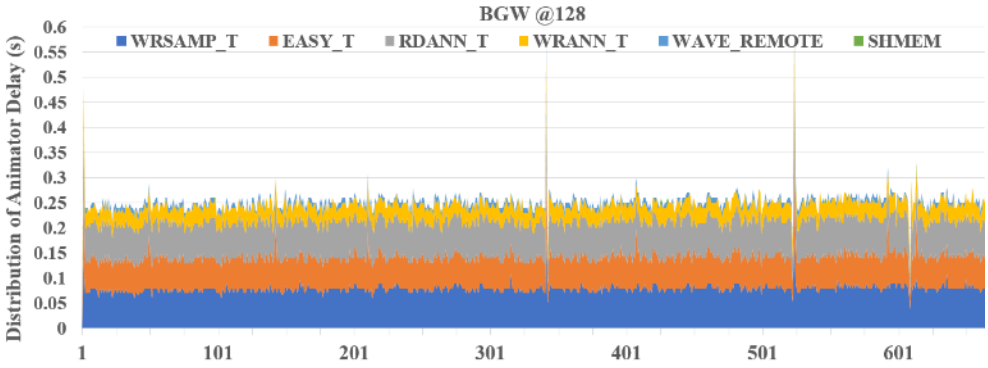


*Figure 3.4.3:* Distribution of animation delays for the Body Gateway.

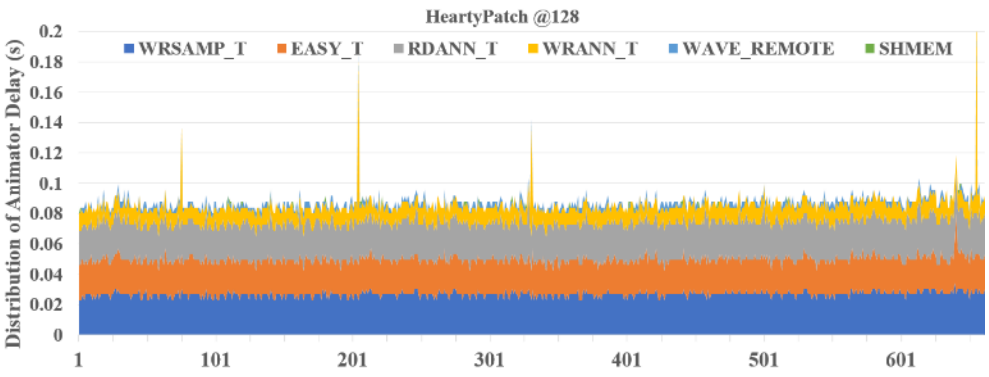


*Figure 3.4.4:* Distribution of animation delays for the HeartyPatch.

As shown, server delays for HeartyPatch are much shorter than those of BGW. The contribution to the different processing delays is similar for both devices. This is mainly due to: a) wrsamp method used for conversion to std EC-13, b) easytest filtering used for heartbeat detection and classification, and c) wrann/rdann used for writing/reading to/from annotation files related to the latest data. Contribution from wave-remote, locking and shared memory constructs are marginal.

Figures 3.4.5 and 3.4.6 show power dissipation during ECG transmission from both devices.

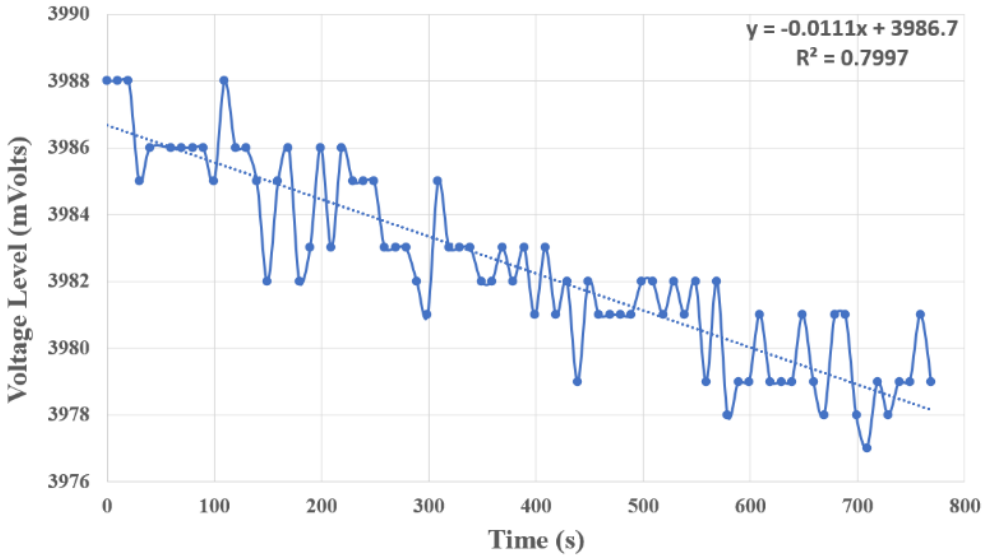


*Figure 3.4.5:* BGW's battery level (in mV) during ECG transmission.

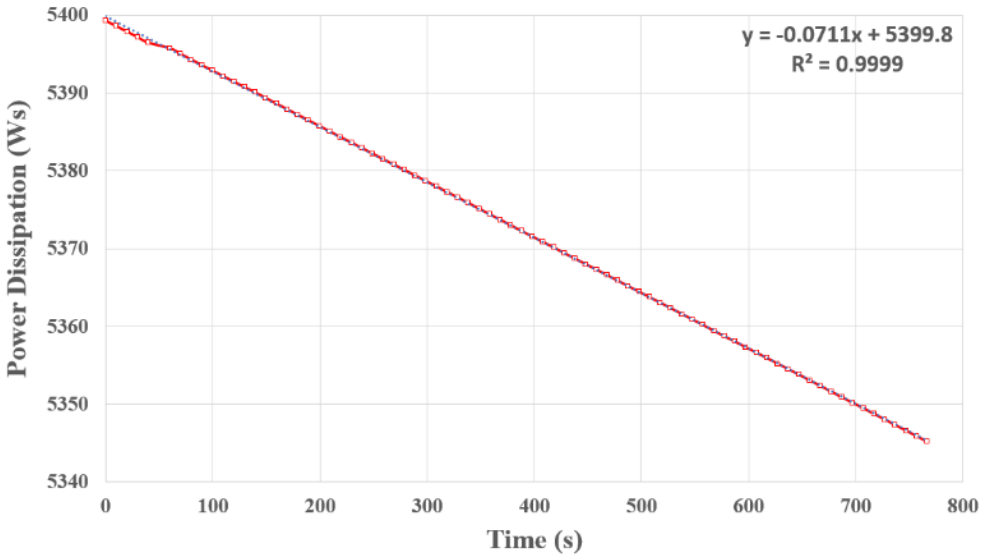


*Figure 3.4.6:* HeartyPatch's power dissipation (in W · secs) during ECG transmission.

For BGW, battery depletion is obtained by programming the sensor to transmit every 10 sec its battery level (along with ECG data). From the graph, and using the trendline model, we deduce that the BGW would be operating until the battery depletes to 3.4V, or, ideally after a maximum of 13.5 hours of ECG transmission.

For HeartyPatch, we have plugged the Wi-Fi enabled Odroid SmartPower 2 energy sensor[12] into the 5V power supply of the board to measure the overall power consumption at a sample rate of 1 Hz (1 sec). SmartPower captures the voltage, current, power, and energy consumed. From the graph, and using the trendline model, we deduce that the HeartyPatch would support 21.1 hours of ECG transmission before the battery power is depleted. In comparison, continuous streaming of all the data (ECG, ACC, respiration, battery, notifications) at the maximum sampling frequency of the BGW reduces the autonomy span to just 3 hours.

Figure 3.4.7 compares the power consumption at the server for BGW and HeartyPatch. Energy consumed at the server is caused by our application (server and animator) running on two ARM Cortex-A15 cores. ARM Cortex-A7 (little cores) and GPU have an insignificant contribution to energy, while memory consumption is very small.

[12] http://odroid.com/dokuwiki/doku.php?id=en:acc:smartpower2

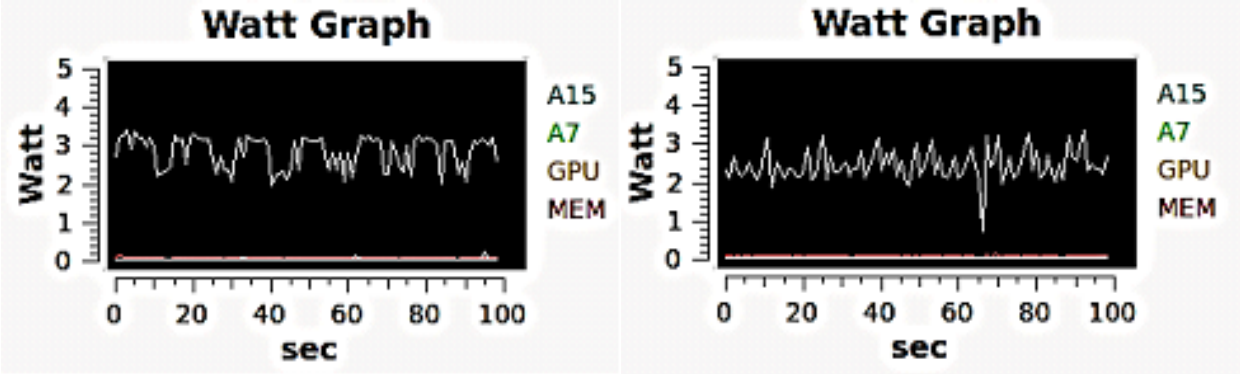


*Figure 3.4.7:* Power consumption at the server: (a) BGW (left), (b) HeartyPatch (right).

According to the watt graphs, the server consumes less energy for HeartyPatch than for BGW (14% smaller average energy dissipation). Instant power consumption is also spikier for HeartyPatch. This may be related to the fact that for BGW, ECG data arrives at the server in large convoys (up to 128 12-bit values), and each one is transferred immediately to preserve real-time. Convoys with HeartyPatch are limited to eight 18-bit values, packed in eight 32-bit unsigned values.

Figures 3.4.8 and 3.4.9 compare the annotated ECG at the server.

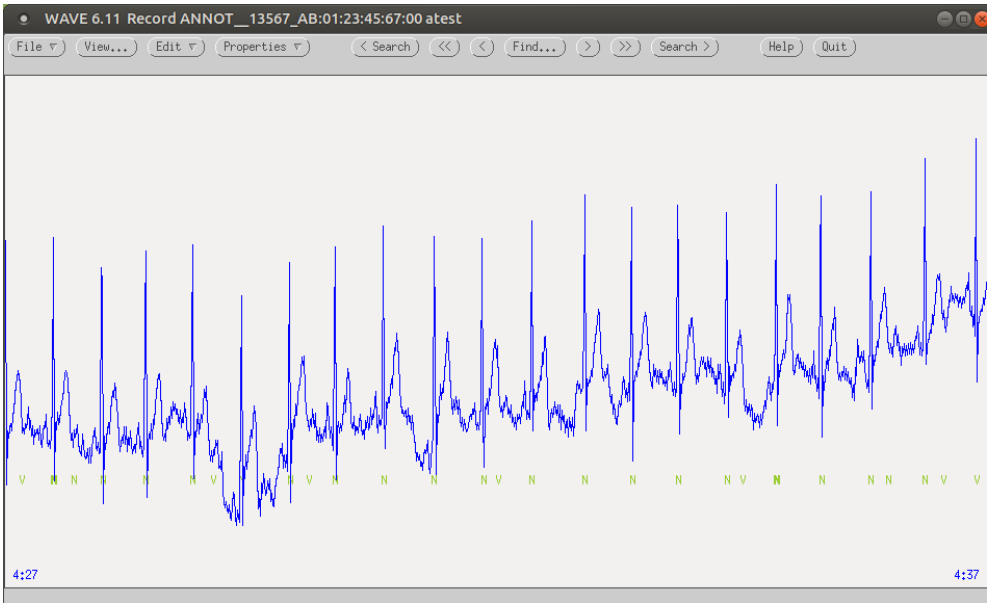


*Figure 3.4.8:* Annotated ECG at the server for Body Gateway.

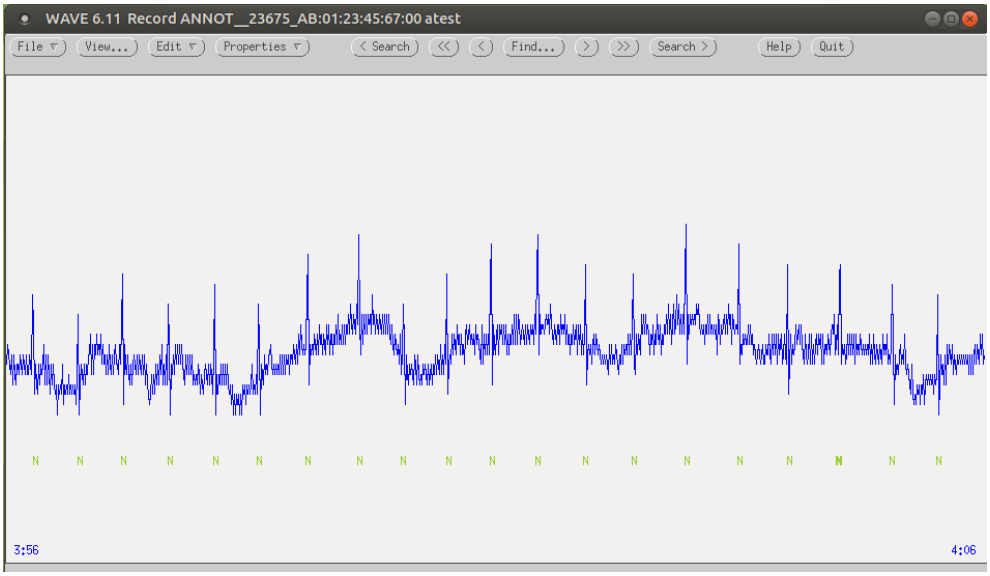


*Figure 3.4.9:* Annotated ECG at the server for HeartyPatch.

Notice that the graph for HeartyPatch has a higher resolution as seen clearly from the images above (spikier). That is because a HeartyPatch ECG value is 18-bit compared to 12-bit precision for BGW. Occasionally, this may cause additional problems during the analysis, e.g., the appearance of false arrhythmia notifications.

# 4 Concluding remarks and future work

In this thesis, we had the opportunity to work with an open-source consumer-grade ECG patch called HeartyPatch. By analyzing its firmware source code, we learned how to communicate with it. This paved the way for developing a program to read the ECG recording it streamed. Using said program, we discovered some issues regarding the performance of its official firmware, that could render the device unusable under certain scenarios.

We also tested the performance of a soft real-time arrhythmia classification and visualization application. We performed two independent series of tests, where the application analyzed the 128 Hz sampled ECG stream it received wirelessly via Wi-Fi, from HeartyPatch or Body Gateway. The distributed application was able to achieve the sought-after soft real-time performance concerning analysis and visualization, with either device connected to it. Overall, the HeartyPatch exhibited slightly better behavior than the Body Gateway.

Unfortunately, the issues we encountered in section 2.6, limit the comparison potential of HeartyPatch with devices that support sampling rates greater than 128 samples/sec. For this reason, we would like to amend (or completely redesign) its software, to support higher sampling rates (up to 512 samples/sec), in a way that avoids these issues.

For scenarios where minor data loss can be tolerated, we would like to experiment with converting the underlying transport protocol to UDP/IP to increase the throughput and/or possibly reduce power usage.

# References


1. HeartyPatch project
   https://heartypatch.protocentral.com/
2. HeartyPatch at Crowd Supply
   https://www.crowdsupply.com/protocentral/heartypatch
3. MAX30003 AFE manual
   https://datasheets.maximintegrated.com/en/ds/MAX30003.pdf
4. Body Gateway datasheet
   https://fccid.io/S9NMHBGW1/User-Manual/Product-literature-1733251
5. Euan A Ashley, Josef Niebauer, “Cardiology Explained”, 2004, Remedica Explained Series, Remedica
   https://www.ncbi.nlm.nih.gov/books/NBK2214/
6. Marcelo Campos, “Heart rate variability: A new way to track well-being”, October 2019, Harvard Health Blog, Harvard Health Publishing
   https://www.health.harvard.edu/blog/heart-rate-variability-new-way-track-well-2017112212789
7. "Heart arrhythmia”, 9 August 2020, Patient Care & Health Information, Mayo Clinic
   https://www.mayoclinic.org/diseases-conditions/heart-arrhythmia/symptoms-causes/syc-20350668
8. Richard N. Fogoros, “What You Should Know About Ambulatory ECG Monitoring”, 31 July 2020, Verywell Health
   https://www.verywellhealth.com/ambulatory-ecg-monitoring-4171275
9. “A Double-Edged Sword: How Over-the-Counter ECG Devices are Impacting Cardiac Care”, 11 March 2020, Diagnostic and Interventional Cardiology
   https://www.dicardiology.com/article/double-edged-sword-how-over-counter-ecg-devices-are-impacting-cardiac-care
10. Hamilton P. S., Patrick S., Tompkins W. J., “Quantitative investigation of QRS detection rules using the MIT/BIH arrhythmia database”, 1986, IEEE Trans. Biomed. Eng. 12, p. 1157–1165
11. Tompkins W. J., “A real-time QRS detection algorithm”, 1985, IEEE Trans. Biomed. Eng. 3, p. 230–236
12. Pinto J. R., Cardoso J. S., Lourenço A., “Evolution, current challenges, and future possibilities in ECG biometrics”, 2018, IEEE Access 6, p. 4746–4776
13. Grammatikakis M. D., Koumarelis A., Mouzakitis A., “Software Architecture of a User-Level GNU/Linux Driver for a Complex E-Health Biosensor”, In: Saponara S., De Gloria A. (eds) Applications in Electronics Pervading Industry, Environment and Society (ApplePies 2020). In: “Lecture Notes in Electrical Engineering”, Springer, Cham, 2021, vol 738, 1—7
14. Grammatikakis M. D., Koumarelis A., Ntallaris E. “Validation of Soft Real-Time in Remote ECG Analysis”, 2021, In: Saponara S., De Gloria A. (eds) Applications in Electronics Pervading Industry, Environment and Society. ApplePies 2020. In Lecture Notes in Electrical Engineering, Springer, Cham, 2021, vol 738, 90—96