NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

HUGO SANTIAGO BRANDÃO DE ASSUNÇÃO

PORTABLE DIAGNOSE DEVICE USING NAAT TEST

CREATION AND CALIBRATION OF A SYSTEM NECESSARY FOR THIS PURPOSE

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
september, 2022

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

PORTABLE DIAGNOSE DEVICE USING NAAT TEST

CREATION AND CALIBRATION OF A SYSTEM NECESSARY FOR THIS PURPOSE

HUGO SANTIAGO BRANDÃO DE ASSUNÇÃO

Adviser: Orfeu Flores
CEO, STAB VIDA

Co-adviser: Carmen Pires Morgado
Professora Auxiliar, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
september, 2022

Portable diagnose device using NAAT test

Dedico o meu trabalho à minha família e namorada pelo apoio e paciência que me deram ao longo dos 5 anos de faculdade. Obrigado.

Acknowledgements

Quero começar por agradecer ao CEO da STAB VIDA Orfeu Flores, pela oportunidade e confiança que depositou nas minhas capacidades. Obrigado pelo convite para integrar a equipa da STAB VIDA que possibilitou o meu desenvolvimento pessoal e profissional ao longo do ano que estive na empresa.

Agradeço ao Doutor Gonçalo Dória, pela orientação e disponibilidade que demonstrou, as suas sugestões e conhecimentos facilitaram o desenvolvimento do trabalho e os desafios propostos que me obrigaram a pensar e encontrar soluções ajudaram a obter um resultado final do qual me orgulho.

Quero também agradecer ao António Santos, Tiago Furtado, Pedro Dionísio e Mariana Conceição e a toda a equipa da STAB VIDA pela disponibilidade e ajuda prestada ao longo da tese. O vosso apoio e colaboração ajudou a elevar o patamar de qualidade do trabalho desenvolvido.

Um grande obrigado à professora Carmen Morgado por estar sempre disponível para esclarecer qualquer dúvida relativa à escrita do documento. Os seus comentários, sugestões e revisões foram cruciais para o melhoramento deste trabalho.

Aos meus pais, Sérgio e Dionísia, obrigado pela educação de que me orgulho! Os vossos conselhos ajudaram-me a focar e a tomar as decisões corretas e é por isso que cheguei até este ponto. Obrigado pelo apoio e motivação que me deram para atingir os meus objetivos.

Quero agradecer ao Belmiro pelo apoio que me deu durante a faculdade, foste sempre simpático e tiveste sempre disponibilidade para ajudar em qualquer problema ou esclarecer qualquer dúvida que eu tivesse.

O agradecimento mais especial e profundo para a minha namorada, Ana Catarina, pelo amor e apoio que me deu desde o primeiro dia. Sem ti não tinha chegado onde cheguei e não seria a pessoa que sou. Obrigado por seres a pessoa mais incrível e simpática que conheço!

Ao meu amigo, Nuno Santos obrigado por teres sido o meu parceiro de projetos durante os 5 anos de faculdade. A tua inteligência, simpatia e boa disposição tornaram melhor os momentos mais difíceis.

Agradeço a todos os meus amigos, sem vocês o meu percurso académico não teria sido o mesmo. Obrigado por alegrarem-me e darem-me imensos momentos agradáveis que vou guardar na minha memória. Desejo que tenham muito sucesso.
Por fim a todas as pessoas que ficaram por mencionar, obrigado!

Abstract

Traditional methods of analyzing medical samples require expensive tools handled by highly skilled staff in a laboratory environment. A sample analysis case that proved to be relevant with the recent COVID-19 pandemic is that of the Loop-mediated Isothermal Amplification (LAMP), due to the conditions necessary to perform this type of test in the case of this disease, which only requires that a temperature be maintained stably over a period of time.

Currently, there are already devices capable of reproducing this process in a compact and mobile way. But if we bring into the equation LAMP tests that require variable temperatures and accurate sensor readings, the necessary hardware and firmware increase in complexity and development difficulty.

In this work, a device produced by STAB VIDA, dedicated to the analysis of samples for the COVID-19 test, was used as a basis to build a new device capable of executing multiple new LAMP tests, which requires, among other functionalities, more complex temperature control. This way, there was no need to start from scratch; however, several aspects of the initial device had to be rethought.

This new device, with the aforementioned capabilities, allows a single healthcare professional to perform the intended tests with any pre-defined protocol they choose, saving them the work of handling medical equipment that would otherwise be required. It also allows the professional to perform the test wherever he is, not limited to the location of the laboratory. That said, the results would never have the same accuracy as in the laboratory, but a high enough degree of correctness usually suffices for the purposes the device is built for.

Keywords: device, temperature, medical equipment, assay, hardware, firmware, laboratory

Resumo

Métodos tradicionais de análise de amostras médicas exigem ferramentas dispendiosas, manuseadas por funcionários altamente especializados num ambiente laboratorial. Um caso de análise de amostras que se revelou relevante com a recente pandemia de COVID-19 é o do LAMP, devido às condições necessárias para executar este tipo de teste no caso desta doença, que somente obriga a que uma temperatura seja mantida de forma estável durante um período de tempo.

Atualmente, já existem dispositivos capazes de reproduzir esse processo de forma compacta e móvel. Mas se trouxermos para a equação ensaios LAMP que exijam temperaturas variáveis e leituras precisas do sensor, o *hardware* e o *firmware* necessários aumentam em complexidade e dificuldade de desenvolvimento.

Neste trabalho utilizou-se como base um dispositivo produzido pela STAB VIDA, dedicado à análise de amostras para o teste de COVID-19, para construir um novo dispositivo capaz de executar múltiplos novos ensaios LAMP que requer, entre outras funcionalidades, um controlo de temperatura mais complexo. Desta forma não houve necessidade de partir do zero, no entanto, vários aspetos do dispositivo inicial tiveram de ser repensados.

Este novo dispositivo, com as capacidades mencionadas anteriormente, permite que um único profissional de saúde execute os testes pretendidos com qualquer protocolo predefinido que este escolher, poupando o trabalho de manuseio de equipamentos médicos que de outra forma seriam necessários. Também possibilita ao profissional realizar o ensaio onde quer que esteja, não se limitando à localização do laboratório. Dito isto, os resultados nunca teriam a mesma precisão dos de laboratório, mas uma precisão alta o suficiente geralmente é aceitável para os propósitos para os quais o dispositivo foi construído.

Palavras-chave: dispositivo, temperatura, equipamento médico, ensaio, *hardware*, *firmware*, laboratório

Contents

List of Figures

List of Tables

Acronyms

ATT Attribute Protocol 10, 11

BLE Bluetooth Low Energy 9, 12, 13, 35, 53, 71, 74

DVP DoctorVida Pocket 1, 2, 6, 10, 12, 13, 14, 16, 17, 31, 34, 35, 44, 45, 47, 53, 55, 57, 60, 61, 63, 68, 69, 70, 73, 84, 85, 95, 103, 104, 106

GATT Generic Attribute Profile 10, 11, 12, 74

L2CAP Logical Link Control and Adaptation Protocol 10, 11

LAMP Loop-mediated Isothermal Amplification vii, viii, 2, 3, 4, 5, 7, 38, 42

MTU Maximum Transmission Unit 10, 54

NAAT Nucleic Acid Amplification Test 1, 2, 3

P Proportional 29

PCR Polymerase Chain Reaction 1, 2, 3, 5

PD Proportional Derivative 26, 27, 28

PDU Protocol Data Unit 10

PI Proportional Integral 26, 28, 29, 30, 50

PID Proportional Integral Derivative xiv, xvii, 16, 21, 22, 24, 25, 26, 27, 28, 29, 31, 32, 33, 39, 41, 42, 45, 46, 47, 49, 50, 51, 52, 53, 68, 104

RT-LAMP Reverse Transcription Loop-mediated Isothermal Amplification 2, 3

SMP Security Manager Protocol 10

UUID Universally Unique Identifier 11

1 Introduction

Medical testing has many difficulties, including the machinery that is expensive and requires qualified personnel to operate. This thesis is about the development of a device that is capable of reproducing, to some extent, a testing machine's capabilities in a small, portable manner.

There was already in production a device capable of producing the temperature required for an isothermal Nucleic Acid Amplification Test (NAAT)[1], but variable temperatures were desired. Therefore, temperature control mechanisms is going to be explored in this thesis. Besides the temperature control component, there are multiple parts that together allow a user to operate the device, and the development of these will be presented in future sections.

1.1 Context & Motivation

In the years of 2020 and 2021, the world has been afflicted by a pandemic. As of the time of writing this thesis, SARS-CoV-2, more commonly known as COVID-19, is still a major threat that affects the daily lives of every citizen. The worldwide number of cases was still rising by the end of this thesis writing. The number of cases has passed well over 550 million, resulting in more than 6 million casualties [1], according to the World Health Organization.

Many pharmaceutical and medical companies have worked relentlessly to take countermeasures against this virus, creating vaccines and developing ways of testing individuals with great effectiveness.

But the costs of this are immense. All the machinery and logistics behind manufacturing the vaccine and testing people surpassed the millions, with the equipment necessary to analyze samples costing well beyond several thousand euros.

STAB VIDA took the initiative to develop a device[2], named DoctorVida Pocket (DVP), with the main focus of executing a COVID-19 test. It does so by using a simple, single step,

[1] NAAT is a more general term in which Polymerase Chain Reaction (PCR) is included as well as the current device's isothermal procedure

[2] A small video presenting the device: https://www.youtube.com/watch?v=1fibWgPFCNc

Reverse Transcription Loop-mediated Isothermal Amplification (RT-LAMP) process[3], as explained in section 1.1.3.

After the main stages of COVID-19 quarantine, STAB VIDA, motivated by the success of the DVP, focused on adapting it to new illnesses. The main one that proved adequate to be executed on the device was lactose intolerance, which also consists of a LAMP procedure, albeit a different and more complex one.

A problem with the initial device was that it was not able to execute more composite procedures, as is the example of PCR or multistep LAMP(RT-LAMP), being limited to the COVID-19 RT-LAMP test. To carry out more complex tests, it was necessary that the device STAB VIDA developed be altered while keeping the high level of correctness of the process. This would allow a more distributed method of testing for conditions, such as lactose intolerance, while keeping most of the guarantees of standard testing procedures.

Since PCR and multistep LAMP assays require multiple and distinct temperature behaviors, correct heating was necessary for valid results. As a result, this new device must include a high-efficiency heater to ensure a quick response time. The previous device's heater is adequate for this. Besides the heating control, if there was some sort of cooling mechanism that was powerful enough to cool down from high temperatures, the device would be able to cut down on assay duration significantly. There would have to be changes to the hardware of the device and the tuning of a controller for such a component, but once this was achieved, the device would benefit from a higher speed at which the temperature can reach the value set by the user, so much so that it would have a visible influence on the assay duration.

The speed at which temperature changes were attained was one component of the previous device that was disregarded since it did not matter when doing the actual assay because the heating portion was completed first. The assay procedure for the new LAMP processes includes a section in which the temperature is carefully and gradually increased. Therefore, one cannot simply minimize the response time of the controller. There had to be careful tuning and extensive testing to be sure of the accuracy of the results.

1.1.1 Rapid Tests

During COVID-19 pandemic, the term rapid test was associated with a test that could be done by the patient and the result was obtained in about 15 minutes. According to Dr. Betânia Ferreira[3], these are immunological tests based on antigen detection and tell us if there is an active infection. They look for the presence of the virus by screening for virus-specific proteins. These proteins on the surface of the virus are the viral antigen (which may induce the immune response), so rapid tests, for COVID-19 detection, are also known as antigen tests.

[3]To test for possible infection, NAAT tests are mainly used, which includes PCR tests, LAMP(RT-LAMP) tests and many others [2], which have different assay duration and details.

Rapid tests that test COVID-19 antigen can be done at the same place and time of where the sample was taken, and their results are available faster than NAAT tests. They are also much cheaper, as they do not need special laboratory equipment and be processed by trained professionals.

Dr. Betânia Ferreira also said that rapid tests are less sensitive than PCR tests, and therefore the likelihood of obtaining false negative results is higher.

1.1.2 PCR Tests

A catch to using rapid tests is that a negative result does not, by itself, exclude an infection as accurately as PCR. It may be necessary to perform a PCR test after a negative result if further confirmation is really needed. These are very specific and sensitive, detecting very low levels of viral RNA in the samples with a high level of accuracy, as Dr. Betânia Ferreira refers to in her article[3].

PCR tests work by amplifying a single DNA template into millions of identical copies *in vitro*. This is a methodical process involving multiple steps with different temperatures variations.

1.1.3 LAMP Tests

RT-LAMP amplifies RNA, allowing detection of a specific gene, whereas LAMP amplifies DNA. It has been successfully used for detecting respiratory viruses such as COVID-19 and genetic features that may suggest mutations or anomalies in the genome.

This is a valid alternative to PCR due to its high sensitivity, cost-effectiveness, and faster result time, adapted from Catarina Amaral et al. article [4].

The COVID-19 RT-LAMP process uses a constant temperature (single step), which is achieved by STAB VIDA's device, and the reaction is analyzed by using, among other possible methods, the fluorescence levels, which increases as the by-product of the reaction involving some enzymes, during DNA polymerization [4]. But there are many LAMP assays that are not isothermal, lactose intolerance being an example, needing multiple steps of different and possibly varying temperatures to amplify the DNA to allow the detection of the specific gene.

Developing a device capable of executing a NAAT test, such as LAMP (RT-LAMP) or PCR, has many difficulties, but if successfully done, it would prove very useful to health workers.

1.1.4 Lactose Intolerance Tests

The main focus of the initial device was to test for COVID-19 close to the patient, this being in the field or at a medical facility, avoiding the logistics of transporting the sample

to a laboratory for analysis.

With the decline of interest in COVID-19 testing by the population, STAB VIDA evaluated the possibility of executing different procedures that make use of the LAMP method. This meaning that the reaction in the sample tube produces fluorescence that is detectable by the device's sensors.

This includes a variety of different cases, and one that was focused on was the lactose intolerance LAMP procedure.

Lactose intolerance depends on the expression of the enzyme lactase, which is encoded by a gene. Mutation in the DNA sequence that regulate the gene have led to a sustained lactase production in the small intestine, giving the person the capability of digesting lactose. People without these mutation have a reduced ability to digest lactose as they get older, resulting in symptoms of lactose intolerance.

The gene that controls the production of the lactase enzyme is obtained as a combination of the genes inherited from both parents. The nucleotide T indicates the ability of producing lactase while the nucleotide C results in low levels of the enzyme. With these two that a person receives from both parents, it is possible to have 3 combinations of two nucleotides: CC, TT and CT. CC indicates that the production of the enzyme should be low, TT is the opposite, where the levels of production of lactase are expected to be sufficient. CT is the middle ground, where the person is capable of producing enough levels of the enzyme but might still suffer from intolerance due to other factors, adapted from Dallas M. Swallow article[5].

The assay to test for the presence of a mutation in a specific DNA sequence is a multistep process.

The process is performed on whole blood samples that were subjected to a specific procedure outside the scope of this thesis. Afterwards, these samples go through multiple temperature changes. More specifically, the temperature is set to 65ºC for 30 minutes, which then changes to 40ºC for 10 minutes, and finally a steady increase of temperature until 80ºC are reached.

These conditions must be replicated in the pocket device. It must reach 65ºC initially, and this is not a concern since that is about what the COVID-19 test temperature was. Once this stage is done, the device must then cool down to 40ºC and this is where the limitations of the hardware are noticeable, since there is no means of actively cooling the chamber. Once this last constant temperature period is finalized, the last step is to increase the temperature at a steady rate, something that the initial design of the pocket device did not conceive of.

1.2 Problem & Challenges

From the described LAMP processes, both for COVID-19 and lactose intolerance, it is clear that they need fairly rapid and accurate changes in temperature to be able to conduct

a valid experiment. Many problems arise if the temperature is incorrectly set or if it takes too long to settle. These are to be avoided whenever possible since they can affect the result without being noticed.

These are a few examples of the problems that arise from incorrect temperature control in the case of PCR process from the BioRad website[6]:

- If the annealing temperature is too high, primers are unable to bind to the template.
- If the annealing temperature is too low, primers may bind nonspecifically to the template.
- If the ramp speed of the cycler is too slow, spurious annealing may occur due to lower temperature and sufficient time for nonspecific binding.
- If the denaturation temperature is too low, the DNA will not completely denature and amplification efficiency will be low.

For the lactose intolerance LAMP test, certain problems can be detected if temperature control is not properly managed.

Briefly, there is an enzyme that is responsible for the production of the fluorescence measured by the sensors. With incorrect temperature control, the peaks of fluorescence can be deviated from what was expected, resulting in incorrect results.

These assays can be done very effectively in laboratories in possession of the correct machinery and workers to operate them, but this is not cheap, as many of these tools cost thousands of euros.

Besides the cost, another problem is that they are also stationary, in a laboratory, meaning that the sample has to be transported there in order to be analyzed, adding up to the logistics cost of the operation.

1.3 Objectives

Initially, the device in production by STAB VIDA was only capable of setting one temperature and maintaining it. With the new addition of temperature changes, it broadened the possibilities of use cases for the device. PCR was the main focus at the start of the thesis planning, but it was certain that many other use cases would emerge, such as the previously mentioned lactose intolerance test using the LAMP procedure, which ended up being the focal point when developing the new device. A device capable of producing multiple temperature cycles has many other uses in the medical field, which will be explored using the newly built device.

Knowing this, the goal of this thesis was to build, using already-existing technology, a device that could go through multiple temperature cycles and temperature ramps.

This included developing the firmware the device needed to execute assays effectively, connecting this firmware to an application responsible for communicating the assay

parameters to the device, and displaying the readings made by its sensors back to the user. To complement this, there is also an API which stores data about the users and assays.

Although the main goal was to make a firmware that works and makes the most of the device's hardware, the application and API were also important for the system to work correctly.

Once these capabilities were fully functioning, the remaining part of the thesis was spent on improving the device, namely implementing an active cooling system, which included both hardware additions and firmware development.

For a more clear overview, the following list presents the objectives that were attained during the thesis development:

- Firmware changes to support new type of assays with multiple temperature steps.
- Script capable of executing and assay on the device while obtaining the data produced by it and presenting it to the user.
- Alteration of the application to be able to execute an assay on the device, much like the script previously did.
- Testing of a new prototype with the new addition of active cooling.

1.4 Document Structure

The document will follow a progressive structure, starting with the state of the art and finishing with the future work. This section will present said structure.

The state of the art will present past work that has relevance to what is going to be discussed in future chapters. It will be mostly about the DVP device and its firmware, giving a bit of an insight into the software later in the chapter.

The third chapter, called Proposal, was created to give the reader a better idea of what the initial plans and propositions made by STAB VIDA were and what later became of these plans.

The most important chapters are the fourth and the fifth. These discuss the development, difficulties, and achievements of both the firmware and software, respectively. These are followed by the sixth chapter, which is dedicated to presenting the results obtained from the work previously done.

The thesis document will end with the last chapter, which will talk about possible future work that could be added to the firmware or software and give a conclusion to the thesis.

2

State Of The Art

The system that was developed uses multiple hardware components. The controller for each component was coded into the firmware of the device and is accessed and manipulated at different points in the device's assay processing. This firmware is composed of many parts, which will be present in this section.

For a better comprehension of the hardware structure of the previous device, this will be the first subject presented. The firmware that is responsible for handling the temperature, sensors, Bluetooth communication, and other hardware components will come afterwards, which will be the main focus of this chapter since it is where most research was made. Finally, the application implementation. This is relevant since this is where the user is able to interact with the DoctorVida device (Figure 2.1) and where various changes were made. Therefore, it is relevant to approach how the application was initially structured.

2.1 Previous Work

In keeping with what was discussed in the preceding chapter, STAB VIDA produced and developed an initial device that could conduct an isothermal LAMP test. This test was done at 65ºC for 40 minutes, and by the end, the user knew if the result was positive or negative for COVID-19 exclusively.

This device, Figure 2.1, utilizes a heater that heats up to a temperature near the 65ºC and once it stabilizes at that temperature the assay begins. It also has a multi-spectral light sensor capable of detecting fluorescence changes for one or more dyes that are excited by the excitation LED sources available in the device (currently two LEDs: blue and green). The results of the sensor readings are shared with an application via Bluetooth. For the processing unit, this device has an ESP-32, which is responsible for controlling all the sensors and actuators.

Using this device as a basis, an improved system was built that is able to adjust the temperature and keep it for a whole cycle, after which the temperature can be modified and a new cycle begins. Temperature changes must be quick and precise, which required

Figure 2.1: DoctorVida Pocket device alongside the application[7]

the use of a temperature controller and temperature actuators. The previous device had no need for a cooler, but since new assays have different temperatures for different cycles, it may happen that the temperature of a cycle is inferior to the previous cycle temperature. This requires cooling the device, which can currently only be done passively. Thus, STAB VIDA considered introducing some method for active cooling of the device, which will be discussed in future chapters.

2.2 DoctorVida Pocket Hardware

The initial device contained all the necessary components to execute multistep LAMP procedures, all the changes made were on the firmware side. This section serves the purpose of presenting the hardware components that are fundamental for the execution of any kind of assay on this device. There are many components that make up the device hardware. To keep this section concise and clear, and due to company confidentiality, only the main ones will be briefly discussed. These include both the components necessary for running the assays and for interaction with the user of the device.

2.2.1 Micro-controller

Both the initial and the current version of DoctorVida use an ESP-32 microcontroller, which is a compact, integrated circuit programmed to manage the device's operation. It is responsible for interpreting data it receives from its peripherals and communicating with them to enact the appropriate action.

Besides being responsible for the operation of the device, it also incorporates Bluetooth and Wi-Fi communications, usually executing the functions relative to these components on a different core, allowing them to be processed in parallel to the main program.

2.2.2 Wireless Communications

The microcontroller comes equipped with Bluetooth and Wi-Fi connectivity, although only Bluetooth is currently used to communicate data to the application, which then does its own processing and communication with the backend via Wi-Fi. This way, there is never communication between the pocket device and the server.

Bluetooth, as said, is used to send data to the application so that it can interpret it and display useful information to the user. Currently, the device is using a variant of Bluetooth called Bluetooth Low Energy (BLE). This version uses a lot less power than traditional Bluetooth because it stays in sleep mode until a connection is made.

The following sections present the insides of the BLE protocol, giving a better understanding of how communication is possible using this technology, as well as the benefits and downsides that come with it.

2.2.2.1 Bluetooth Low Energy Protocol Stack

The BLE protocol can be represented by a stack, Figure 2.2, that has multiple layers ranging from the lower levels, which handle signal transmission using radio frequencies and comprise the Controller part of the stack, to the upper levels of the Host part, responsible for defining how communication between connected devices should be prosecuted.

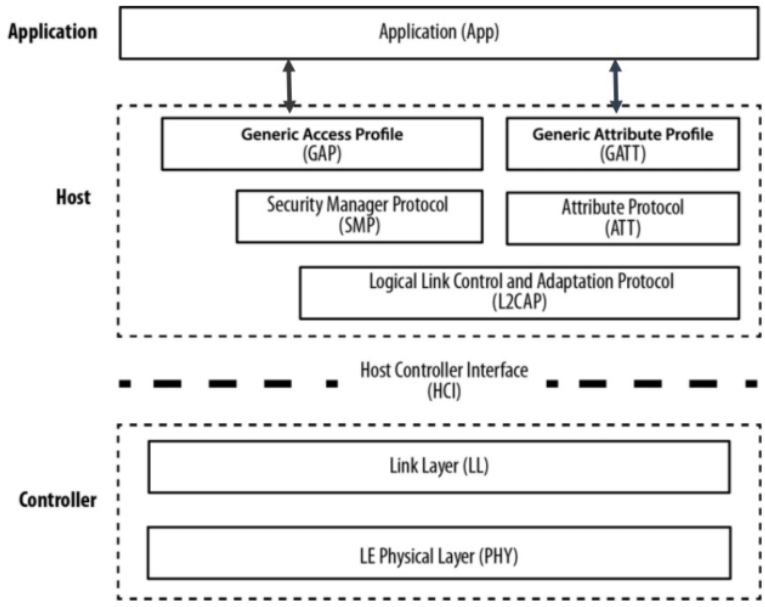


Figure 2.2: Bluetooth Low Energy protocol stack representation[8]

2.2.2.2 BLE Physical Layer

BLE makes use of the 2.4 GHz Industrial Scientific Medical (ISM) band and has 40 Radio Frequency (RF) channels spaced from each other by 2 MHz. Each channel can be

one of two types: advertising channels or data channels. The first are used to discover devices, establish connections or broadcast transmissions, while the latter are used for bidirectional communication between connected devices [9].

2.2.2.3 BLE Link Layer

BLE can be used to broadcast data, receive broadcast data, or for bidirectional communication between devices. When broadcasting, the device, which is known as an advertiser, transmits data in advertising packets through the advertising channels at intervals of time called advertising events. On the other hand, if a device is listening to advertising channels, they are called scanners[9].

Two devices must be connected to each other for bidirectional communication. An advertiser device announces that it is available to connect through the advertising channels. The device that listens to these advertisements and wishes to connect to the advertiser is known as the initiator and is the *master* of the connection. The initiator sends a Connection Request to the device it wants to connect to, known as the *slave* of the connection, which then creates a point-to-point connection between the two. Therefore, a *master* may have multiple *slave* connections, which is a network with a star topology nominated piconet. Currently, a device can only belong to one piconet. In the case of the DVP, up to 4 of them can be connected to one application at the same time.

After the connection is established between *master* and *slave*, the physical channel is divided into time units called connection events, which do not overlap. In these connection events, any data sent through the physical layer is done using the same data channel frequency[9].

2.2.2.4 BLE L2CAP

Logical Link Control and Adaptation Protocol (L2CAP) is a protocol responsible for multiplexing the data of the three higher layer protocols, Attribute Protocol (ATT), Generic Attribute Profile (GATT) and Security Manager Protocol (SMP)[9]. Fragmentation and recombination of L2CAP data units at the link layer allow L2CAP and higher-lever protocols to use larger payload sizes. When fragmentation is used, larger packets are broken up into multiple link layer packets, which are then put back together by the link layer on the peer device.

The default size of a L2CAP packet is 23, as explained by the Texas Instruments guide on BLE[10]. In their guide, devices assume a Protocol Data Unit (PDU) of 27 bytes, corresponding to the maximum size capable of being transmitted in a single connection event packet. Since the L2CAP protocol header is 4 bytes, the default Maximum Transmission Unit (MTU) is 23 bytes, Figure 2.3.

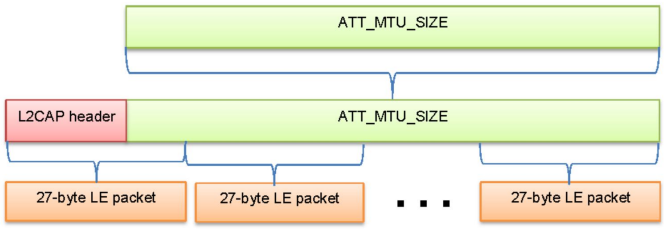


Figure 2.3: Bluetooth Low Energy L2CAP packet fragmentation[10]

2.2.2.5 BLE Attribute Protocol

The ATT is used to configure the communication between two devices on top of a dedicated L2CAP channel, in which one has the role of server and the other client. ATT is used by the server device to expose a set of attributes and their corresponding values to the client.

Any device can play the role of a server or client, but every device can only have an instance of a server at any given time[8]. The server exposes the attributes, which contain 3 properties: the attribute type, the attribute handle, and a set of permissions. The client can then discover, read, and write these attributes.

The communication between devices usually follows a request-response type of protocol. This pair (request-response) is considered a single transaction, meaning that if a client submits a request to the server, it will not send any other request to the same server until it has received the response to the request[8]. In the case of notifications, the server can transmit information and not wait for any response. These alerts are sent to any device that has chosen to be notified when an attribute value changes.

The data found in an attribute from the ATT is managed by the GATT protocol, which will be discussed next.

2.2.2.6 BLE Generic Attribute Profile

The GATT is a service framework that is built on top of the ATT. The data structure, which can be seen in Figure 2.4, is hierarchical, in the sense that it has a GATT profile constituted by services and these by characteristics. Therefore, the server contains the attributes, and the GATT profile defines how to use the Attribute protocol for discovering, reading, writing, and notifying these attribute values.

A characteristic is used to expose values to the client devices, and, depending on the permissions defined by the server, the client can read, write, be notified or all three of them. The characteristic initialization must set its ID using a Universally Unique Identifier (UUID) and the set of permissions it will support (read, write, and notify). This can be seen in Figure 2.5, which is a Bluetooth code example and sets the characteristic permissions to read, write, and notify.

Besides its basic information, the service can also add a descriptor to the characteristic

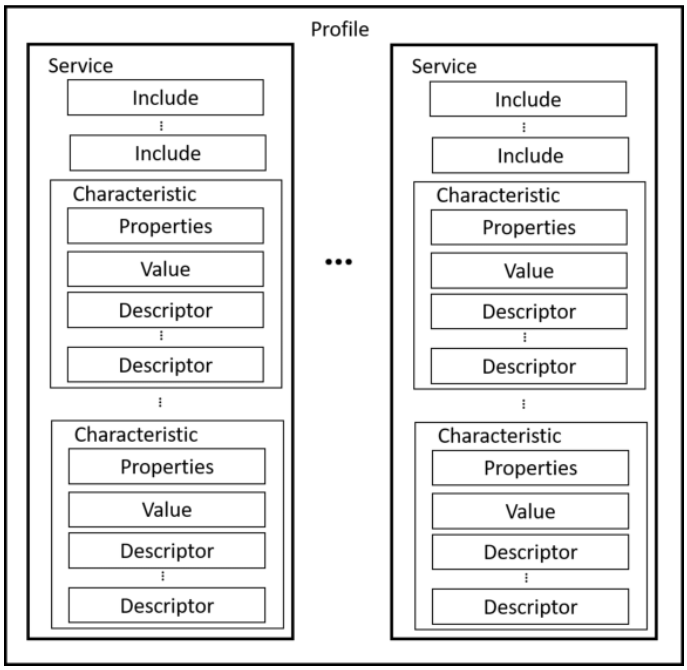


Figure 2.4: Bluetooth Low Energy GATT profile hierarchy[8]

upon creating it. Descriptors are mainly used to provide the client with metadata about the characteristic. They can provide a user-readable description of the characteristic, extra properties or, most importantly, switch on or off server-initiated updates, which is a very useful feature for some use cases[11]. For example, the one seen in Figure 2.5, is the BLE2902 created by Kolban[12], and would be used for assisting in notifying clients.

A very useful feature of the BLE library used by the DVP firmware is the callback function, which can be added to a characteristic. These callback functions define what the server should do in case a read or write is requested by one of the clients. This allows for clients, which would be the application in the case of DoctorVida, to trigger events and state changes by calling for a read or write of the corresponding DVP device. This is used in a variety of situations, including sending the initial protocol, reading sensor values, resetting the device to its initial state, and so on.

```
exampleChar = exampleService->createCharacteristic(
    BLEUUID(EXAMPLE_CHARACTERISTIC_UUID),
    BLECharacteristic::PROPERTY_READ | BLECharacteristic::PROPERTY_WRITE | BLECharacteristic::PROPERTY_NOTIFY
);
exampleChar->addDescriptor(new BLE2902());
exampleChar->setCallbacks(new ExampleCallbacks());
```

Figure 2.5: Bluetooth Low Energy characteristic definition example[8]

Unfortunately the device limits the number of characteristics that he can possess. This was defined upon creating the Bluetooth service, here the number of handles was indicated. The handle is a 16-bit unique identifier for each attribute in a particular GATT server[11].

Each characteristic needed a certain amount of handles depending on the amount of descriptors they had. The calculation of the total number of handles needed for a service can be better explained by using an example. Considering that there are three characteristic in a service, where the first characteristic has two descriptors, and the second characteristic has only one descriptor, and the third characteristic has no descriptor. Then the number of handles needed should be 1 (service attribute) + 4 (1 characteristic declaration attribute, 1 characteristic value attribute, 2 descriptors attribute) + 3 (1 characteristic declaration attribute, 1 characteristic value attribute, 1 Descriptor attribute) + 2 (1 characteristic declaration attribute, 1 characteristic value attribute) = 10.

This equation was explained by Espressif contributor Yulong-espressif in an answer to a question made by BLE2902 creator Kolban[13].

2.2.2.7 Bluetooth vs Wi-Fi

In the final device and since the initial design, the pocket device communicated exclusively with the application using BLE, and up to 4 devices could be connected at once to the application. One possible alternative to this would be the pocket device to directly communicate with the backend via Wi-Fi.

There are several reasons for only using Bluetooth rather than also using Wi-Fi. Dilip Geevarghese [14] makes a good comparison between both technologies. In it, he explains a few reasons that make BLE preferable over Wi-Fi in some cases. The main reason is the power consumption of these technologies. Since Wi-Fi signals can travel ten times as much compared to BLE it also consumes a lot more power. Dilip Geevarghese refers that Wi-Fi can consume up to forty times more energy than BLE. Since the mobile phone running the application is assumed to be close to the pocket devices in a point-of-care use case, this extra energy would be wasted and could even heat the system, interfering with the assay results.

Another relevant topic is transfer speed, Wi-Fi greatly surpasses BLE in this department, but it is important to notice that the size of the data transferred is very small, up to 600 bytes. Therefore, BLE is adequate for this since it supports a maximum data rate of 1Mbps [15], which is more than enough for the data generated by DVP devices.

This being said, STAB VIDA, by the end of this thesis, was considering introducing Wi-Fi communications between the DVP device and the backend. This would allow the user to leave the device in a location without the concern of losing the BLE connectivity, which would break the flow of communication between the app and the pocket device. Additionally, it would enable users to use the backend to send commands to the pocket over Wi-Fi from any location as long as there is network connectivity, and to control any number of pockets since the application would be communicating with the backend.

Although these points seem very positive, shifting the device to use Wi-Fi requires massive changes in the firmware, possibly the hardware, API, and application, and raises security concerns and logistical concerns such as connecting the device to a new network

and energy consumption. Despite these obstacles, STAB VIDA believes that this project is practical and would pave the way for future endeavors that are outside the scope of this thesis.

2.2.3 Excitation Source

In order to excite the enzymes within the sample tube, there must be an excitation source to trigger the reaction. The DVP has 2 sources of excitation, a blue LED and a green LED. Each of them emits light at a specific wavelength. This way, it is possible to execute more protocols than if there was only one excitation source, since each one can excite different enzymes present in the sample, which will produce fluorescence with different wavelengths, captured by the multi-spectral sensor.

2.2.4 Fluorescence Reading

Throughout the program's execution, sensory data must be collected, namely the fluorescence levels found in the sample tube, a product of the reaction within. This is the responsibility of the multi-spectral sensor, which reads up to six channels. Each channel captures a different light wavelength therefore they can be associated with a color, which are the following, in order from lower to higher wavelength: violet, blue, green, yellow, orange, and red.

The excitation and reading wavelengths are independent, meaning the user can choose to excite the enzymes in the sample at a wavelength different from the one used to read the fluorescence levels present in the sample tube. This is usually the case due to the nature of the enzymes, and this information is usually provided to the professional who is using such enzymes.

2.2.5 User Interaction Components

The pocket has a means of transmitting to the user information via light signals or sound signals. Light signals use an RGB LED present in the device to display various color-coded messages, per example, if the device is ready to run an assay.

Sound signals are produced by a buzzer and are mostly used to inform of some event that needs the user's attention, per example, if the device is waiting for the sample tube.

2.2.6 Heating System

The resistor is the component responsible for heating the chamber where the reaction is occurring. If there is a problem with this piece, the whole assay may not start or, worse, be invalid. The resistor has a lower temperature limit of room temperature and an upper temperature limit of around 100ºC. Although such high temperatures put a high strain on the device, primarily on the resistor itself, due to this, the resistor's rate of heating degrades considerably as temperature rises.

2.2.7 Temperature sensor

The temperature sensor, which is responsible for the correct reading of the temperature inside the chamber, is arguably one of the most important components of the device. Since every single assay that is executed on DoctorVida's pocket requires reasonably accurate temperature control, if this sensor has any problem, it may invalidate the whole procedure. The temperature needs to be able to hold a certain temperature for a period of time defined by the assay protocol. Using the resistor as the heater and with the temperature sensor, it is possible to calibrate and control the temperature output to the chamber.

2.3 DoctorVida Pocket Firmware

The firmware is responsible not only for controlling the different hardware components using their drivers, but also for running the program that is able to execute an assay. It was developed and is maintained using the PlatformIO work environment [16]. PlatformIO allows you to select the board that is in use, and it will download the required tool-chains and install them automatically. This way, it is straightforward to obtain and use external libraries, simplifying the process of declaring dependencies on external projects, as is the case with SparkFun AS726X [17], which is an Arduino driver to work with the multi-spectral sensor.

Besides library and project management, with PlatformIO it is possible to directly compile the project for a certain board, that has been selected by the user and flash the compiled firmware to the respective board.

2.3.1 Firmware Main Loop

To make use of the different hardware drivers, the firmware has a main program that works in a never-ending loop. Before entering this endless loop, the device on initializing will execute a setup of multiple elements, such as the initial LED color, memory management, device settings, the state machine, and then go through a device testing procedure, which decides if the device is able to run any assays it needs to.

Once the initialization process is complete, it will then begin the main loop. This loop consists of a state machine (Figure 2.6) executing the set state. For context, the initial state is called **Init State** and the main purpose of it is to set the LED to blue, heat up to a default temperature, and reset the state machine variables. After this, it waits for the arrival of a protocol that triggers a state change.

Every state has an enter function that is executed once (1 loop) upon entering the state and a run function that will run endlessly (x loops) until there is a state change. This run function always returns the next state to be executed. If the workload of the current state is unfinished, it keeps returning a pointer to itself, otherwise it indicates the next state. For example, the **Init State** will keep sending back a pointer to itself until it receives a

protocol from the application over Bluetooth. After that, it will send back a pointer to the next state. The application is actively listening for state changes and periodically asks the device what state it is on, this allows the application to decide what screens to display during an assay.

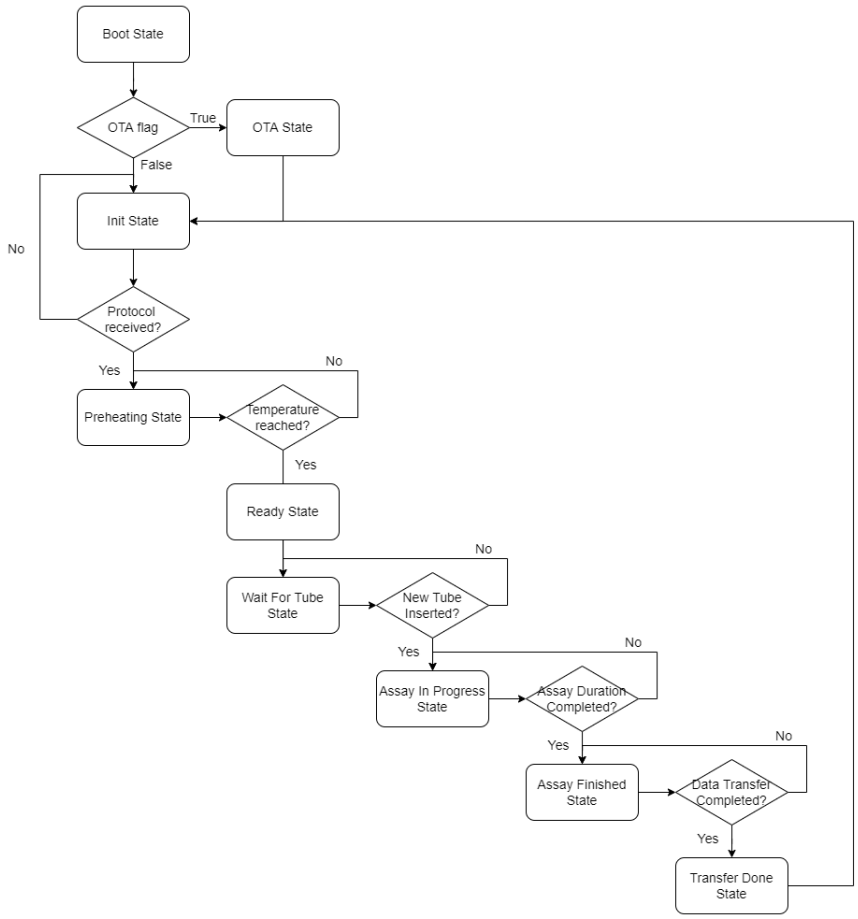


Figure 2.6: Initial DoctorVida Pocket device state diagram

2.3.2 Temperature Controller

To complete this DVP device firmware insight, it is most relevant to reference the temperature controller. Currently, the device is using a PID library created by AdysTech called PIDPWM[18]. PID is a form of feedback control that uses the current output as an input to calibrate the next controller output.

Feedback controlling and, more concretely, PID will be discussed in the next sections, giving a better understanding of the previous work and the inner workings of this type of temperature control algorithm. Besides the feedback control, there are more details present in the firmware of the driver that are worth mentioning, such as the square wave used by PIDPWM to control the signal output to the resistor.

2.3.2.1 Duty Cycle

The duty cycle is the percentage of time a digital signal is in the high state over a period of time. Above 50% duty cycle means the signal spends more than half of the time in a high state and the opposite if the duty cycle is below 50%, example in Figure 2.7. If the duty cycle is 100% it is the same as setting the voltage to constant high and 0% is the same as grounding the signal [19].

$$Duty\ \text{Cycle} = \frac{T_{HIGH}}{Period} \times 100$$

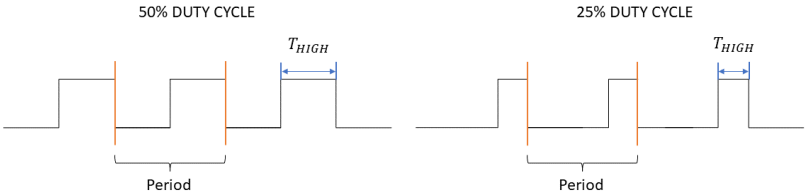


Figure 2.7: Duty cycle example based on Figure from Sensor Solutions[20]

2.4 Feedback Controllers

The DVP device initial configuration did not account for temperature variation during an assay. With the new device that was developed this had to be taken into consideration, thus some research was made in the field of temperature control.

In this section, the research on feedback control is presented, giving a small introduction to various types of feedback control technologies and later showing what is the norm when using them and what has been made in this field throughout the years. Afterwards, a focus on proportional integral derivative control will take place, since this is one of the main implementations.

Usually, a system used by the industry consists of a reference input to the system, the value of which is inserted by the user to correspond to what he desires the output to be. This value is treated by a controller that outputs an input to the system process, this process, also known as an activity or plant, being the component that generates an output measurable by the user. Therefore, in the case of feedback control, this output is feed back to be inputted into the controller. The changes in the system affect the system itself, as illustrated in the Figure 2.8.

There is a process variable, measured by some instrument, that is the parameter controlled by the system and that is feed back to be compared with the set input value by the user. This variable suffers the changes made by the controllers and transfer function in order to reach the desired value defined by the input value.

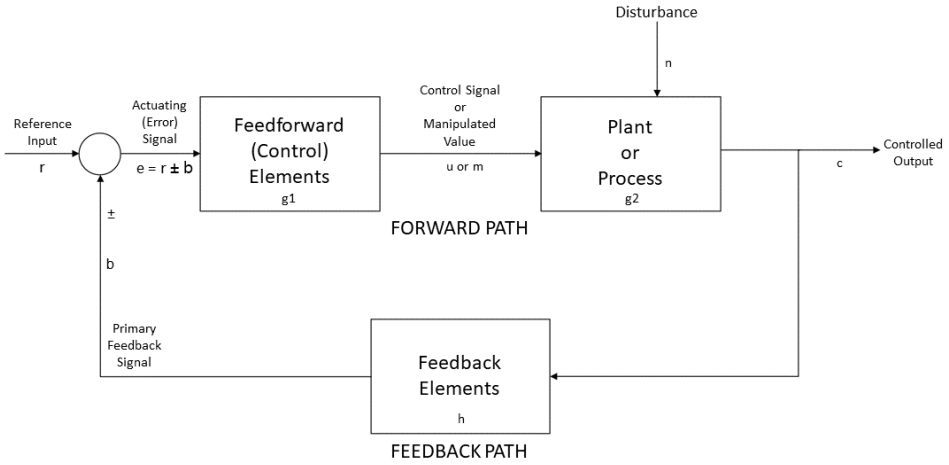


Figure 2.8: Representation of the feedback control loop[21]. Definitions in annex (I.2.1)

Maggay [22] affirms that feedback controllers are closed-loop controllers in which the output is feed back to the input. This means the control process depends on the desired output. This is harder to design and costlier, but also more accurate than open-loop controllers, where output is not feed back to the input, as can be seen in Figure 2.9, shown in the same article.

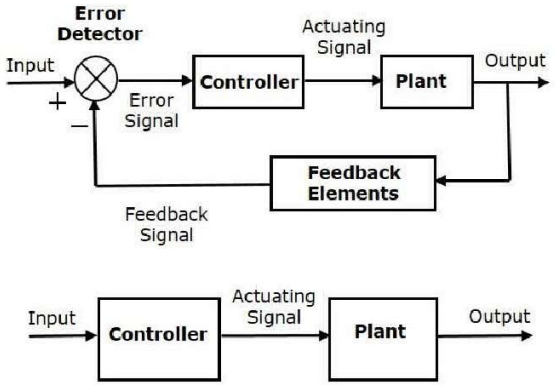


Figure 2.9: Difference between feedback controllers and no feedback controllers [22]

There are two types of feedback: positive feedback and negative feedback. Both of these

are going to be discussed in future sections. They have the common factor of using a transfer function to characterize the output based on the input received.

2.4.1 Transfer Function

Normally, there is a differential equation in the temporal domain that characterizes the system, transforming the input value into an output value. The Newton Law $F = ma$ can be used as an example. This is the example given by professor António Dourado [23]. Imagine a system that consists of a body and in which an external force is applied (U) and there is also attrition affecting the object ($B\frac{dy}{dt}$, where B is the coefficient and the derivative is the velocity). This means there is a system with an input (applied force) and an output (object acceleration, represented by the second degree derivative) that can be represented with the following expression:

$$U - B\frac{dy}{dt} = M\frac{d^2y}{dt^2} \tag{2.1}$$

The most effective way of dealing with this equation is by using a Laplace transform, applying it to both the input and output and solving the differential equation, going from the temporal domain to the complex one.
From this, the transfer function is obtained[1]:

$$Y(s) = \frac{1}{Ms^2 + Bs}U(s) \tag{2.2}$$

$Y(s)$-Laplace transform of the output
$U(s)$-Laplace transform of the input
$\frac{Y(s)}{U(s)} = \frac{1}{Ms^2+Bs}$ - Transfer function

this function is for when the initial conditions are null[2], in which case it gives the relation between the output and input Laplace transforms, since there is no dependency between the output and input at this stage.

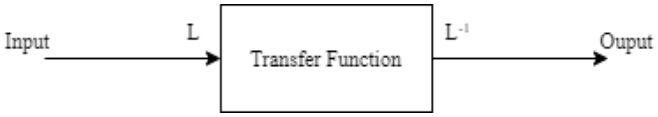


Figure 2.10: Transformation of input to output

Finally, it is necessary to convert the result from the complex domain back to the temporal domain, using the Laplace Inverse, Fig. 2.10.

This example demonstrates the purpose of the transfer function and how it allows calculating an output based on an input, effectively characterizing the system. In the case

[1] The expression is relative to the Newton's Law example, with calculations steps attached
[2] Otherwise it is not a transfer function and there is another parcel for the conditions of an ongoing system, Equation I.1

of feedback control, the transfer function is used in the controller to process the input plus the feedback and produce a valid input to the process/plant.

2.4.2 Positive Feedback

When it comes to positive feedback, the reference input is added with the feed back output (process variable) to generate the new input to the controller, as seen in Figure 2.11, this results in small disturbances being escalated into noticeable changes. This means a positively feed back system will grow until an external effect affects it. As said by Donella Meadows [24] "... a positive feedback loop is self-reinforcing. The more it works, the more it gains power to work some more. ... Positive feedback loops are sources of growth, explosion, erosion, and collapse in systems. A system with an unchecked positive loop will ultimately destroy itself. That's why there are so few of them. Usually, a negative loop will kick in sooner or later.".

$$T = \frac{G}{1 - GH} \tag{2.3}$$

Transfer function of the positive feedback controller, more detailed in annex I.3, based on John Lester Maggay article [22]

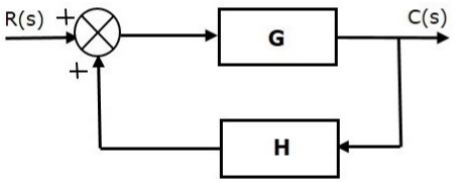


Figure 2.11: Representation of the positive feedback control[22]

2.4.3 Negative Feedback

Negative feedback serves to maintain system equilibrium. It strives to compensate for any disturbance, this being either a positive or negative alteration. If the system measures an output higher than the reference input value, the controllers will correct the input, leading to a decrease in the output value, which in turn reduces the error feed back. The same principle is used when the output measured is below the desired value, but in this case, the transfer function will aim to increase the output. When the objective is attained, the feedback error is zero, resulting in no changes being made since the system is in equilibrium. Figure 2.12 represents the Negative Feedback model.

The transfer function below, from John Lester Maggay's article [22], demonstrates that the value feed back will penalize the process variable depending on the size of the control error, the opposite to what happens in positive feedback, leading to eventual stabilization of the system.

$$T = \frac{G}{1 + GH} \tag{2.4}$$

Transfer function of the negative feedback controller, more detailed in annex I.2

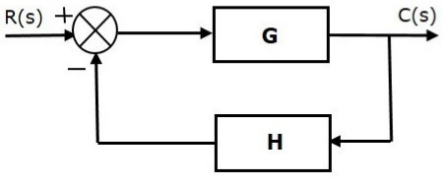


Figure 2.12: Representation of the negative feedback control[22]

2.4.4 Types of Feedback Control

There are some types of feedback control that can be used to control a system's behavior. These range from the simple on-off controller to the more complex PID controller.

2.4.4.1 On-Off Control

When the process variable measured from the process is lower than the reference input value, the feedback instructs to increase the input to the process (p.e. max the power supplied to a heater), on the other hand if the process variable is higher, the feedback is to decrease the input (p.e. turn the power to the heater off). The controller switches between supplying a very positive input signal to the process and a very negative one[25]. An extra care can be taken to prevent the system from switching between the positive/negative signal very rapidly when approaching the ideal value. This would be to differ, by a set amount, the threshold value at which the signal is switched between positive feedback and a negative one, also known as **hysteresis**[26].

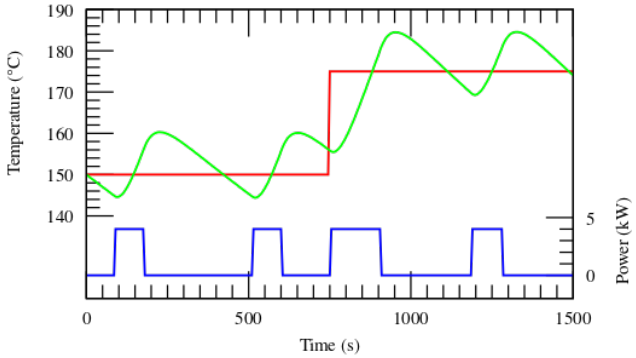


Figure 2.13: Representation of the on/off control[26]

Even if perfectly configured, this type of control would be far from ideal for the purpose of this thesis because it is necessary to maintain a temperature for a long period

of time without much deviation, so its mention was merely to provide a foundation for the next controllers analyzed.

2.4.4.2 Proportional Integral Derivative Control Components

The PID control mechanism utilizes three different components: proportional, integral, and derivative, in order to work the signal it receives from the user (input value or set value) plus the error that is feed back, monitored through a process variable, generating a new signal that it inputs to the transfer function, which then outputs a value usable by the process of the system and also feed back to be used on the next iteration. Each of the three components plays an important part in the system's response to changes in the set value, which is going to be discussed next.

2.4.4.3 Proportional Control

The proportional component of the PID control mechanism, also known as *proportional gain*, uses the difference between the reference input value and the process variable, otherwise known as control error. It applies a multiplier to the control error term, which is the gain, resulting in an accelerated response by the control mechanism, as can be seen in the graph 2.14. The proportional gain can control any type of activity as long as there is feedback that generates an error value. Although versatile, it rarely properly resolves Feedback Control individually due to limited performance, where ringing can happen, and the fact that it has steady-state error different from zero [27]. This meaning that it is not able to settle on the reference input value, and instead the end result signal will be offset. This can also be seen on the graph, where we can verify that as the proportional gain component is increased, the response time decreases, but the before mentioned problems arise, resulting in unwanted performance.

The ringing problem that happens with proportional control is caused by a gain value that is too high leading to a sequence of overshoots, each with less amplitude than the previous, due to the error getting smaller with each overshoot, until the process variable stabilizes close to the set value [26].

Steady-state error, as explained, means the process variable does not stabilize at the set value. As the process variable nears the set value, the error will decrease and eventually get closer to zero. The system may be externally influenced by, for example, a heater that loses heat to the surrounding environment. This type of inefficiency is common in the real world. Because proportional gain is a scalar that multiplies the control error, if this last value is close to zero, the proportional gain multiplier will be insufficient to compensate for the system's energy loss and force the error to zero.

2.4.4.4 Integral Control

Integral control is used to eliminate steady-state error. It does so by summing the error value throughout the process of correcting it until it reaches 0. That is, even minor errors

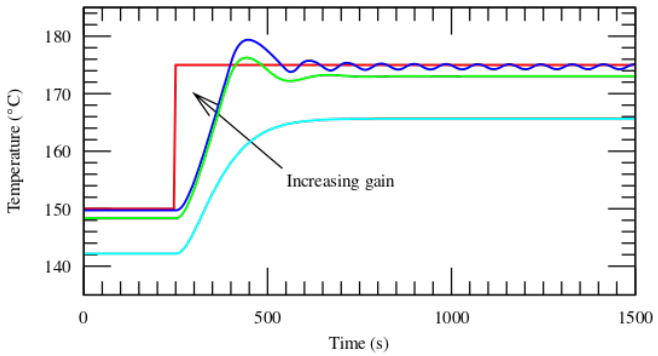


Figure 2.14: Representation of the proportional control[26]

will accumulate over time and eventually become large enough to force a response from the controller.

This controller can be multiplied by a value known as *integral gain* or *reset level*, which will vary the response time of this component, as well as other aspects such as the settling time and stability[26].

This component may introduce a problem known as wind-up when there is an abrupt change in the set input value, generating a rise (or descent) in the process variable, and it is aggravated by a large integral gain.

Since the integrator keeps summing the errors during this, it may happen that it accumulates a large amount of error value, leading to an overshoot. This can be stopped in many ways, one being conditional integration, where integral action is only applied when certain conditions are met, such as the error value being low [28].

2.4.4.5 Derivative Control

This type of control introduces the derivative action, which acts on the rate of change of the control error [27], meaning that if the reference input value suddenly changes, either by the user or some external factor, introducing a large control error to the equation, the derivative action will make the response act rapidly while the error is large, leading to a drastic change in the process variable, decelerating as the error gets smaller and eventually disappearing as the error gets closer to constant. This is useful to mitigate the overshoot and stability problems, such as *ringing*, that happen when using a proportional control with high gain[26].

The derivative component can be multiplied by a constant, known as *damping constant*, which will control how rapidly the response is influenced by the derivative control. If this value is too high, the response will be slow, but if it is too little, the response may overshoot and cause ringing, as can be seen in Figure 2.15, where the derivative is applied before the proportional control actuates.

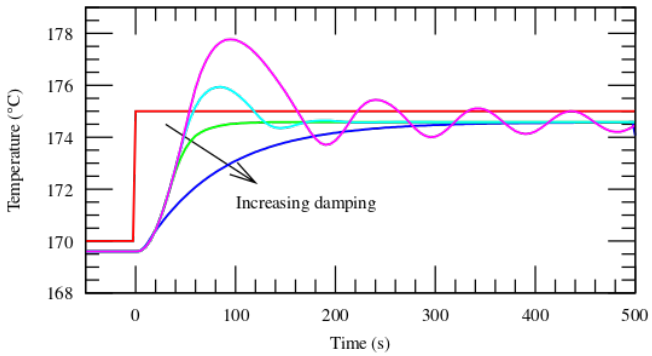


Figure 2.15: Representation of the derivative control[26]

2.5 Proportional Integral Derivative Control

The PID controller used as reference in this document follows the structure present in Figure 2.16, this structure assumes the proportional gain is applied to the integral and derivative components, while these themselves are multiplied by some constant. This is known as the *Ideal Form* of the PID controller[3]. This is one of many approaches. Other ones may process all the components in parallel, or process them all sequentially. These are going to be talked about in 2.7.

Some models use time constants I.4, in this case a long integral time constant is the same as a low value of I, this can be understood by the analysis in annex, where $I = \frac{1}{T_I}$, whereas a long derivative time constant means a large value of D, adapted from [26].

The structure used here is the same used to explain the different controllers in the previous segment and follows the following function:

$$W = P\left((T_S - T_O) + D\frac{d}{dt}(T_S - T_O) + I\int (T_S - T_O)dt\right) \tag{2.5}$$

Where W is the outcome of the controller, Ts represents the set value, To represents the output of the process variable, and P, D, and I represent the multipliers applied to the proportional, derivative, and integral components, respectively.

This function was obtained from:

$$u = K_P e + K_I \int e dt + K_D \frac{de}{dt} \tag{2.6}$$

which was adapted from the company TLK ENERGY website [29], extra steps in annex I.4. This equation is the most commonly used mathematical representation of the ideal

[3]The professor that was referenced for the structure of the PID and subsequent controller explanation freely referred to the controller as being of the parallel form, but later explained that it is in fact called the ideal form. Because these two types of controllers are so similar and conversion from one to the other is possible, the industry and many manuals on the subject refer to them as parallel form. Better analysis is going to be done in the future section 2.7

form of PID controller and, using the notions mentioned, there was a conversion from the time constants T_I and T_D (present in I.4) to I and D, respectively.

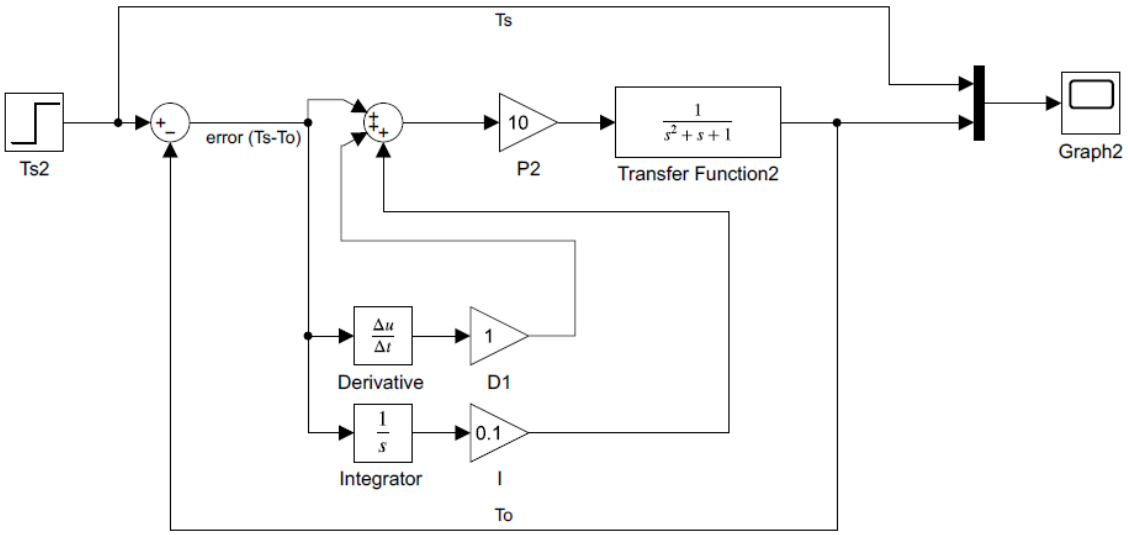


Figure 2.16: Proportional Integral Derivative control structure that was considered[4]

Analyzing each component of the PID control mechanism, it is possible to use them to complement each other, attenuating the downsides each has. A tool named Simulink was used to demonstrate the behavior of these components when put together. This tool is useful for modeling and simulating systems. Using Simulink, models representing a system with the different components were put together, so that the different components and their behaviors and impacts on the system could be analyzed.

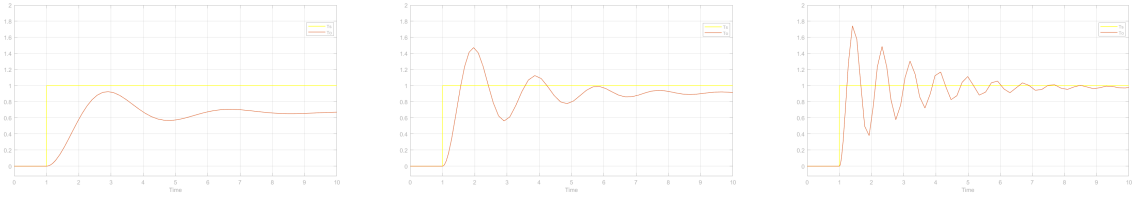
For the sake of demonstration, a transfer function was chosen that allows us to neatly display the behaviors in question, namely the ringing and the steady-state error, while also showing that adding more controllers may fix problems that would happen if said controller was not present, as can be seen in the soon to be presented graphs. Therefore, this is merely an example, as different systems and transfer functions may suffer from different issues at different values of the parameters respective to each controller.

It is important to note that there is a step function that will trigger a response from the system. This can be seen in every graph presented, since there is no action otherwise.

The first control mechanism, proportional action, will apply a scalar to accelerate or decelerate the response. As previously explained, if this scalar is too large, meaning there is too much gain, the response will be so rapid that it will overshoot / undershoot causing a chain of consecutive corrections around the set value. This is known as *ringing* and is displayed in the following graph 2.17c, that depicts the proportional control behavior. Every proportional controller has a point at which it starts ringing, depending on other factors, such as the transfer function and the value of the scalar it is set to. Although if

[4]Other structures are referenced in future sections2.7

this parameter is far too small, the response will be lethargic, which may not be desired, as demonstrated in graph 2.17a.



(a) (b) (c)

Figure 2.17: Proportional control with: a) low gain, b) average gain, c) high gain

The model used for this exemplification considers that the derivative and integral components are done in parallel. Therefore, we could consider one of the latest two as being active, deriving two subtypes of controllers from PID, these being Proportional Integral (PI) and Proportional Derivative (PD) controllers. Each has their own pros and cons, as discussed next. Either way, if it is decided that omitting one is beneficial for the performance and correctness of the PID controller, the programmer of said controller can set the component to 0, effectively removing that component's behavior from the system.

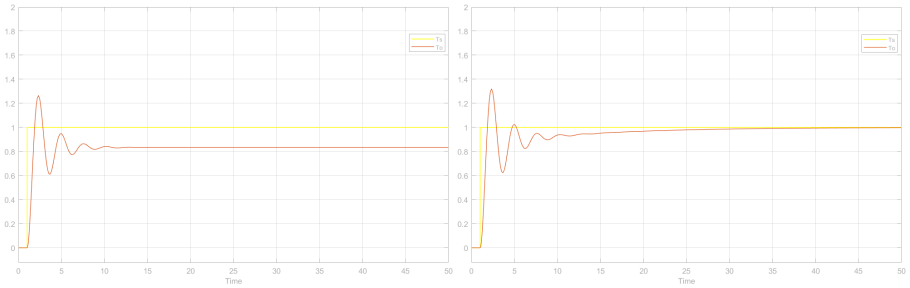
2.5.1 PI Controller

As referred to before, the integrator component is useful to prevent the steady-state error that the proportional gain suffers from. The effect can be seen in the graphs below, where the time axis was stretched in order to fully see that the steady-state error was present. This problem was solved by introducing the Integrator component, which slowly pushed the control error to zero. The issue of overshooting is still present, to which the integral component is not useful, it can even contribute, if windup is present and not accounted for.

As can be seen in the Figure 2.18b, this specific controller may be useful in cases where speed of response is not highly valued. The stabilization at the set value is not rapid and may take some time to reach it, even though the proportional gain speeds up the process, so the response is much faster than if there was only the Integrator component acting on the process variable[30]. It is also important that the system using a PI controller is not bothered by an initial overshoot, as long as the final value of the process variable stabilizes at the set point. In these cases, the PI is preferable to the PID controller, as it is cheaper and less complex to set up and manage.

2.5.2 PD Controller

The stability and overshoot that happen when using a proportional controller can be mitigated by using a term that is proportional to the derivative of the control error. Where said proportion is the damping factor as previously written. A benefit of using this controller with a proportional controller is that it allows the latter to have higher gains



(a) (b)

Figure 2.18: Proportional control with: a) no integral control added, b) integral control added, showing that it removes steady-state error

since it will compensate for the overshoot this approach would generate [31]. This can be seen in Figure 2.19. It can also be seen that derivative control cannot resolve steady-state error, as the process variable does not stabilize at the set point. Therefore, the system which decides to use a PD controller must be aware of this and deem it right to use.

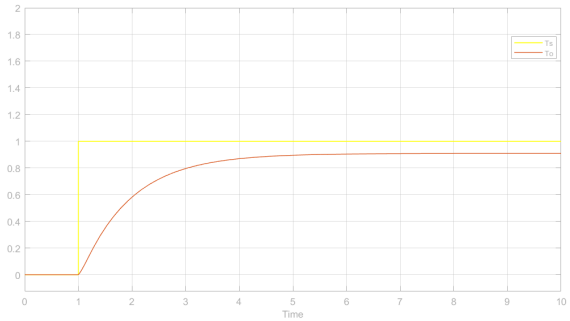


Figure 2.19: Derivative control added to proportional, showing that the overshoot and ringing instabilities are corrected

Even though the PD controller is able to set a stable temperature, it cannot be used for this thesis due to the fact that it possesses steady-state error. The exact temperatures that must be achieved in the device would be off the mark and invalidate the assay.

Finally, having all these types of controllers analyzed, it is interesting to see the result from combining them all together. Having proportional gain accelerate the response and integrator and derivative control correct the steady-state error and stability / overshoot respectively, it is possible to design a system where the response is rapid and there are no instabilities or inaccuracies. This is illustrated in Figure 2.20, which shows there is no overshooting, no steady-state error, or any other problems associated with the individual components that constitute the PID controller.

This is not absolute. The example depicts a favorable scenario. If poorly conceived, the process variable will show inaccuracies, which may be caused by incorrectly parameterizing one or more individual controllers. For example, if windup is present in integrator control, the process variable may still overshoot, even with derivative control present.

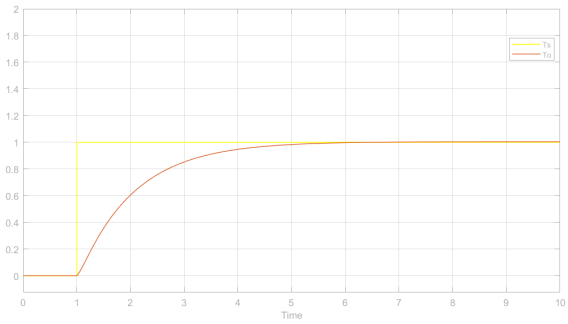


Figure 2.20: Proportional Integral Derivative control structure that was considered

The PID controller, if set correctly, is usable in the context of the device being developed. Even if the configuration needs to have a rapid response, meaning the proportional gain needs to be high, the integral and derivative components should be able to accommodate such a restriction. Of course this is just in theory, and in further sections the practical application of this knowledge is going to be tested.

2.6 Tuning a PID Controller

Correct tuning of PID controllers is essential for them to exert the user's desired behavior. Done incorrectly, and the negative aspects of one or more of the controllers will affect the outcome. Obviously, this is to be avoided, and that is achieved with the correct tuning of the PID controller. In this chapter, several mechanisms of tuning are presented for PID, in order to analyze and compare them.

These mechanisms are only for PID. There are tuning procedures for PI and PD controllers.

Since these tuning methods are for PID controllers, all of them have to be preventive of the PID nuances and problems, displayed in table 2.1 adapted from [32]. As it has been said throughout this document, this table is relative to the structure of the PID used up until now. Other structures have differences in the behaviors.

Table 2.1: PID controller components relevant characteristics and responses to changes in settings where Kp, Ki, Kd, represent the multipliers applied to Proportional, Integral and Derivative control respectively

Closed-Loop Response	Rise Time	Overshoot	Settling Time	Steady-State Error	Stability
Increasing K_P	Decrease	Increase	Small Increase	Decrease	Degrade
Increasing K_I	Decrease	Increase (Windup)	Increase	Decrease (Zero)	Degrade
Increasing K_D	Increase	Decrease	Increase	No Change	Improve

2.6.1 Manual Tuning Method

Manually tuning the PID controller is seen as a valid method and is still done on many occasions. This process is of trial and error until a viable model with the correct parameters is obtained.

Many sources utilize different forms of manually tuning the PID controller. The method discussed next revolves around raising the proportional gain first, then tuning the integral and derivative components.

To better understand the practical tuning of PID, a method based on the company TLK ENERGY[29] is utilized for demonstration. To take into consideration, the mathematical representation of PID used by said company is equivalent to the ideal form, which is the same used so far in the document. Although it uses time constants 2.6, as seen before in this chapter, these are convertible to the I and D constants that have been used so far, therefore valid conclusions can be taken from this analysis.

This tuning process is done in steps, where in each step a small change is made relative to the last step, such as slightly increasing one of the component multipliers, and observing the change in the system until the desired response from the process variable is achieved.

2.6.1.1 Proportional Component

The process begins by setting the proportional component, while the integrator and derivative components are ignored (set to 0). By beginning with a low gain, we start off passively. According to the guide[29], "A good starting point for K_P can be found by considering in which order of magnitude changes in the manipulated variable cause a change in the controlled variable. You then take a fraction of this value". This ensures the process begins rather slowly and, as the value is increased, the changes in response time are gradual and controlled, since there is a reference point where the process started.

In proportional control, the process variable will never hit the set point exactly. Firstly, the response will be slow and very off target. As the gain is increased gradually, the response time will also increase at a proportional pace, although it will also begin to oscillate. As the oscillation gets aggressive and ringing begins to be noticed in the response, it is a clear sign that the proportional gain was overdone, and it must be toned down. According to the source mentioned, a good set point for proportional gain response is when it begins to oscillate, but it rapidly settles down and stabilizes. This proportional gain value will be kept throughout the rest of the tuning process.

It is possible to parameterize the proportional gain higher than the set point mentioned, as long as there is compensation in the derivative component, since this will cure the extra oscillation introduced by the overly optimistic proportional set up. Adapted from Ellis G et al.'s book, the proportional gain can be raised by 25% to 50% over the value from the Proportional (P) and PI controllers[31].

2.6.1.2 Integrator Component

Following a proportional gain set up, that ensures a fast response, the objective is to define the integral component, that will eliminate steady-state error. The TLK ENERGY guide[29] utilizes integration time with a high initial value, which, translating to the paradigm of the model used in this document, corresponds to a low value of the **I** multiplier (integral gain) of the Integrator component.

When tuning Integral gain, it will be gradually increased with each attempt, taking care not to overshoot and cause system instabilities.When the system begins to overshoot but rapidly stabilizes at the set point, it indicates the integral gain is well-balanced between response time and stabilization. The overshoot, if problematic for the system, can be handled with the derivative component. Therefore, the integral gain can be set to a higher value, much like the proportional gain.

2.6.1.3 Derivative Component

Finally, the Derivative component is tuned. Up until this point, the system slightly overshoots, then stabilizes at the set point. What the derivative component brings is the stabilization of this overshooting, which allowed the previous components to be optimistically set to higher than normal values. According to Ellis G et al. "The expectation is that the P and I gains will be about 20%-40% higher than they were in the PI controller"[31].

As usual until now, the process begins by setting the Derivative damping factor to very low, in the guide $\frac{1}{10}$ if the integration time was used. Going by example, the damping factor could start as $\frac{1}{10}$ of the Integrator gain, gradually increasing until the response of the process variable is as desired. If the damping factor is too small, there will still be oscillations in the response. On the other hand, if the factor is too large, the system gets completely destabilized.

2.6.2 Ziegler–Nichols Tuning Method

The Ziegler-Nichols method for tuning controllers requires fewer observation steps and uses the proportional gain as the basis for the remainder components.

Following the instruction from Peter Woolf et al. book [33], the process begins by setting the set point, T_S to a typical value the system would reach, while turning off (setting to zero) the Derivative and Integral actions. Next, the proportional gain is set to a minimum value and progressively increased, until the system presents ringing or oscillations that are self-sustaining.

The gain at this point is known as maximum or ultimate gain and is here represented by P_{MAX}, it is also important to note the period of the oscillations, t_u. Some models use the frequency, which is equal to $\frac{1}{\text{Period}}$, so there is no real difference, but it is important to pay attention to this detail, as they are not directly the same.

Using the 2 values noted from before, it is just a matter of setting the parameters of each component as defined next:

Table 2.2: Ziegler-Nichols Method based on [33]

	P	I	D
P-Control	$\frac{P_{MAX}}{2}$	0	0
PI-Control	$\frac{P_{MAX}}{2.2}$	$\frac{t_u}{1.2}$	0
PID-Control	$\frac{P_{MAX}}{1.7}$	$\frac{t_u}{2}$	$\frac{t_u}{8}$

This method of tuning only works for plants that can be rendered unstable with proportional control, since overshoot and oscillations are necessary to calculate the two values, P_{MAX} and t_u, used for defining each of the PID's component parameters.

Since this is a very old method that has been thoroughly analyzed over the years, many, if not all, the different possibilities of system configurations and tuning nuances have been analyzed and are available online, making it a proven tuning method that has been widely used up until the present day for its ease of use and robustness.

There are plenty more tuning mechanisms that are currently being used by the industry. These may be applicable in specific situations or for some form of PID control that requires said tuning mechanism. Ziegler-Nichols was described here to give an idea of the type of tuning mechanism other than manual tuning.

The current DVP device already utilizes a PID controller that was manually tuned and has great performance in terms of temperature step response, therefore a similar method might be used in the context of this thesis. The use of some software for visual confirmation of the tuning process could be used, and there is room for experimentation with different tuning mechanisms.

2.7 Different variants of PID controller

So far, the structure of the PID controller considered followed an ideal form, but there are two more structures commonly considered. These forms perform equally well when properly tuned. In practical means, it is hard to find the perfect form as each have their drawbacks.

Many authors such as Mona Amin et al. [34] and Jacques Smuts, author of the book Process Control for Practitioners [35] analyze and put these variants to practice, although each different individual may utilize different nomenclatures for each variant, an example of this is that Mona Amin et al. refers to the Series form of the PID controller as it is, while Jacques Smuts, in its website [36], where he summarizes some information from his book, refers to this same form as the Interactive arrangement.

2.7.1 Ideal PID controller

The ideal form of the PID controller is widely used in modern industry. It follows the following equation:

$$u = K_C\left(e + \frac{1}{T_I}\int edt + T_D\frac{de}{dt}\right) \tag{2.7}$$

In this form, the proportional gain is also known as controller gain since this type of gain affects all the components, Proportional, Integral and Derivative and is normally represented by K_C

2.7.2 Serial PID controller

The Series form of the PID controller follows the equation:

$$u = K_C\left(e + \frac{1}{T_I}\int edt\right)\left(1 + T_D\frac{de}{dt}\right) \tag{2.8}$$

It also uses the controller gain notion since it also affects the three PID components.

If the derivative action is set to 0, then the ideal and series form are identical. Many times, this component can be omitted, so these two forms of the PID controller end up being the same.

2.7.3 Parallel PID controller

The parallel has all the components processing the control error independently before summing the result of each. It follows the equation:

$$u = K_P e + K_I\int edt + K_D\frac{de}{dt} \tag{2.9}$$

It has no controller gain, only proportional gain, due to it not being applied to all the components. This controller is easy to understand as all the parts are independent of one another. According to some authors, it may be more confusing to tune since there is no controller gain, although since the three actions are decoupled, it is easier to set up this form of the controller [37]. According to the last source, which is on the industry side of things, the ideal and parallel are the most used.

2.8 Temperature Control Conclusion

Temperature control, namely PID, was extensively discussed throughout the previous sections, laying a foundation to what is going to be one of the main tools used in this thesis. Since the thesis project requires new settings of temperature, it is important to have solid knowledge of the subject. Each part of the PID plays an important role in obtaining the most accurate control possible, allowing the pocket device to execute new types of instructions that up until the completion of this thesis would not be possible.

Taking into consideration the extensive research available on this subject it was evident that PID controller was the best solution to achieve the objectives of this project, applying all the previous knowledge in tuning the PID controller to suit the pocket device needs will, in theory, simplify the firmware development process.

2.9 DoctorVida Mobile Application

This section serves the purpose of succinctly presenting the application, since it will be subject to alterations as the firmware of the pocket changes. The screens presented in this section represent what a user would find in the initial version of the application. Although there were changes, the order of screens will not change, nor will the visual presentation. They will be presented in the order they would be shown in a normal assay use case.

2.9.1 Home screen

The home screen gives the user several options, Figure 2.21, in order:

- Pairing with a DoctorVida device which is the first thing the user must do, therefore makes sense it is displayed firstly.
- Start a new assay using the tutorial option, which will instruct the user along the process of introducing assay information
- Start a new assay using the fast track option, which only requires the user to scan qr codes that are present in the different elements needed for an assay, such as the sample tube and DoctorVida device.
- Resume running assays gives the user the possibility of returning to a particular assay, since it is possible to have 4 running at the same time. It also allows the user to monitor the assay progress.
- Available results stores and displays the previous results the user obtained.

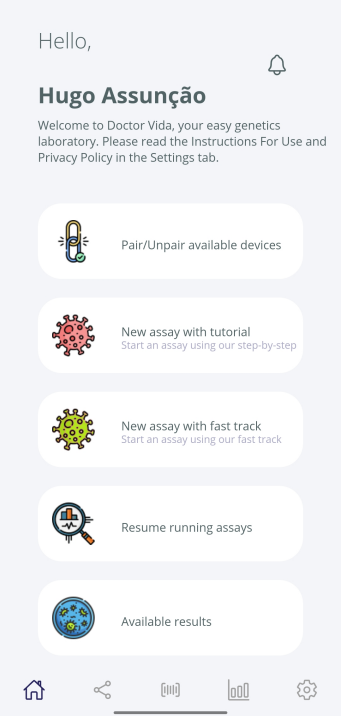


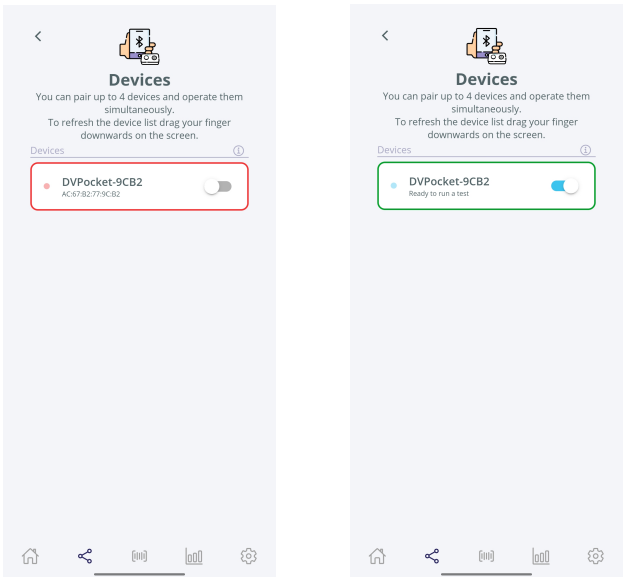
Figure 2.21: DoctorVida application home screen

2.9.2 Bluetooth Pairing

Before commencing any type of communication between application and device, both of them must be paired via Bluetooth. The screen for this, Figure 2.22a, lists all the devices in the vicinity of the cellphone, whether connected or not, allowing the user to manage the DVP devices through the application. Once the user pairs with the devices he wants, they will be highlighted from the rest, Figure 2.22b.

2.9.3 New Assay

To create a new assay, one must introduce 3 pieces of information: the desired device, the sample information, and the protocol information. It is possible to do this with the tutorial mode, where every step has some assisting text to guide the user through the process. The alternative is to introduce this information using the fast track mode, where it is only needed to scan the QR codes for the respective elements, as seen in Figure 2.23.



(a) (b)

Figure 2.22: Application Screens: a) DoctorVida application device screen pre pairing, b) DoctorVida application device screen post pairing.

2.9.4 Protocol Transmission

The protocol transmission to the pocket device occurs upon obtaining said protocol from the API. This is done upon introducing the protocol information or scanning the respective QR code in the new assay screens. Once the protocol is obtained and the user decides to begin the assay, it will submit it via Bluetooth to the pocket device, using a BLE characteristic specifically designed for this purpose, as explained in section 2.2.2.6.

2.9.5 Assay Preparation

Before the actual assay starts, the DVP does some verifications, such as if the temperature is correct, if there is no previous tube still in the chamber, Figure 2.24a, and if a new tube is present in the chamber before the assay starts, Figure 2.24b.

2.9.6 Segment Transmission

During the assay progress, the application periodically requests the sensor data segments stored in the device to be presented in a graphic, Figure 2.25a and Figure 2.25b. These segments are also transmitted to the backend and stored in a database. The period of requesting the segments is defined in the protocol. Requesting the segments also uses a specific BLE characteristic, specially designed to return the segment with the given segment number.

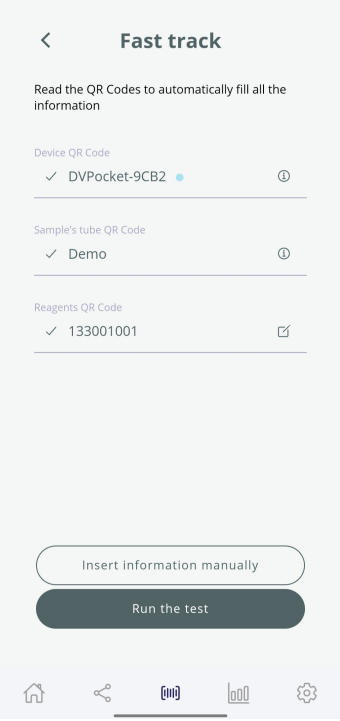


Figure 2.23: DoctorVida application fast track page already filled with assay details

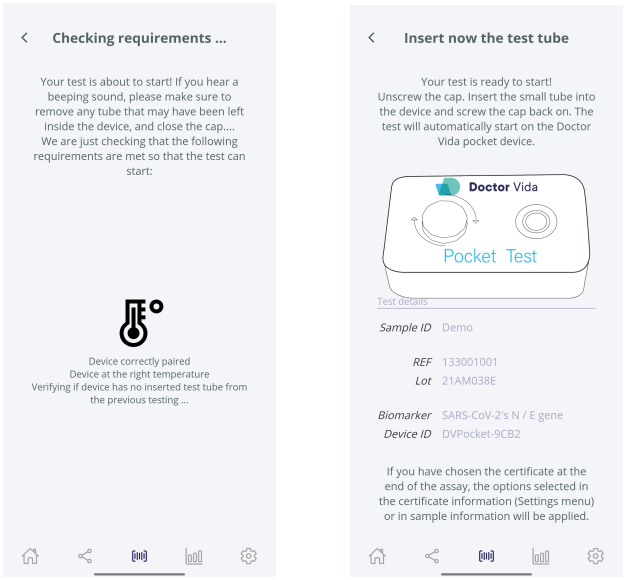
2.9.7 Pocket Reset

Finally, even after the assay is completed, the application may not have all of the segments due to Bluetooth disconnection or other issues that may affect data transmission. The pocket device firmware was designed in such a way that it will wait after completing an assay for the ending segment that resets the device back to the initial state. During this period between ending the assay and receiving the ending segment request, it will answer any request for any segment it has stored in memory, allowing the application to get any missing segments, Figure 2.26a. Afterwards, the memory is cleaned, and the pocket is prepared for the next assay.

By the end of the assay, the user has access to the assay result, which in the initial app indicated if the subject was positive or not for COVID-19, Figure 2.26b.

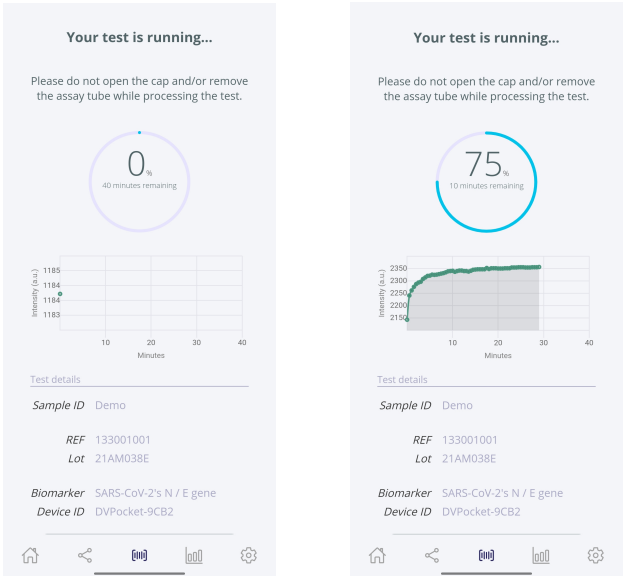
2.10 Conclusion

To conclude the state of the art chapter, every section had a relevant topic for the development of this thesis. The firmware played the dominant part in the development phase, where most of the work and time was spent. This includes tuning the current temperature controller as well as creating new firmware states that allow the usage of different instructions with different temperature settings.



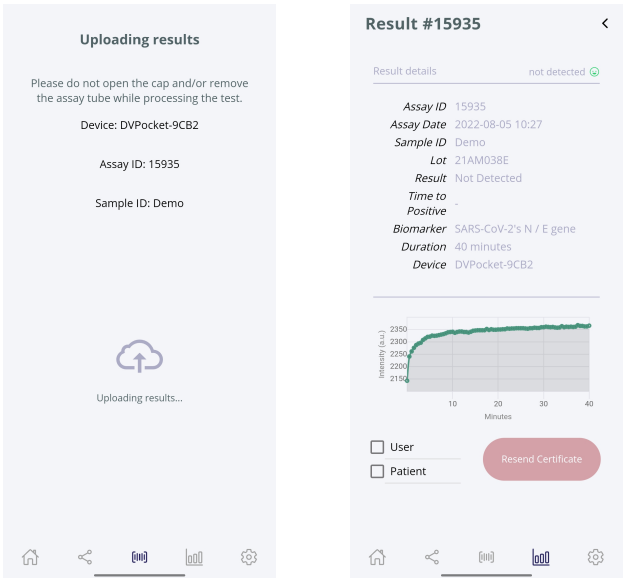
(a) (b)

Figure 2.24: Application Screens: a) DoctorVida application assay pre requirements, b) DoctorVida application screen for waiting the tube.



(a) (b)

Figure 2.25: Application Screens: a) DoctorVida application assay progress at the beginning, b) DoctorVida application assay with further progress.



(a) (b)

Figure 2.26: Application Screens: a) DoctorVida application page while there are segments still being uploaded, b) DoctorVida application for displaying the assay result.

The application also consumed a large portion of the thesis time since it is the connection between the user and the pocket device. Therefore, it needed changes to adapt to the device firmware upgrades.

All these components play a part in completing a LAMP assay and obtaining valid results, while continuing to be able to execute any type of previous protocol, such as the COVID-19 one.

3

Proposal

STAB VIDA proposed several subjects for the making of this thesis, most were kept, some were abandoned or delayed for a future project and there were also new subjects added to the previously proposed ones. These section will discuss these topics of interest.

The majority of the time was to be spent working on the device's firmware, and the rest on the software to work with the device, including a python script and a mobile application.

3.1 General Idea

To give a better idea of the subjects the general schema depicted in Figure 3.1 presents the components approached during the development of this thesis; this schema will be used for reference throughout the thesis document. It includes the firmware and its various components, the application with its main screens and functionalities, and the backend / API, which, even though it was not the subject of this thesis, makes sense to include for a better understanding of the connection between the different elements.

This schema does not include the Python script as this was not supposed to be included in the final product for the public and was only used internally by STAB VIDA.

The firmware had well-defined goals that were proposed when the thesis was first introduced. These included the development of an algorithm capable of producing steep heating ramps and configuring the PID controller to achieve this, mechanisms to mitigate errors caused by the device optic hardware, communication via Bluetooth with an application, and implementation of the firmware for a cooling system. Throughout the development, some of the attention had to be shifted to different objectives as new opportunities emerged.

The application component of this thesis was not clear at the early stages, but it was certain it would need changes to accommodate the new firmware of the DoctorVida device. At first, the only goal was to change the application. However, these goals became clearer as time passed, and they will be shown in the next sections.

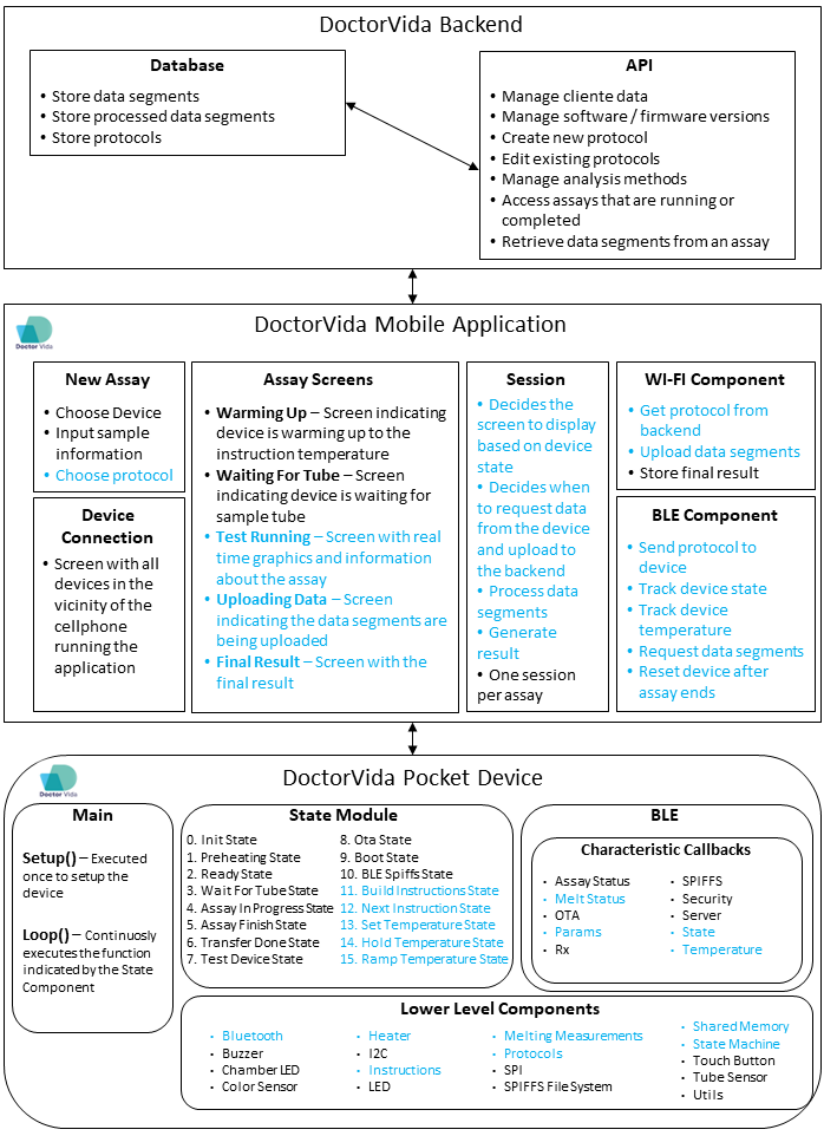


Figure 3.1: General schema of the firmware, application and backend. In blue colored font are the components that were worked on for the context of this thesis.

Therefore, the proposal for this thesis is composed of several parts. The topics are divided into 2 main ones: the firmware of the device and the software, both the application and a python script that proved necessary for working with the new firmware while there was no application ready.

3.2 Firmware

The firmware occupied most of the thesis duration. The initial objectives were presented before, but, as said, these suffered many changes. The reasons for this will be discussed

in future chapters, but mainly they were related to new challenges that emerged when developing the firmware, which STAB VIDA prioritized. Moreover, as the thesis went on, the initial goals became more specific and were split up into different parts of the firmware, so it makes sense to list them here.

The new objectives that were proposed are the following:

- Change the format of the data the pocket device stores throughout the assay — This was changed to accommodate the extra channel reading by the fluorescence sensor. Up until the development of the firmware, the device could only light up a LED and read all six channels. The device can now turn on both LEDs but only read one channel. The reason for this will be explained in section 4.6. But in short, it had to do with how much memory the device had, which in some cases was not enough for the test.

- Tuning of the temperature controller, namely the PID feedback controller — New PID settings were tested to see if they improved the performance. This will be discussed more in section 4.1.

- Create a procedure in the device capable of receiving a new protocol format and parse it into a set of instructions that the pocket then follows.

- Allocate memory in advance for the purpose of storing this set of instructions as well as the data segments gathered throughout the assay.

- Create a procedure to correctly read the fluorescence sensor even in short periods of time.

- Create all the necessary Bluetooth elements to support the execution of an assay — This includes adding new characteristics to the Bluetooth service and creating the callbacks used by these characteristics, such as to respond to requests for data segments, return the current state, receive the protocol, give information about the device, like its mac address, and return the current temperature of the device.

- Build upon the existing state and add new states to build the instructions, set the correct temperature, and execute the instructions. These states are extra to what already existed, so the device is backward compatible with the previous mobile application and protocol.

Looking at what is listed above, there were major changes in the topics initially proposed. Some of the objectives were kept and made more specific, while others, like the mechanism to fix errors caused by the device's optical hardware, were taken out because they could not be achieved in the time of developing the thesis as was initially thought or they lost relevance as STAB VIDA set new objectives for the device.

In order to complete these firmware objectives, there had to be initial tests to check if the heating component (section 2.2.6) was able to maintain the steady temperature increase necessary to execute a LAMP assay. During this period of testing the heating ramps, many configurations of the PID controller were applied to see how they affected the temperature profile.

Past this phase, it was possible to start doing actual work on the firmware. The first priority was to be able to receive the protocol via Bluetooth and providing the device with a way to parse the protocol into a format that it could follow to complete the assay. The protocol also allowed the pocket to calculate the total amount of memory it needed to allocate straight from the get-go. From this point, the pocket could start the actual assay.

The plan was to create three states that together can execute any type of assay, the **SetTemperatureState**, **HoldTemperatureState** and **RampTemperatureState**. The first state will set the initial temperature of the next instruction; the second state will hold a temperature for a given amount of time; and the last state will go from an initial temperature to a target temperature. This will be thoroughly explained in the next chapter.

Once an assay is completed, the device waits for confirmation that the application has all the data segments, after which it clears all the memory allocated and returns to the initial state, ready for the next assay.

Finally, the only major thing left on the firmware side was to build upon the existing Bluetooth infrastructure to allow a client to access the information generated by the device. This includes being able to get to the new data segments and the temperature of the pocket device, both of which have their own characteristic.

3.3 Software

The software side of the thesis started from only changing the application to be able to work with the device to creating a python script and changing the application in a major way.

It made sense to separate the objectives of the Python script and mobile application into two different sections since they were two completely independent parts.

3.3.1 Python Scripts

Initially, the Python script was designed to send the protocol and start an assay on the device as a way of testing the device functionality. Eventually, due to a situation where the new device had to be presented to a STAB VIDA partner, the script had to be developed even further and ended up being able to do real-time plotting, compute the final result of the assay, and generate a PDF report that was sent to the patient's email. This will be discussed further in the section 5.1.

To accomplish the goals mentioned above, the proposal was that the script be divided into different sections that were approached incrementally.

The initial focus was on being able to send the protocol so the device could kick-start the process. This was the only purpose the script had for a while.

Afterwards, the script was developed even further to be able to store assay data segments in a CSV file. The need for constant communication between the pocket device and the Python script to manage the states and progress of the device made this the most difficult and time-consuming stage to complete. This was thought to be the most important part of the Python script because it was enough to finish an experiment and collect the data needed to do an outside analysis.

With the previous script completed, it was possible to build upon it. The next step was to plot the data segments in real-time as they were received from the device. Even though it seemed easy at first, this task needed extra care because the data had to be plotted constantly. It was necessary to add multiprocessing to the script so that the plotting task was executed in parallel. Every time a new instruction begins, the plot clears and starts from zero. This made it easier for a user to follow the progress of the assay.

Once the assay is completed, all that is left is to analyze the data and submit the results to the user. The analysis used an algorithm created by STAB VIDA. A peak detection algorithm had to be used to be able to calculate fluorescence peaks in order to determine the genotype of the sample. This will be further explained in the section 5.1.3.

Finally, with the results calculated, a PDF was generated and sent to the user. The PDF format had a certain layout and included both the plots seen during the real-time plotting phase and the final analysis result. The positioning of each PDF element had to be manually configured, using coordinates to place every component in a specific spot. Afterwards, this PDF was sent to the user via email, using the email address given as an input to the script.

Once all the steps were completed, the script had to reset the pocket device and then it could end.

3.3.2 Application

The application objectives also got more specific and complex compared to what was initially predicted to happen. The application by the end of the thesis was capable of displaying the real-time plots of the data segments, submitting the data collected from the device to the backend, calculating the result of the assay, and storing the assay results and processed data[1] in the backend for future access.

The application development started off with creating the functions to send the assay protocol to the device and receive data segments from the device that would be later stored in a file in the device's file system. This was done using the application's Bluetooth service, which allows you to subscribe to some characteristic notifications or directly

[1] For calculating the result, the data has to be processed in a specific way that will be discussed in section 5.2

request the value of a characteristic. This way, it was possible to use the application to operate the DVP even if the analysis had to be done externally.

Now that the application had access to the data segments, it could display the real-time plots much like the Python script did. So this was the next step. Create a screen that could display each instruction's real-time graphic. This screen would be on display for the majority of the assay duration, therefore it was given the most attention. Various details were introduced, such as a progress bar indicating the progress of each instruction and of the total assay, the state of each instruction, since they were executed sequentially, it could be **Finished**, **Running**, **Waiting** and the temperature progress between instructions, due to the pocket having to change temperature between them. This concluded the development of this screen, which was the one that required the most effort. But this was only gathering and displaying data. Once the assay ends, the app must process the data and generate a valid result for the user.

For this it was necessary to guarantee that the application had all the data segments, hence synchronization with the pocket device was added after the assay ended. Another important mechanism implemented was the resuming of an assay. If by any means, the application suffered any type of disruption, it had to be able to recover and continue the test. This requires reconnecting to the device, checking what segment it is currently on, and checking if it has every segment the pocket device possesses. In a way, it requires synchronizing with the pocket device. There are many causes of disruption, such as WI-FI signal loss, Bluetooth out of reach, application shutdowns, crashes, etc.

Once an assay is finished and the outcome is generated, the application can conclude the assay process and the result is stored in the backend along with the raw and processed data segments. This way, a user can always access the analysis later and the data segments are available if a new analysis is required.

4

Firmware

This chapter will present the work done in the firmware. The device had many areas that needed attention in order to make sure that it could execute the protocols that were planned.

The initial concerns were to check if the device was capable of producing the temperature ramps that were planned, while still being able to keep a set temperature with fairly accurate results. It was known that holding a temperature was no problem, since the COVID-19 protocol was basically holding 65ºC for 40 minutes. Along with this, there were also memory capacity and sensor reading time concerns.

This chapter will talk about these problems and how they were handled, as well as the other interesting parts of the firmware of the DVP device.

4.1 Temperature Control Tuning

Temperature tuning was the first subject to be given attention. This controller used PID feedback control and the tuning used both manual and Ziegler-Nichols tuning, to decide what would be better.

Although these methods are explained in section 2.6, in this case there were some nuances. These have to do with the way the sensor works and the performance limitations inherent to the device.

4.1.1 Rate of heating limitation

On the topic of heating, the main performance setback was the rate of heating the device was capable of. While the laboratory machinery could increment at rates that could reach 4ºC/s, the pocket was limited to at most 0.2ºC/s at lower temperatures and even less at higher temperatures(Figure 4.1), which could hinder the ability of the device to execute some of the assays the laboratory machine is capable of running.

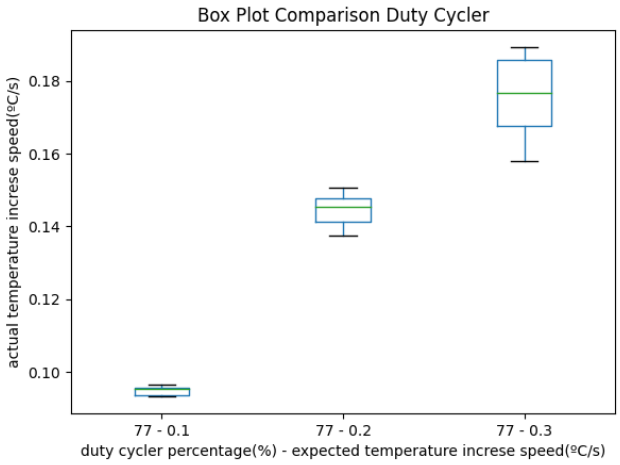


Figure 4.1: Boxplot of the temperature increase with duty cycle set to 77% maximum.

4.1.2 Duty Cycle Tuning

Before beginning tuning the PID controller it was noticed that the temperature controller configuration only allowed up to 77% duty cycle percentage.

In an attempt to further improve the heating velocity, the duty cycle percentage, explained in section 2.3.2.1, was altered from the initial version. It could initially range from 0% to 77%, but this was changed to 0% to 100%. This means that the controller could reach a constant high signal of 5 volts, which increased the rate at which the device could heat up, shown in Figure 4.2 and Figure 4.3.

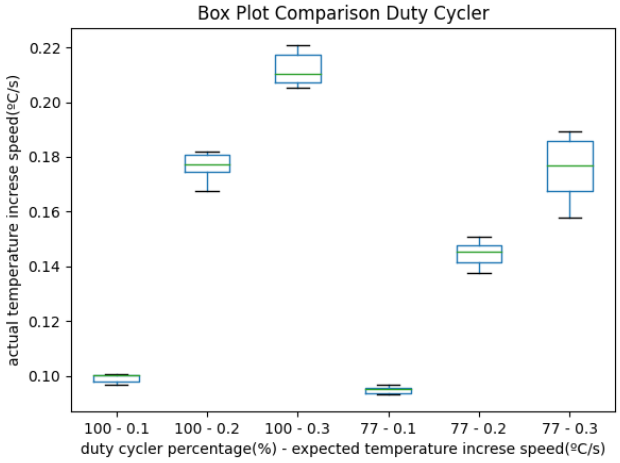


Figure 4.2: Boxplot comparing the temperature increase with duty cycle set to 100% maximum and 77% maximum.

The initial device used 77% maximum duty cycle percentage rather than 100% mainly to save energy. At the time, there was no need for a controlled temperature increase. The

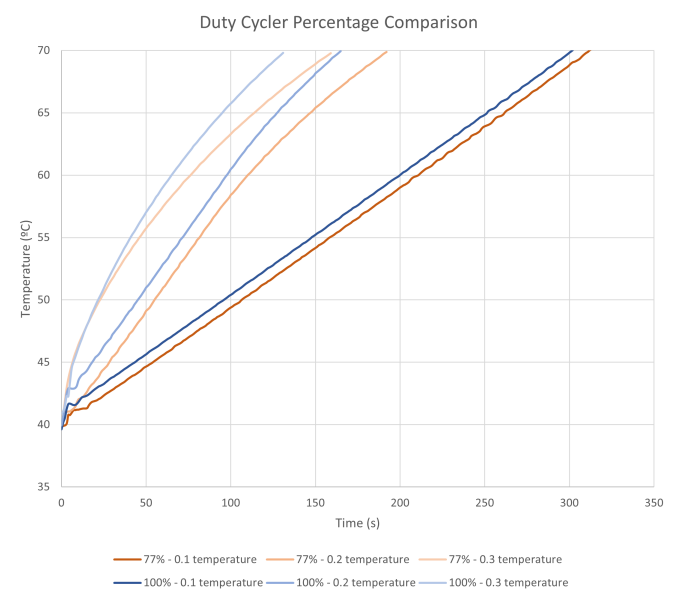


Figure 4.3: Comparison of the curves for 77% and 100% Duty Cycle.

COVID-19 protocol only required the device to stay at 65ºC for the duration of the assay. The time it took to reach this temperature was irrelevant; it only had to be done once. Afterwards, the device just stayed at 65ºC until it was shut down.

With this change to the duty cycle, the DVP device was able to reach 0.2ºC/s and if the ramp was at a lower temperature, it could even reach 0.3ºC/s, Figure 4.3, but it is clear that it was at its limits and a larger rate of heating would yield inaccurate results, especially at higher temperatures.

4.1.3 PID Tuning

Following the duty cycle correction, the idea to resolve the rate of heating shortcoming was to alter the PID component values in order to obtain a steeper heating ramp. As previously said, both manual and Ziegler-Nichols tuning were used.

Both of these methods started with the same step. This was to steadily increase the proportional gain until ringing was achieved. The results are presented in Figure 4.4.

4.1.3.1 Manual Tuning

The problem with proportional gains that do not cause ringing is that they suffer from steady-state error, where the temperature of the device never reaches the desired temperature. This is solved by using the integral component of the PID, as explained in section 2.4.4.4.

After trying several proportional gains, the next step was to tune the integral component. For this, the first proportional gain used was 1000, since it reached the highest

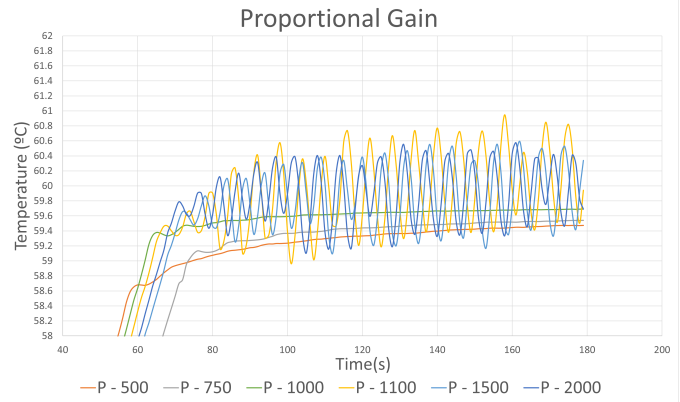


Figure 4.4: Result of several different proportional gain values.

temperature faster without ringing. Using this gain, the integral gain was set to several values for comparison, Figure 4.5.

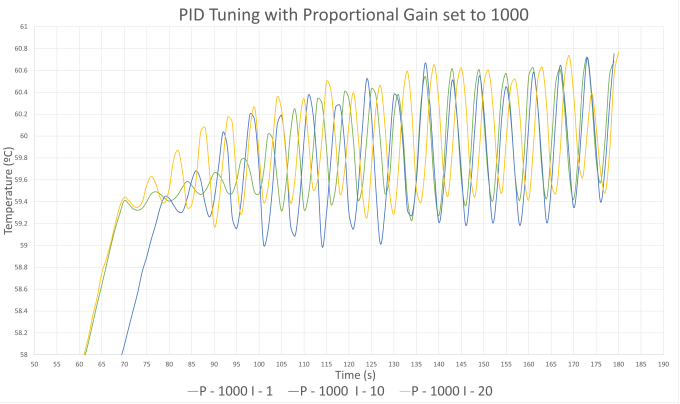


Figure 4.5: Tuning of the integral component with proportional gain set to 1000.

The increase in the integral gain resulted in severe ringing once again, reaching temperature differences of over one degree, which is expected when integral gain is raised too much. That was not the case, since even the value of 1 caused this behavior. In an attempt to prevent the ringing caused by using the integral component, a lower proportional gain was used. This new setting made use of proportional gain set to 750, Figure 4.6.

Knowing the issues caused by the integral component in the previous proportional gain setting, this time smaller and finer values were used. What was obtained was either the temperature not reaching the predefined value in time, meaning the integral component was ineffective, or severe ringing once again.

Finally the proportional gain was set to 500 and once again integral control was added, Figure 4.7.

[1] This plot might give the idea that the lower integral gain was faster but slowed down earlier. The cause of this is due to the initial temperature not being the same for both curves, since the device has a margin of ±1ºC from the ramp start temperature.

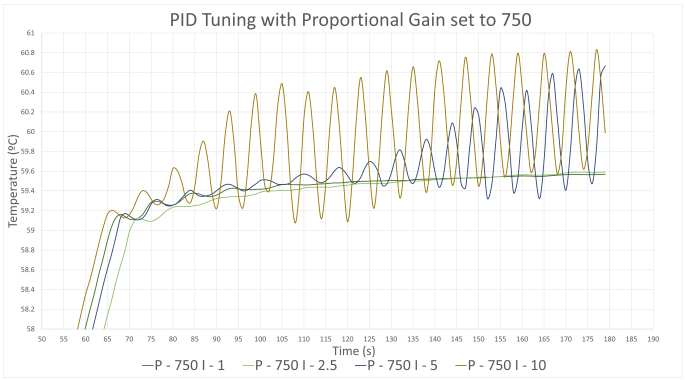


Figure 4.6: Tuning of the integral component with proportional gain set to 750.

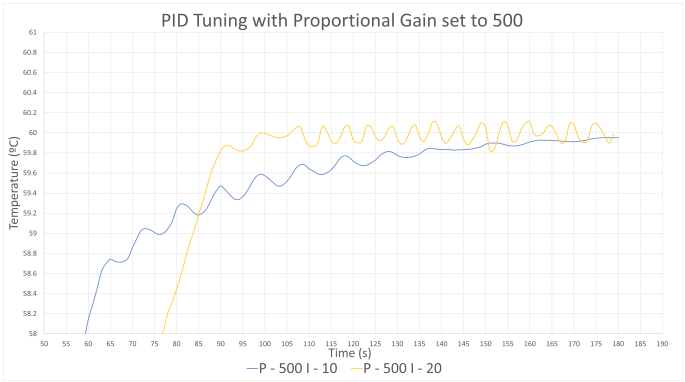


Figure 4.7: Tuning of the integral component with proportional gain set to 500[1].

With these settings, more reasonable behavior was achieved. While using integral gain with a value of 10 still displayed slowness when reaching the desired temperature, the gain set to 20 reached the desired temperature faster and resulted in little ringing, under 0.2 degrees difference.

Ringing is bound to happen once the temperature is reached, independent of the PID setting. This is caused by external factors such as variations in the room temperature, residual temperature in the device casing, etc. This affects the internal temperature of the device, which as a response tries to correct the difference, resulting in some ringing. The problem emerges when the already present ringing is amplified by that of the PID components. Hence, the setting mentioned before as acceptable is the one that least influences the inherent ringing of the system.

The best setting used for the PID controller of the device was achieved without using the derivative component, hence it was not used. This component was used throughout the manual tuning process (Figure 4.8), but it was not always having the desired results.

The fact that this setting was acceptable was known prior to executing the tuning test. This is because this was what the initial device used for the COVID-19 test firmware, which only needed to hold a temperature. The initial device had already been subjected to extensive tuning and testing, so it was not unexpected that the PID setting inherited

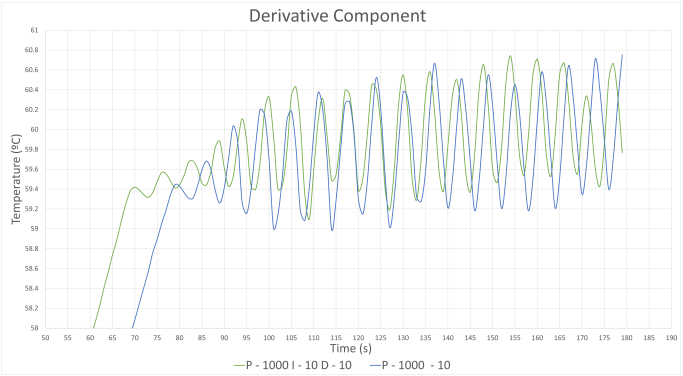


Figure 4.8: Addition of the derivative control to the tuning of the PID controller with lackluster effects.

was already properly tuned.

4.1.3.2 Ziegler-Nichols Tuning

Before deciding the final PID settings, one final tuning methodology was attempted. This was the Ziegler-Nichols tuning procedure, explained in section 2.6.2.

For this, the first step was to decide the proportional gain. The value of 1100 was the smallest value that presented ringing. Afterwards, the period of the wave, after the ringing commenced, was calculated and was approximately 6. Using the Table 2.2, it is possible to obtain both a PI and PID controller.

The PI controller settings would be:

$$\begin{aligned} &\text{Proportional gain:} \\ &P = \frac{1100}{2} \\ &P = 500 \\ &\text{Integral gain:} \\ &I = \frac{6}{1.2} \\ &P = 5 \end{aligned} \tag{4.1}$$

Since this setting shared the proportional gain with the settings mentioned in the previous section, but with a smaller integral gain, it was expected that it would not be able to reach the target temperature either, and for this reason it was not experimented with.

The PID controller settings were:

Proportional gain:

$$P = \frac{1100}{1.7}$$
$$P \approx 647$$

Integral gain:

$$I = \frac{6}{2} \tag{4.2}$$
$$P = 3$$

Derivative damping factor:

$$I = \frac{6}{8}$$
$$P = 0.75$$

This was tested and the results obtained, Figure 4.9, showed that the temperature was not able to reach the target temperature fast enough. This ended up excluding the setting obtained from the Ziegler-Nichols method of tuning as a viable option for the PID controller.

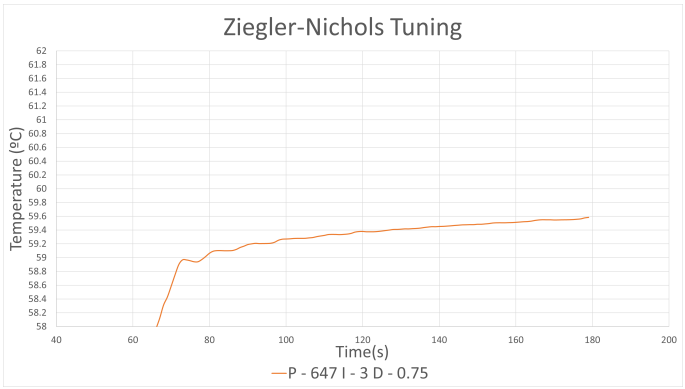


Figure 4.9: Tuning of PID using the Ziegler-Nichols method.

4.1.3.3 PID Tuning Conclusion

The new device added a new functionality in the form of temperature ramping. This was the reason for the extra tuning of the PID, in an attempt to achieve the set temperature in the fastest possible time. This means that the ringing that happened after the temperature was reached was not relevant, since by then the instruction would have already been over. This being said, the other instruction, that was meant to hold a temperature, had to stabilize at a set temperature, thus controlling the ringing was important in this case.

The middle-ground had to be achieved, and with the initial setting the best results were obtained and therefore this was kept for the rest of the firmware development.

4.1.4 Rate of heating fall-off

Along with the rate of heating problems previously mentioned, another crucial performance issue was noticed. The device's rate of heating decreased as the temperature rose. This can be seen in Figure 4.10, where the rate of heating was set to 0.3ºC per second and from 40ºC to 50ºC was around 0.3ºC/s, but after this mark the device took progressively longer to reach the same temperature increase, which explains why the calculated rates of heating throughout this section stay below what was indicated.

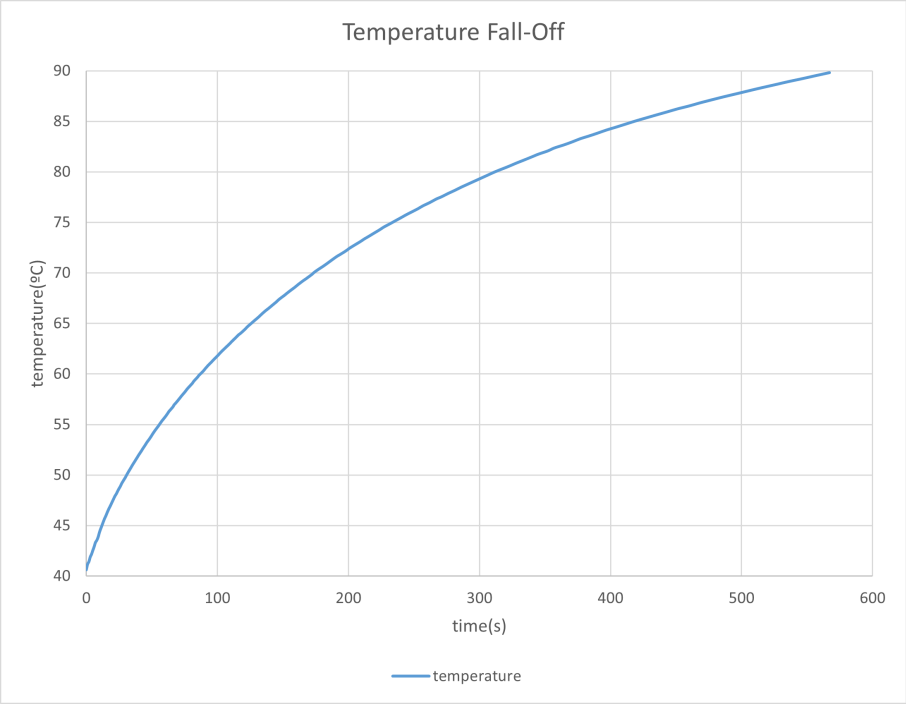


Figure 4.10: Example of rate of heating fall-off as temperature increases.

This could be misleading since, when testing, higher rates seemed to decrease the assay duration. This was true, but only for the lower part of the temperature ramp. That is, higher temperature rates worked as intended at lower temperatures but also took longer at higher temperatures. Therefore, the decrease in assay duration was only due to the lower temperature portion getting a shorter duration, because as the ramp progressed the rate of heating became similar, independent of the initial rate of heating, seen in Figure 4.3, where after the 65ºC mark this is noticeable

Some slowing down on the temperature ramp was acceptable, but too much could have changed the reaction inside the sample tube, so the new type of protocol only allowed up to a certain rate of heating.

Unfortunately, the fall-off was unavoidable since it was a physical limitation of the resistor. This was tested with and without PID to be sure that it was not limiting the signal output to the resistor. This means that without PID the resistor was receiving a constant high signal, but the results showed that with or without PID the temperature

fall-off was present (see Figure 4.11).

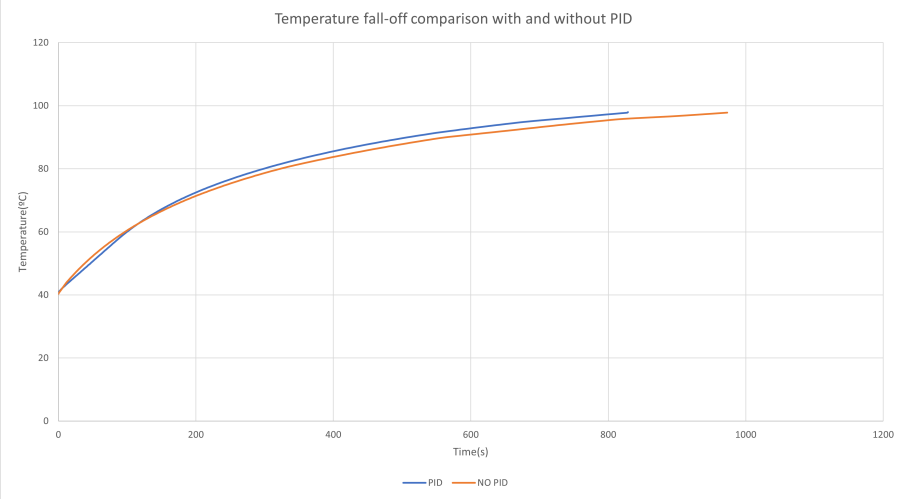


Figure 4.11: Comparison of the temperature fall-off with and without PID[2].

Having done this evaluation of the limitations of the DVP device, it was concluded that it was capable of reaching a maximum of 0.3ºC/s until the 50ºC mark and progressively slow down until eventually it reaches 0.1ºC/s at around 65ºC, based on Figure 4.10 data. Protocols that needed extreme temperatures (above 80ºC) were considered not viable to be executed on the device, since it would be pushed to the limited for a long period of time, reducing its life span and potentially generating invalid results due to the incapability of the device increasing the temperature at a relevant rate, usually at that point it would be much less than 0.1ºC/s.

Even in this scenario, the device was capable of executing a multitude of assays that complied with the limitations discussed above. It was almost guaranteed that all assays would need a temperature hold step, which presented no problem to the DVP device, even at higher temperatures, which would just increase the time to reach the necessary temperature. What needed to be evaluated is the protocol ramp requirements. If they exceeded what the device could handle, the results might have been invalid and therefore not worth the investment in reagents and the time to prepare the assay.

4.2 Bluetooth Component

During the development of the device's firmware, the Bluetooth component was altered and new features were added. These included both new characteristics added to the BLE service and changes to already existing ones.

There were 3 characteristics altered or created for the purpose of this thesis. A characteristic was only created if it was not possible to repurpose an already existing one to add

[2]The difference displayed in the graph between the two lines is not relevant since the two tests were executed on different occasions and with different conditions. The graph is only used to demonstrate the presence of temperature fall-off with and without PID.

the functionality intended. This is because the number of handles limited the number of new characteristics that could be created, explained in section 2.2.2.6, and STAB VIDA did not intend to expand the number of handles that were already defined.

4.2.1 Params Characteristic

This characteristic dealt with the receipt and storage of the protocol from the application. Previously it only had to handle the COVID-19 protocol, but with the new firmware it needed to be changed so it could detect what type of protocol was being emitted, and set a flag accordingly. This way the firmware can decide what is the correct behavior for the device.

4.2.2 Melt Status Characteristic

This was the main characteristic used throughout the melt assay. This characteristic had two main functions, emit the segments the application requested and set a flag that tells the device it can reset back to **Init State**.

If the request was not to reset the device but instead was for an actual data entry, the device would take the requested entry number, check how many more entries it had stored starting at the requested entry, and notify as many as it could starting at the requested data segment. The segments that were notified were attributed to the value of the characteristic so any connected peer could request to read it and obtain the same data segments that were notified. This was the same behavior as the characteristic for the COVID-19 procedure.

The maximum number of data segments that could be emitted was 15. For this, the MTU had to be changed from the default 23 bytes to 512, but to make sure there were no problems with the packet header, only 500 bytes were considered. Given the data segment length, calculated in 4.4.1, it was decided that the device would send at most 15 data segments; this way, there was plenty of space left if future alterations were made.

4.2.3 Temperature Characteristic

The creation of a characteristic that could return the current temperature of the device made sense and was requested by STAB VIDA. Basically, during select states, such as the **Set Temperature State**, the device will notify the temperature every second. This is merely for display, giving the user a better notion of how long it will take until the desired temperature is reached.

So far only the characteristics that were created or altered to work with the melt procedure have been presented. There were also pre-existing characteristics that did not require any changes and were used in the new firmware; one example is the state characteristic, whose sole purpose was to notify of the current state.

4.3 Handle Protocol

Every assay is described in a protocol that indicated the steps in order to generate a reaction and obtain a valid result. Usually, the protocol information is found in the manual of the reagent producing company. This protocol was explained previously, but briefly, it consists of 3 steps:

1. Hold 65ºC temperature for 30 minutes
2. Hold 40ºC temperature for 10 minutes
3. Ramp the temperature from 40ºC to 80ºC

STAB VIDA after careful investigation concluded that the reaction, which takes place on the last step, always occurred before the 70ºC mark. Knowing this, the protocol was changed to only ramp from 40ºC to 70ºC, thus saving time, both heating up and cooling down for the next assay, and relieving some stress from the DVP device hardware components.

Although the lactose intolerance protocol was prioritized during development, it was never intended to be the sole protocol implemented on the new device. The protocol generation and parsing had to be dynamic, meaning that any given procedure that was meant to be executed on the device could be represented using the new protocol format, even the previous COVID-19 protocol[3].

4.3.1 Protocol Format

The design of melt protocol the device was expecting consisted of a set of instructions that could be either a **Hold Temperature** instruction, Listing 4.1 or a **Ramp Temperature** instruction, Listing 4.2, and some information about the assay. This information is stored in C++ structures(*struct*) which is a user-defined composite type to store fields that can have different types[38]. The protocol itself is a large *struct* that contains general information and 2 smaller *structs*, one for each instruction type.

This protocol was named **Melt Protocol**, since it handles any type of melting procedures that included temperature ramps. Both this protocol and the COVID-19 one were store in the Protocols component, Figure 3.1.

[3]COVID-19 protocol basically consists of holding 65ºC for 40 minutes

Listing 4.1: JSON representation of the protocol section referring to a Hold Instruction

```
{
    "order": uint8_t,
    "readFlag": bool,
    "readInterval": uint8_t,
    "holdDuration": uint32_t,
    "temperatureSetpoint": float,
    "sensorParams": {
        "analysisChannelA": uint8_t,
        "ledColorA": uint8_t,
        "sensorGainA": uint8_t,
        "integrationA": uint8_t,
        "analysisChannelB": uint8_t,
        "ledColorB": uint8_t,
        "sensorGainB": uint8_t,
        "integrationB": uint8_t,
    }
}
```

Listing 4.2: JSON representation of the protocol section referring to a Ramp Instruction

```
{
    "order": uint8_t,
    "readFlag": bool,
    "rampTemperature": float,
    "rampPeriod": uint32_t,
    "startTemperature": float,
    "targetTemperature": float,
    "sensorParams": {
        "analysisChannelA": uint8_t,
        "ledColorA": uint8_t,
        "sensorGainA": uint8_t,
        "integrationA": uint8_t,
        "analysisChannelB": uint8_t,
        "ledColorB": uint8_t,
        "sensorGainB": uint8_t,
        "integrationB": uint8_t,
    }
}
```

Each instruction in the protocol had a component called *sensorParams*, which was also a *struct*. This instructed the device on how to setup the sensors to make a reading. The

ledColor indicated what color the LED should use, either green or blue, and the *analysisChannel* indicated what wavelength the fluorescence sensor should use to make a reading. There were two LEDs that could illuminate the chamber to generate a reaction inside the sample tube, explained in section 2.2.3, therefore the protocol contained settings for both of them, thus the A and B suffix. The remaining fields, *sensorGain* and *integration*, indicated how sensitive the sensor should be and how long should the sensor take to make a measurement, much like the exposure time of a camera.

The number of instructions had to be limited to three hold and two ramp instructions due to memory limitations of the DVP device. These could come in any order, which was indicated by the *order* field, but when writing the protocol, the three hold instructions had to come prior to the two ramp instructions. This way, when sending this information to the device, it could parse the protocol, which came in bytes, into a format that he could understand and follow. This process is explained in section 4.4.

Another requirement was that the device always receives 3 hold and 2 ramp instructions since it was expecting a specific data size when parsing the protocol bytes into useful data. This posed the problem of not being able to execute fewer than the predefined number of instructions. This was resolved by adding two fields to the protocol, indicating the number of instructions of each type, informing the device that the extra ones were to be ignored. So if the field indicating the number of hold instructions was two, only the first two out of the three in the list of hold instructions were considered. The fields of the instructions that were supposed to be ignored were filled with dummy data.

With these two instruction types and their format, it was possible to represent any combination and any temperature behavior as long as it complied with the temperature limitations of the device.

Once the protocol was established, the next step was to make the DVP device recognize this new type of protocol. This was fairly simple to do; it did not even require the creation of a new characteristic dedicated to handling the receipt of the new protocol. It was simply a matter of adding a verification for the size of the data received to the already existing COVID-19 characteristic. Both the COVID-19 and the melt protocols have fixed sizes to allow the firmware to parse the bytes into an object.

4.4 Build Instructions

When the device received the protocol information through the **Params Characteristic** it would verify the size of the protocol received, decide on whether it was a normal COVID-19 protocol or a Melt protocol, and set a Boolean variable accordingly.

The state machine decided what the next state was based on this variable, assuming it was in the **Init State**; otherwise, this process was ignored. Only the case of the protocol being a melt will be discussed.

After receiving the melt protocol, the device would proceed to the **Build Instruction**

State, Figure 4.12, and build an instruction list indicating the type (hold or ramp) of instruction and its order. It did so by iterating through the protocol instruction information and storing the type and order of each one. Consider the example of two hold instructions followed by a ramp instruction, this would result in the instruction list represented in Figure 4.3.

Listing 4.3: Instruction list obtained from protocol

```
[
    {
        "instruction": Hold,
        "order": 0
    },
    {
        "instruction": Hold,
        "order": 1
    },
    {
        "instruction": Ramp,
        "order": 2
    }
]
```

At this point, the device had two objects to dictate its behavior. The protocol contained the information necessary to execute each instruction. The instruction list contained the information to decide which instruction is next. Combining these two structures, the device was able to execute any assay that could be described with this format.

While the instruction list was being built, the number of entries the assay was going to generate was calculated by summing the individual numbers of entries each instruction was going to produce. This was done as follows:
Amount of entries for a **Hold Instructions**, this instructions registers an entry every few seconds, defined in the protocol as *readInterval*, Listing 4.1[4]:

$$Entries = \frac{Duration(s)}{ReadInterval(s)} + 1 \tag{4.3}$$

Amount of entries for a **Ramp Instructions**, this instruction registers an entry every time there is a temperature increment equivalent to the *rampTemperature*, defined in the protocol, Listing 4.2 [5]:

$$Entries = \frac{TargetTemperature(°C) - StartTemperature(°C)}{TemperatureIncrease(°C)} \tag{4.4}$$

[4]The +1 is to account for the entry that is stored at the start of the instruction
[5]The denominator represents the temperature component of the rate of heating

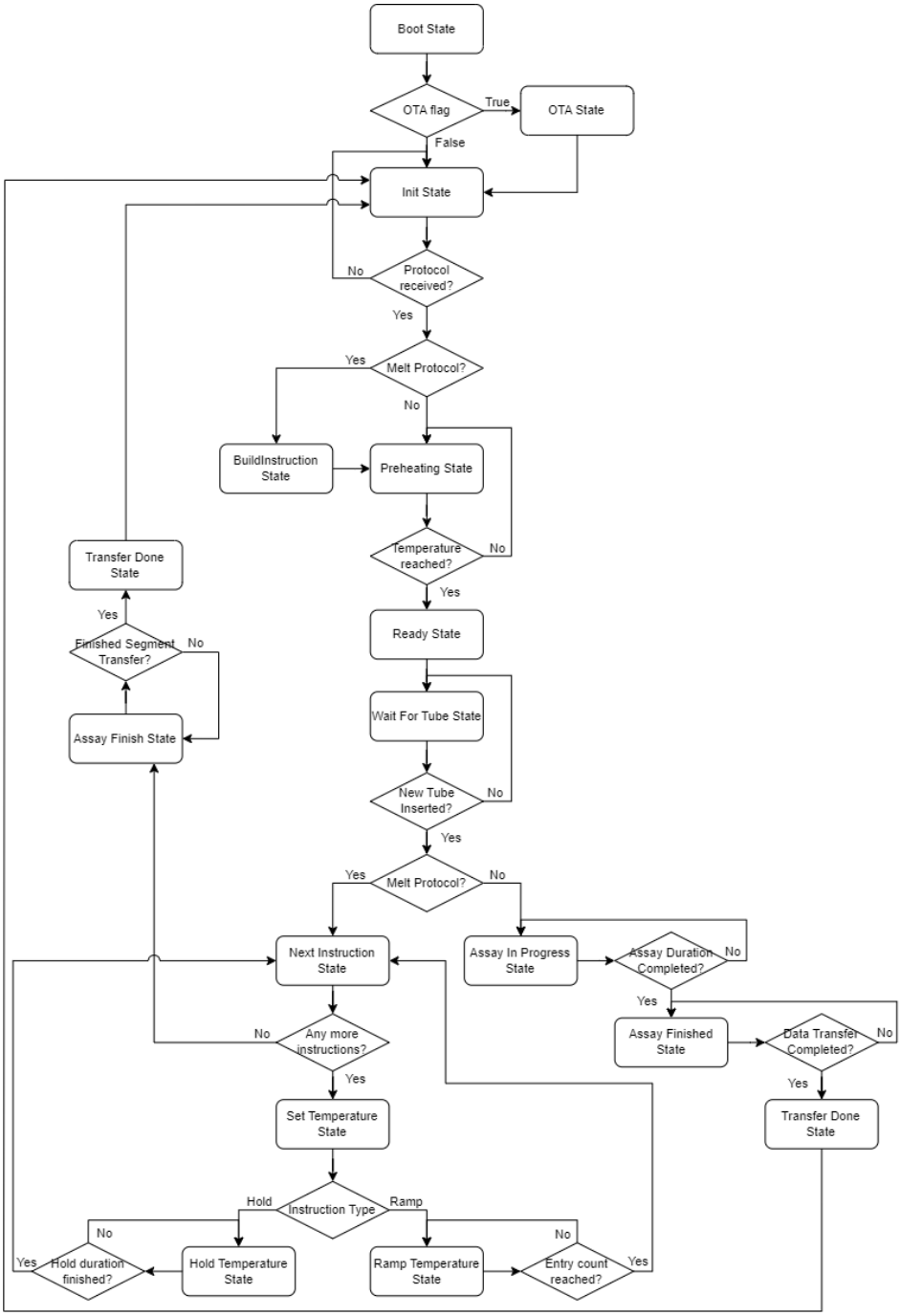


Figure 4.12: Final DoctorVida Pocket device state diagram.

This process was concluded by allocating the necessary memory to store the information that the assay would generate.

4.4.1 Allocate Memory

The data entries that were produced by either one of the two types of instructions had the same format, Listing 4.4. This simplified the memory allocation process since all

the instructions could share the same data structure. The firmware made use of the number of entries that was calculated during instruction building step to allocate the specific amount of memory space. The *SharedMemory* component, Figure 3.1, was where all the information regarding the execution of an assay was stored, this included the data segments as well as all other functionalities of the device not mentioned in this thesis, such as the file system SPIFFS.

The number of instructions referred before, 3 **Hold Instructions** and 2 **Ramp Instructions** was calculated taking into consideration the memory space the device possessed.

Using the Listing 4.4, it is simple to calculate the amount of memory each entry occupies by using the size of each value stored. These come in bytes and one entry is equivalent to 27 bytes, as can be seen in the following equation:

$$\begin{aligned} memory = 2(uint16\_t) + 1(uint8\_t) + 4(int) + 4(float) + 1(uint8\_t) + \\ 4(float) + 1(uint8\_t) + 4(float) + 4(uint32\_t) + 2(uint16\_t) \\ memory = 27bytes \end{aligned} \tag{4.5}$$

With the amount of memory one entry occupied it was possible to estimate how much memory it would be needed. This would be the case of a protocol with the longest hold and shortest read interval and the ramp with largest temperature difference and smallest temperature step. According to STAB VIDA the longest hold instruction expected to run on the device should take one hour and since the maximum temperature the device could safely achieve was 80ºC, it was possible to infer what the largest instructions would be. The most extreme case of an hold instruction that had the duration of an hour and would read data every 10 seconds, while for a ramp it would have been one that started at 40ºC and finished at 80ºC, with an increment of 0.1ºC every second. This would result in a total of 9720 bytes needed for the hold instructions and 10800 for the ramp instruction.

The free memory after the DVP pocket initialization was around 74000 bytes, using this and the values calculated before, STAB VIDA settled for 3 hold instructions and 2 ramp instructions. This configuration would take, on the worst case:

$$\begin{aligned} maximum\_memory = (360 * 27 * 3) + (400 * 27 * 2) \Leftrightarrow \\ maximum\_memory = 50760bytes \end{aligned} \tag{4.6}$$

This would leave around 23 KB of free memory for miscellaneous ESP-32 processes and any future changes the device might receive.

Once the assay was completed and the mobile application possessed all the data segments the device would receive a notice informing that it could restart, this included releasing all the memory allocated during the assay.

Listing 4.4: Data entry format

```
1    {
2        "entryNumber": uint16_t,
3        "instructionOrder": uint8_t,
```

```
        "instructionType": int,
        "readTemperature": float,
        "ledA": uint8_t,
        "channelA": float,
        "ledB": uint8_t,
        "channelB": float,
        "dataTime": uint32_t,
        "crc": uint16_t
    }
```

4.5 Melting Procedure

Having allocated all the memory it needed for the assay the device would then begin the melt procedure. This is a continuation to what was explained in chapter 2 about the firmware main loop, section 2.3.1.

As before the DVP device stayed in the **Init State** until a protocol arrived. But with the new firmware there was a slight variation when a protocol was received. The device needed to decide what was the next step whereas before it just assumed it was a COVID-19 protocol and advanced to the **Preheating State**. The Figure 4.12 depicts the flowchart for the melt firmware where it is possible to see that the initial behavior is still present and it was just built upon to handle the new protocol. All the new states, Figure 3.1, were created following the structure that was already in place, explained in section 2.3.1.

Assuming the protocol received was a melt, the device would advance to a new state, called **Next Instruction State**, this was where it was decided what the next instruction would be using the list built during the **Build Instruction State**.

Before the device actually executed the code corresponding to the next instruction, there was an important step, which took the form of a state. This was the **Set Temperature State**, that prepared the device for the execution of the subsequent instruction, namely setting the correct temperature indicated in the protocol.

If the next instruction was a hold the device would be heated/cooled to the hold temperature whereas if it was a ramp the device was heated/cooled to the start temperature, as seen in Listings 4.1 and 4.2.

Only then the device was ready to start executing an instruction. Both the hold and ramp instructions are going to be explained with greater detail in the next sections since these are the main interest of the firmware developed.

To terminate, there was one final decision the device had to make. Once an instruction ended the state that followed was always the **Next Instruction State**. Here it was verified if the just completed instruction was the last of the list, if so the device would proceed to **Assay Finish State** where it waited for the application to indicate that it could reset back to **Init State**, otherwise the loop continued, Figure 4.12.

4.6 Fluorescence Reading

An important point to consider is how the device made the fluorescence readings. Initially, the device read all six channels and stored this information, along with other information about the entry, in memory. Usually there were only 80 entries, which corresponded to reading an entry every 30 seconds for the 40 minute duration of the COVID-19 assay.

With the changes made to allow the device to execute several instructions, this had to be changed. The amount of memory needed to store the entries for the instructions referred to in section 4.4.1 would exceed the device's total memory. This would be even more severe since there were now two excitation sources, whereas before there was only one. This meant that each entry would save in memory twelve channel values, Listing 4.5, resulting in a total of 67 bytes of memory needed to store this information.

Listing 4.5: Data entry format if 6 channels were registered per excitation source

```
{
    "entryNumber": uint16_t,
    "instructionOrder": uint8_t,
    "instructionType": int,
    "readTemperature": float,
    "ledA": uint8_t,
    "channelA1": float,
    "channelA2": float,
    "channelA3": float,
    "channelA4": float,
    "channelA5": float,
    "channelA6": float,
    "ledB": uint8_t,
    "channelB1": float,
    "channelB2": float,
    "channelB3": float,
    "channelB4": float,
    "channelB5": float,
    "channelB6": float,
    "dataTime": uint32_t,
    "crc": uint16_t
}
```

Knowing this it is possible to make the same equation as before to calculate the total amount of memory needed to store all the assay entries in the worst case scenario in term of assay duration:

$$\begin{aligned} maximum\_memory = (360 * 67 * 3) + (400 * 67 * 2) \Leftrightarrow \\ maximum\_memory = 82760 bytes \end{aligned} \tag{4.7}$$

Since it was already established that the DVP device free memory was around 74000 bytes, there would not be enough space to store the entries for this assay.

What STAB VIDA noticed was that most of the time only one channel was used for analysis. With this in mind, the solution of only registering a single channel was decided.

With this, the problem of not having enough memory was resolved, but while this was no longer a problem, there was a new complication that came with the new temperature ramp instruction.

While the device now only stored one channel, when it came time to make the fluorescence reading, all six channels were still being updated every time a new entry was registered. This was unavoidable since that was how the sensor driver was programmed to be. What could be controlled was the integration time. This value would indicate to the sensor how long it had to observe the fluorescence emission, where higher exposure levels result in finer values, giving more details about the reaction happening inside the chamber. A more common example of this is the shutter speed of a camera, where when it is low the lens are exposed to the light for longer.

This itself could take some time. The integration time used a byte-sized variable, meaning that the maximum value would be 255. Besides this, reading each channel value would also take some noticeable time since the sensor would need to poll the registers where the fluorescence values were stored until they were free to be read.

Adding all this time needed and multiplying by two, for each excitation source, and very occasionally this process would exceed the 1 second mark, which was the smallest period of time that the temperature ramp had to increase in temperature (0.1ºC/s, 0.2ºC/s, etc), and if the integration time was high enough, this would almost always occur. This would cause the temperature ramp instruction to register sensory information with an accumulating delay, which in turn greatly increased the temperature ramp duration, since the next temperature step only started after the previous was completed, which included both the temperature increase and the fluorescence reading. The delay could also affect the results obtained since the temperature of the reaction required would not be correctly controlled.

The solution to this problem was conveniently the same as the solution to the memory issue. Reading only one channel decreased substantially the time it took to make a fluorescence reading, allowing the temperature ramp to use intervals of one second.

While this is true for lower values of integration, for higher values of integration of 200 or above, there were still delays when registering a new entry. In Figure 4.13 this phenomenon can be seen, where when the integration was set to 200 or above, the time it took to complete the same temperature ramp was significantly higher. This is caused by the sensor taking longer than one second to measure an entry and storing it in memory, causing a delay when registering the next entry.

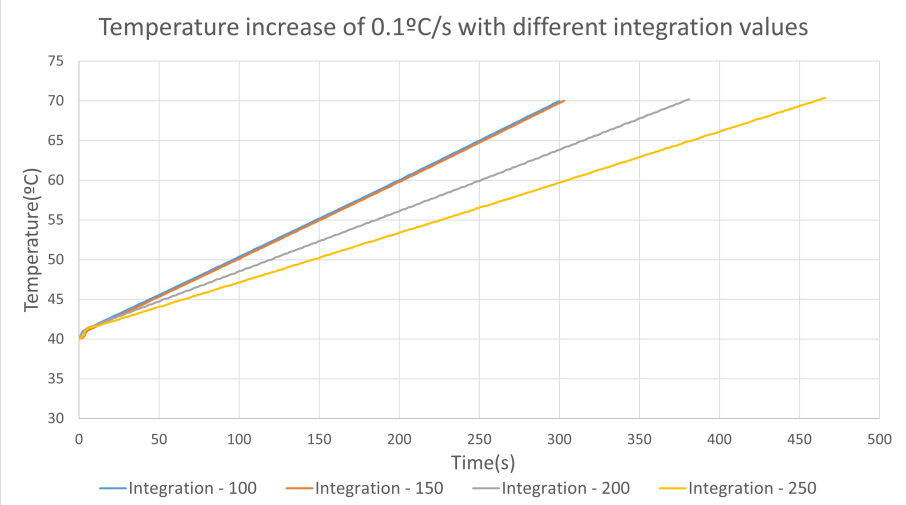


Figure 4.13: Comparison of the duration of an assay with the same parameters for different values of integration time.

Having considered these issues, the final decision made by STAB VIDA was to only read and store a single channel's fluorescence value per excitation source and only use integration times of up to 150.

With the correct settings for reading fluorescence values without affecting the assay, the next was to develop the two new instructions, Figure 3.1, **Hold Temperature** and **Ramp Temperature**.

4.7 Hold Instruction

The temperature holding instruction was approached first since it was much simpler and some of the code for the COVID-19 assay could be used as a base. The Figure 4.14 represents the fluxogram describing the behavior of this instruction.

Like in all other states, it had 2 main functions: the *run()* function and the *enter()* function. The first was designed to setup the variables needed for the execution of this instruction. The second was going to be executed continuously in every loop of the microcontroller until the instruction was over.

The *enter()* function started off by resetting the entry counter variable and the time-related variables, notifying the application that the new state had started and saving the time at which the instruction began, so that any calculation that involved time could be done relative to the beginning of the instruction. It would then obtain the information about the hold instruction and register the first entry. Finally, it would turn on the blue LED and beep for 1 second to signify to the user that the instruction had started.

There were two important functions related to time management. One was to update the current time variable, and the other was to check if a whole second had passed. These are important, both for knowing when to register an entry and to notifying the application

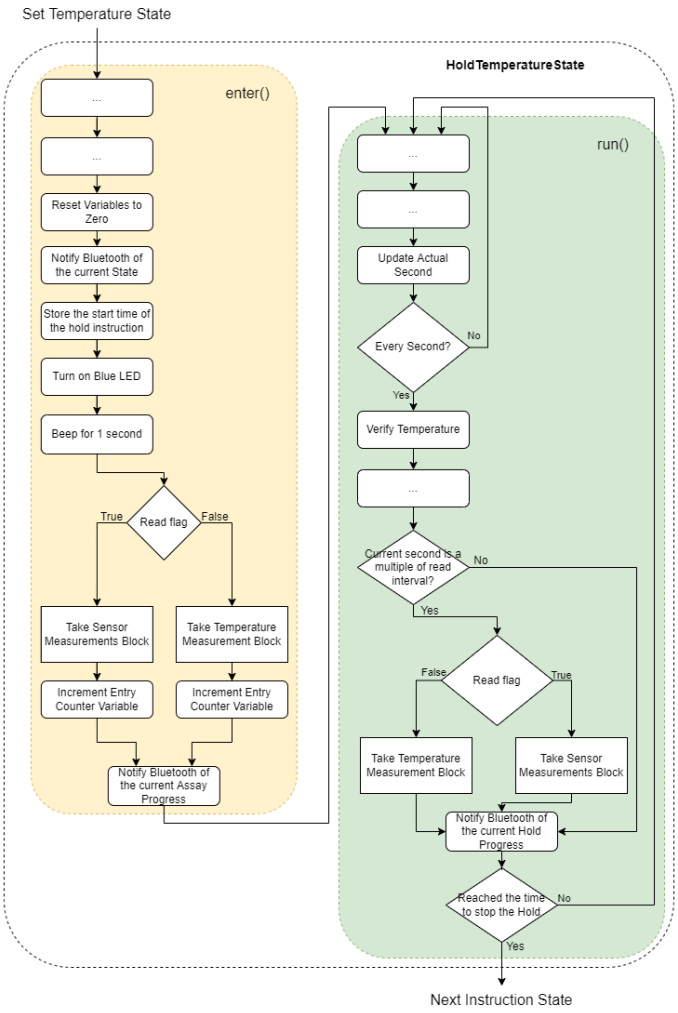


Figure 4.14: Flowchart of the temperature hold instruction.

of the instruction progress.

Every time the *run()* function was executed, the function to update the current time variable was called. Therefore, this variable marked the time at which the *run()* function started. But the protocol read time came in whole seconds. This is when the second function comes into action. It checked to see if the current time variable had reached the next whole number. If it had, it meant that a second had passed, and a new variable would be updated with this new value.

The way to know if it was time to register an entry was by dividing the last second variable by the read time of the protocol and checking if the remainder was zero. This is possible since both numbers were whole numbers.

When it was the correct time, one of two function from the *MeltingMeasurements* components was called, Figure 3.1. If the protocol indicated that the fluorescence values were to be registered, then a function on this component would store the entry's basic information, such as temperature, time, id, etc., plus the fluorescence readings. If, on

the other hand, the protocol instructed not to read fluorescence values, then only the basic information would be stored and the fluorescence values would be set to 0. This meant the space occupied was the same no matter if the instruction was meant to read the fluorescence levels or not.

After registering the entry, the device would notify the application that a second had passed. This allowed for the application to keep track of the assay progress and to know when to ask for a new entry, since it could also calculate when the device would have registered a new one.

The *run()* function was executed until the current time variable had exceeded the instruction duration, which was defined in the protocol. By then, the firmware would proceed to the **Next Instruction State** where the next instruction to execute was decided.

4.8 Ramp Instruction

The temperature ramping instruction shared some aspects with the temperature holding instruction, namely the time management functions and variables and the fluorescence registering component. What differed was the temperature control and when to register an entry, which introduced new variables and behavior. The Figure 4.15 represents the fluxogram describing the behavior of this instruction.

For this instruction, the *enter()* function also reset the variables needed for the execution of the instruction and stored the start time. It also notified the application of both the state change and the instruction progress, which by then would be 0. This behavior was shared with the hold instruction.

A major difference between the temperature holding and the temperature ramping instructions were the conditions to register an entry. It used an extra function called *updateStepVariables()*. This function was called once on the *enter()* function to start the process and it would be called continuously on the *run()* function.

The *run()* function also updated the application of the time progress, much like the temperature hold instruction, but this was where the similarity stopped. Before continuing, the function would verify if the number of registered entries had already reached the total amount expected; if so, it would exit to the **Next Instruction State**. If this was not the case, then the instruction continued.

The next thing to check was if the temperature step period had already passed, so if the temperature ramp step was 0.1ºC/s, it was checked if a second had passed. When this was the case, then the extra function, *updateStepVariables()*, was called.

This function had two important responsibilities: advance to the next temperature step and verify if the conditions to register a new entry were met. The advancing of the temperature step and the registering of a new entry were independent, meaning that the temperature ramp might be at a higher temperature value than what is being recorded in the latest entry.

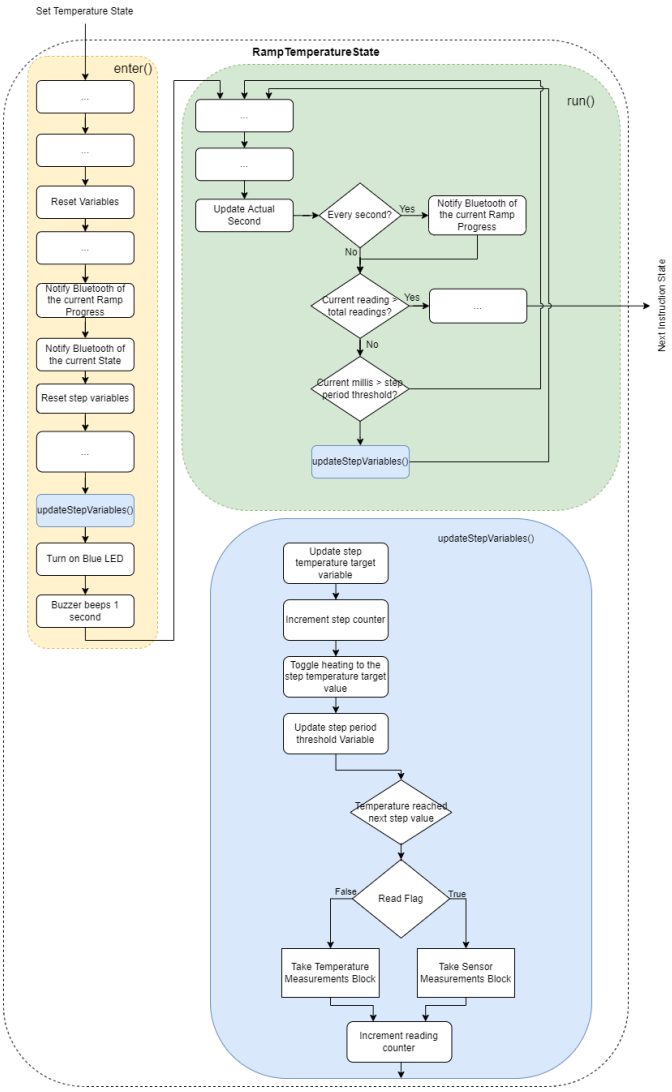


Figure 4.15: Flowchart of the temperature ramp instruction.

The reason for making them independent was that the number of steps the device took to reach the final temperature was not possible to calculate *a priori*, since the ramp had a one-degree error margin around the initial temperature, so it could start at different temperatures, and besides that every assay had multiple factors influencing the temperature that were not constant, such as room temperature, different devices, etc. Summing all these nuances up, it was not possible to guarantee the same number of steps every time a ramp instruction was executed using the same protocol.

The solution that was used for this was to separate the moment of registering an entry from the temperature ramp progress. This way an entry was registered once a temperature threshold was reached, when this happened the threshold was updated to the next temperature step, this is depicted in the flowchart, Figure 4.15. As an example, if the ramp temperature step was 0.1ºC/s and the start temperature was 40ºC, the first

entry would be registered at the 40ºC mark, the second at 40.1ºC, the third at 40.2ºC, and so on.

This explains why some entries were registered with a few seconds' difference on examples from previous sections about rate of heating limitations and temperature fall-off. Because that was the time it took to reach the next temperature threshold.

This method of increasing the temperature step every period of time was used instead of defining a PID configuration that would have the desired behavior, because the latter was not viable. Given the multiple protocols with different temperature behaviors and the fact that no two DVP devices have the same temperature profile[6], it was impossible to find a PID configuration that met every possible temperature ramp. Configuring the PID components dynamically would not be feasible either, since that would increase the duration of an assay by a significant amount and could lead to a lot of mistakes due to multiple factors such as the device temperature limitations and surrounding conditions.

4.9 Transfer Results

Transfer results was the last step before the device got reset back to the **Init State**. The state it self, called **Assay Finished State**, only verified if a flag was true or false, the real process was done in the Melt Status Callback, Figure 3.1, here every time the application requested a segment, the firmware verified if that was the last segment of the list, when this was the case the flag was set to true.

This worked for both the old COVID-19 protocol and the new melt protocol, since all that the **Assay Finished State** needed to verify was a flag, which could be set by either of the callbacks. When this was done the state machine would advance to the **Transfer Done State**.

Here the memory allocated for the previous assay, that had just finished, was released and the device would advance to the **Init State**, thus concluding the assay.

4.10 Firmware Bugfixing

The development of the firmware was completed in stages, the first being related to temperature control, which was discussed in previous sections. Here there were few problems, and most of the time was spent testing new configurations that could improve the temperature driver performance.

Then came the development of the various states that were created for the purpose of the melting process. Here, most of the problems came from memory allocation and registering new entries.

The initial design of the device's firmware stored all fluorescence channels and did not make any calculation of the space required to execute the assay, since all the device

[6]The components that go into the device are not exactly equal, the conditions vary between every device, even if ever so slightly.

was designed to do was to hold a temperature for a given amount of time at a given temperature, while reading entries every few seconds. What was done was to reserve a predetermined amount of memory to cover all possible cases, since they were not that many and not that memory hungry.

This method, when applied to the new instructions, could have led to memory problems. This is why the channels stored were reduced to one for each excitation source, and the device only reserved just enough memory to store the entries that it needed. This way there was no wasted memory, and it was possible to comply with the memory requirements of any possible melt protocol.

The only major bug that was detected had to do with the base firmware that the new one was built on. The base firmware possessed a form of storing information about the assay during runtime in a log file. The amount of information and how detailed it is could be controlled by the user. This involved writing in files that were stored in flash memory and were identified by the assay ID number. This functionality was fairly recent at the time, and even though it was thoroughly tested, some problems could not have been found, which was the case for this bug.

The problem was detected whenever a user indicated that the device should store more detailed logs. This process would take longer than normal and could cause the DVP device to crash. The problem was found when the melting process was being made, since it used the base firmware that had the bug.

After analyzing the crash causes, the conclusion was reached that the watchdog was resetting the device when storing or reading large amounts of data on a file.

4.10.1 Watchdog Starvation

The watchdog is a mechanism that ensures the interrupt service routines are not blocked for a long period of time. This is undesirable mainly because it prevents task switching. Using the information on the ESPRESSIF website[39], the watchdog timer is expecting the tick interrupt on each CPU to be able to feed the timer. When the timer is not fed, it will run out and starve, which in turn will invoke the panic handler, indicating to the watchdog that this particular CPU is stuck with some high-priority process that does not allow it to switch tasks. This will cause the chip to hard-reset, stopping any processing that was hogging the CPU at the time.

The process dealing with file reading or writing had a high priority, indicating to the CPU that it could not be interrupted. This was due to the need to ensure that the file information was handled safely, or else it could become corrupted. This triggered the previously mentioned watchdog panic response.

This problem's solution is outside the scope of the thesis. It was not resolved by STAB VIDA; third parties were involved.

4.11 Active Cooling

Lastly, to complete the topic of the firmware, adding active cooling to the device was a subject that was planned to be approached during the thesis development but had to be relegated to second plan as the melt procedure became a more pressing matter when the lactose intolerance use case was made the primary target of STAB VIDA for the next version of the DVP device.

This being said once this matter was completed, some testing was done just to prove the concept of using active cooling with a fan to decrease the duration of an assay by shortening the time the device took to cool down.

4.11.1 Thermal Profile

The thermal profile of a device with no active cooling was compared to that of a prototype device with a small fan installed, Figure 4.16. The size of the fan had to be small enough to fit the already existing case while still being able to move enough air to cool down the device chamber, and the device would need vent holes from where the air could be pulled in by the fan.

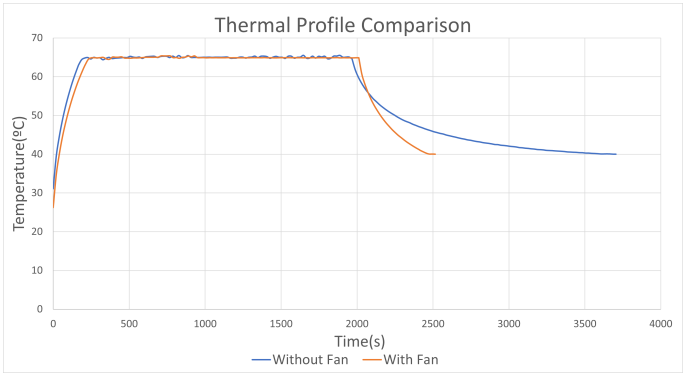


Figure 4.16: Comparison of the thermal profile of a device with no active cooling and a prototype with a fan installed.

It is possible to see that if this alteration to the device were to be implemented, the device would be able to cool down much faster than normally, around a 30% improvement. This would greatly improve the assay duration, which proved a major burden to the device during the development of this thesis.

4.12 Conclusion

This chapter covered all relevant subjects that were worked on during the thesis. With the new assay protocol for lactose intolerance in mind, many changes had to be made in order for the device to produce valid data in real-time that could be used for analysis and display results to the user. These included changes to the state machine, which controlled

the states the device could execute, and changes to the Bluetooth service, in order to allow external software to communicate with the device using BLE.

This resulted in the creation of a device capable of running multiple protocols with varying temperature control and specifications while also producing fluorescence readings. Even though the new device had a lot of changes, the original behavior was kept. This means that a user can still use the same assay protocols as before, like the COVID-19 one.

All of this effort had to be made useful, for this, some form of interaction with the device had to be created that could initiate a test in the device, display to the user the progress of the assay and generate a result. This will be discussed in the next chapter, where the software created to interact with the device is explored.

5

Software

This chapter will present the work done in the software. The development plan was to create the firmware for the DVP device capable of executing the melt procedure and then carry on to the mobile application that would interact with this device. This plan had to be altered due to a short-noticed occasion that required a working demo with the device. Since it was expected to take more than a month to get at least a working version of the mobile application, STAB VIDA decided to make a Python script that could show the data in real time, analyze it at the end of the assay, and give the user a result.

After this step was completed, the focus would shift to the application, where some changes were made to allow it to execute the new melt protocol. The application would then take over the script, which would no longer be used.

5.1 Python Script to Execute Melt Assay

The script had to begin by sending a melt protocol to the device, so it started by creating the instruction list, with the specific instructions the user desired. In this case, the instructions were about the lactose intolerance test, which was explained in a previous section.

After asking the user what the assay ID is and the email address to send the results to, the script would prepare a directory to store the information relative to the assay that was about to be executed. This directory was identified by using the device and current date and time and was meant to store the data file where it would save the data segments, the graphics that were generated in real-time throughout the assay, and the final PDF report generated after the analysis was completed.

After creating this directory, the script started the two main processes relevant to the procedure. These were the primary process, which carried out the assay, and the real-time process, which plotted data segments in real-time.

5.1.1 Main Script

The main script executed the main process and made calls to auxiliary scripts. It was given the device address, that was suppose to execute the assay, as an argument, this allowed to find the device and connect to it.

All the Bluetooth-related communication in the Python scripts used the bleak[40] platform, which is a GATT client software capable of connecting devices using BLE.

To start the process, the script would use the BleakScanner from the *Bleak* API, which was used to discover BLE devices by listening to advertisements emitted. This only obtained devices that were not paired and connected already, because when this happens, the devices stop advertising themselves. Using the list of devices that were advertising it was possible to search by the address that was given as an input to the script. If no device was found with the given address, the script ended with a message indicating the inability to find the device.

After scanning for devices, the main cycle begins (see Figure 5.1). This loop was executed continuously until the assay was done and the finish state was detected. It used the BleakClient, which received the address of the device and connected to it. Upon successfully doing so, the script instructed the client to start listening to BLE notifications from specific characteristics produced by the device. The device, upon receiving the request for a data segment, would update the characteristic with the correct entries and notify them. So any connected peer could either listen to the notifications or request to read the characteristic value itself, explained in section 4.2.

Then, each iteration began by determining whether the device was disconnected. This could have occurred for a variety of reasons, but what is relevant is that the script attempted to create a new BleakClient to connect to the device until it was successful, pausing for a period of time between each attempt.

Having the device connected and ready to execute an assay, the next step of the script would start. This is where the actual assay procedure starts, with all the previous work being related to setting up the connection.

The current state of the device was obtained in each loop, and the script had specific behavior for each state the device could be in.

Upon checking that the device was in **Init State**, the script would send the protocol information via Bluetooth using the BleakClient. This information had to be converted to bytes, which required the use of the Python package *Struct*, which is a binary data service supplied by the Python library. This package used a format string to indicate the data types of the data that was to be converted to bytes. Each data type has a format character to represent it; this can be found in the Python documentation[41]. This is similar to C++ which used its own version of a *Struct* type to store protocol data.

When the **Build Instruction State** was detected, not much was done, since all the device was doing was preparing to execute an assay. So the script simply printed an informative text for the user to know what stage the device was at, while also saving the

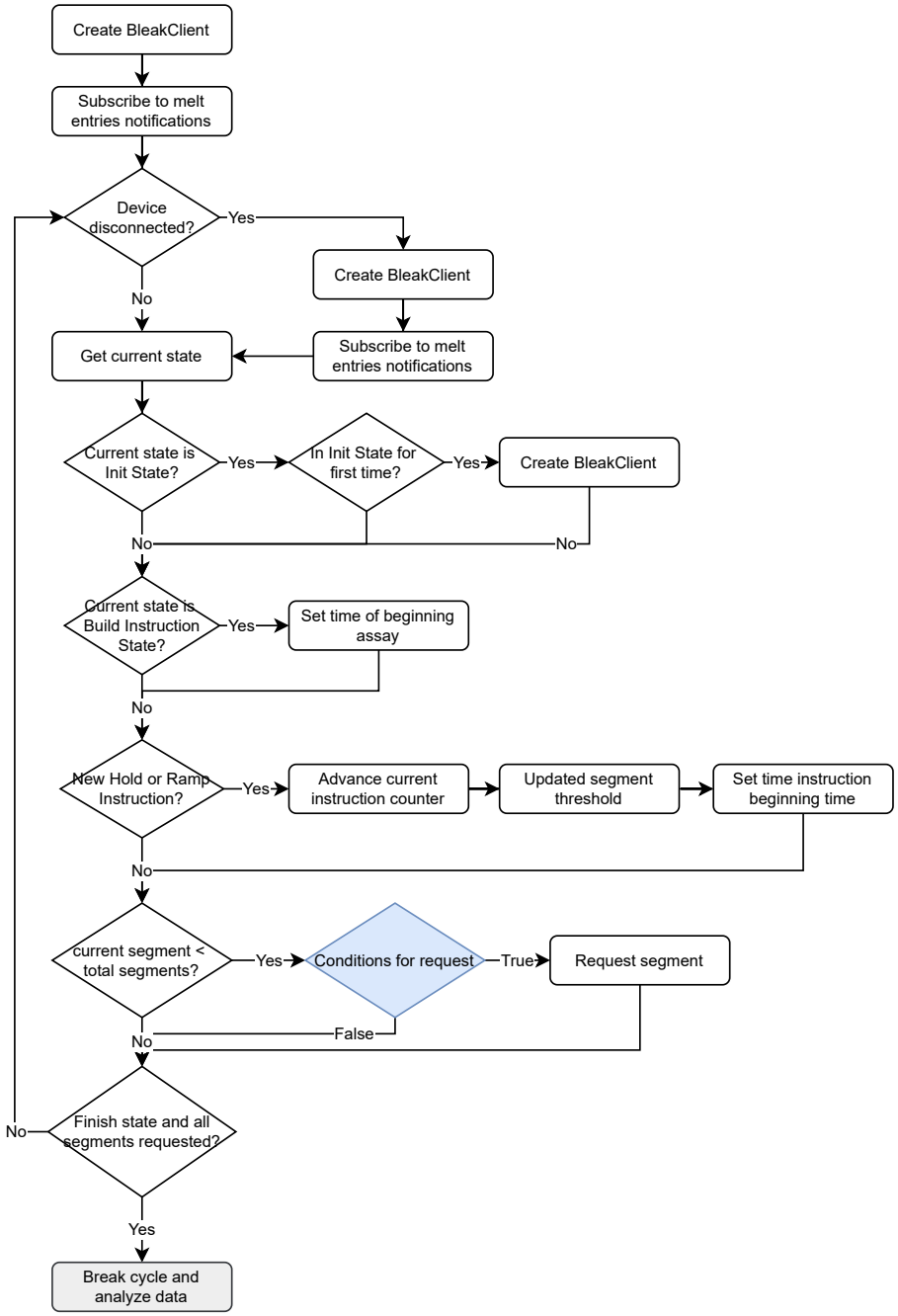


Figure 5.1: Flowchart of the Python script main process.

time of the assay's start.

The script would then wait until one of the two instructions was executed on the device. When this happened, it checked if it was a new instruction state, different from the state found in the previous iteration; if so, it would update the current instruction counter, which indicated what state the device was in, as well as increase the threshold of segments it could request from the device.

The request and storage of segments by the script had to be made independent from

the actual state the device was in. This was decided due to the script sometimes not keeping up with the device, mainly when it came to the temperature ramp instruction. Every time the device advanced to the next instruction, the script updated the maximum number of segments that he could request. This way the script could progress through the assay at his own pace, which normally was not very different from the device's, but this served as a preventive measure in case some segments took longer to request and process.

After checking for any state updates from the device, if an instruction was already being executed, the script would then try to request new segments from the device. Before doing so, it had to verify a few conditions, namely if the ordinality of the segment it was trying to request was less than the total amount allowed by the segment threshold and if enough time had passed since the last time a segment was requested; this last condition had to comply with the read time that was registered in the protocol for each instruction.

Upon successfully obtaining a segment from the device, the script would map the bytes into the corresponding Python data formats, using the same *Struct* package, and then store the information in a CSV file for later usage.

Lastly, the loop would verify if the device was already in the **Assay Finish State** and then if all the segments were obtained. If these conditions were met, then the script would emit the finishing segment to the device so it could reset back to the **Init State** and break the loop, thus terminating the assay cycle and advancing to the next phase dedicated to analyzing the data obtained throughout the assay and generating a PDF with the result.

5.1.2 Real Time

The main process was responsible for handling the device state changes and requesting the data segments accordingly. Once a segment was obtained, the data would be appended to the file designated for that assay. The file would be the same for every instruction, since the format of the data segment did not change between them.

The data file would be used for the real-time component of the script. This was a process that had to be executed in a different thread using the Python *multiprocessing* package due to the script plotting graphs in real-time without closing them using a function named *FuncAnimation* from *matplotlib*[42], which would block the execution of the rest of the script.

The thread started before the main process connected to the device. The *FuncAnimation* function received another function as an argument, which was called repeatedly, as well as the plot that is supposed to be displayed each frame and, lastly, the interval at which to call the function.

This function had the job of updating the plot to be displayed each frame. It started off by reading the CSV file and storing it into a data frame using the *Pandas* library. From there, it would divide this data frame into smaller ones based on the instruction order of each line. This way, each smaller data frame contained the data relative to each

instruction. The script would then use the last of the smaller data frames, which is the one that contains the data of the last instruction[1]. Since this was being executed in parallel to the main process, this last data frame contained the data of the instruction being executed at that moment.

With the current instruction's data, it was simply a matter of plotting this information. Before doing so, the script would process the data segments. This was the same processing that would be used when analyzing the data to obtain a result, which will be discussed next.

The graphs were customized according to the protocol. If the LED color in the protocol was blue, then the graph line would also be blue, and the same if the color was green. This allowed the user to distinguish the fluorescence readings of each channel since they were both displayed on the same plot, Figure 5.2.

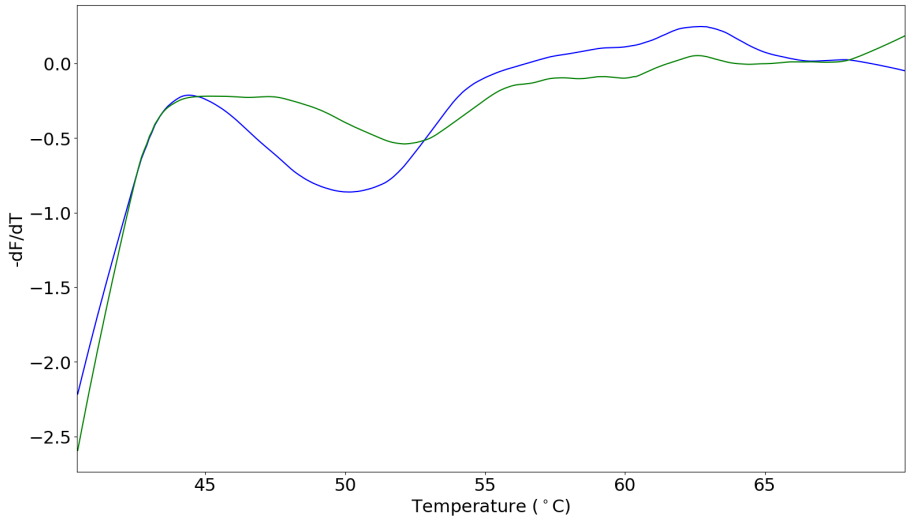


Figure 5.2: Example of the Python script plot with two channels being read.

The LED color was used instead of the channel color since it was possible to read the same channel for both excitation sources, resulting in the same color for both lines, while the opposite was not possible; the excitation sources had to be different, thus guaranteeing that the graph lines would have different colors.

After all the data points have been processed and are ready to be plotted, the function would clear the current figure and plot a new one with the most recent data.

This process would be executed every second, even if there were no changes to the data frame, and would go on until the final instruction was completed.

Finally, when the script detected a new instruction order in the data file, it meant the previous instruction was over. When this happened, the figure of the previous instruction was saved into an image and stored in the file system for later usage.

[1] While the assay was not yet producing data segments, the plot would be empty.

5.1.3 Result Analysis

Once the assay was finished, the file would have all data segments in the format from the Listing 4.4. Using the instruction number and order, it is possible to divide the data into distinct instructions and analyze each instruction separately, as was done in real-time.

The script began by defining relevant parameters for individually analyzing each instruction. There was a list containing the parameters for each hold instruction and another list containing the parameters for each ramp instruction. An instruction was only considered for analysis if one of the excitation sources was active, as determined by the script's examination of whether the excitation LED was blue or green; otherwise, both excitation sources were deactivated and the device did not record fluorescence readings. If so, the analysis script would simply ignore this instruction.

When eventually the script found an instruction with fluorescence readings, it had to check if it was a ramp or a hold instruction, and act accordingly.

5.1.3.1 Hold Temperature Analysis

The hold analysis combined the results from both channels to produce a final result. It received a data frame with the specific data that was extracted from the larger data frame.

The next step was to analyze each excitation source's fluorescence readings individually. Both were analyzed the same way, and basically it detected if the fluorescence levels increased enough to cross a threshold, which was defined by a trained professional. If so, then the result would be positive; otherwise, it would be negative.

The final result was saved in a dictionary built *a priori* with tuples as the keys and the final result as the value. The tuples were a pair with possible results from both channels. Each channel could have a positive, negative, or not available[2] analysis result. With the result from each channel, it was possible to obtain the final analysis result of the hold instruction from the dictionary.

The meaning of this result depended on the assay protocol and analysis, but for instance, it could be used to validate the assay. If the fluorescence did not increase above the threshold, the assay would be invalid. If the protocol only had one hold instruction, the result from the analysis would be the final result.

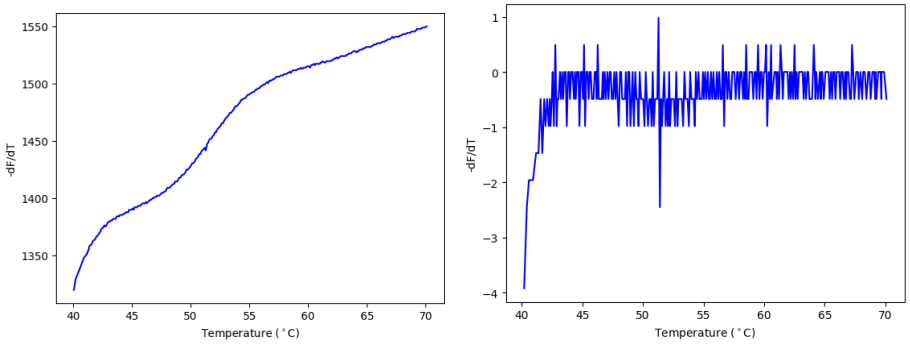
For the lactose intolerance test, this analysis was ignored; this can be seen in a future section where the analysis of the hold instruction gave the result *Proceed*, Figure 5.9, meaning that the script should continue evaluating, even though the curve did not cross the necessary threshold.

To end the hold instruction, a graph with the fluorescence curve and a line representing the threshold would be made. This graph would then be saved in the file system so that the user could check for himself how this instruction was analyzed.

[2]When the excitation source was not used, the result was not available.

5.1.3.2 Ramp Temperature Analysis

The analysis of a temperature ramp instruction started off by normalizing the data inside the data frame. It did so separately for each excitation source, and then the script would calculate the derivative for all the fluorescence points over the temperature increase. The result would be a curve that had peaks at the temperature values where the fluorescence suffered more noticeable changes in its values, meaning that there was an increase in the curve slope[3], Figure 5.3.



(a) (b)

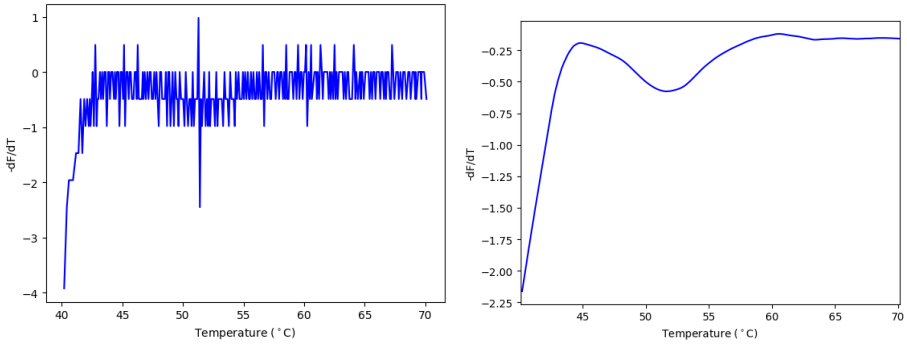
Figure 5.3: Ramp Analysis: a) Raw fluorescence data, b) Fluorescence data after normalizing and applying derivative.

Having normalized and applied the derivative to the fluorescence data values of each channel, the script would then check if the LED of each excitation source was either green or blue; otherwise, it was ignored, just like it was done for the temperature hold instruction. The normalization and derivation were made prior to this since it was easier to apply these modifications to the whole table using Pandas data frame functions, simplifying the code. Both of this operations were based on a script made available online by Yi Liu from the University of California [43]

Once it was asserted that the fluorescence channel data was meant to be analyzed it would apply smoothing to the curve of the derivative, removing all the noise, Figure 5.4. This smoothing was made using a python module named *statsmodels*[44], which provided a function called *lowess()* that would return the data with the desired smoothness based on the parameters passed on to the function. This used **LOWESS** (Locally Weighted Scatterplot Smoothing) which creates a smooth line that goes through the data points. It does so by considering a localized subset of data to fit a low-degree polynomial using weighted least squares, as explained in NIST's handbook[45]. It is weighted since it gives more relevance to points near the point of estimation and less relevance to points further away. The subset of data used for each weighted least squares fit is obtained by a nearest

[3]The peaks were inverted, meaning that when there were increases in the curve slope, there was a decrease in the derivative curve. This was decided to be so by STAB VIDA

neighbors algorithm where the user specifies how much of the data is supposed to be used to fit each local polynomial using a parameter called *smoothing parameter*.

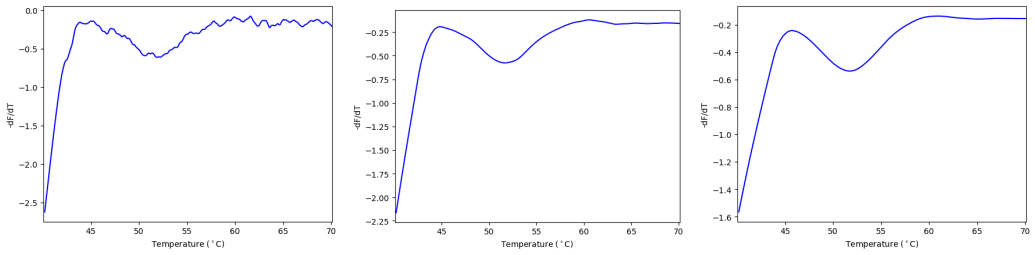


(a) (b)

Figure 5.4: Ramp Analysis: a) Fluorescence data after normalizing and applying derivative, b) Final curve for analysis with smoothing applied.

During development many values for the smoothing parameter were tested and it was chosen the one that kept most of the curve characteristics while being smooth enough not to falsely trigger the peak detection (Figure 5.5).

The peak detection algorithm used the smoothed curve to try and detect any peaks by applying a function from *SciPy*. This was the *find_peaks()* function[46], it received as input the smoothed data, the minimum height and the minimum width a peak should have and returned the indices of the peaks found. This index would be on the temperature axis and would indicate at what temperature the peak occurred.

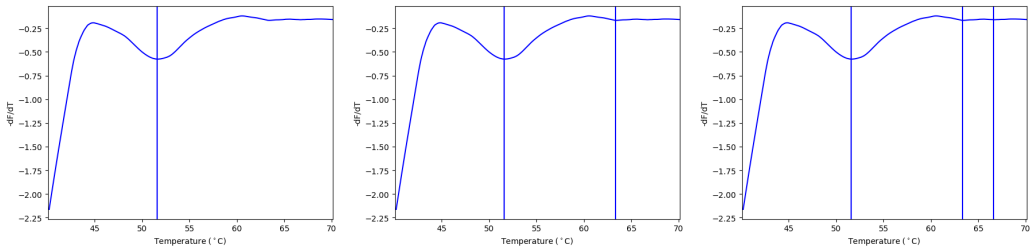


(a) (b) (c)

Figure 5.5: Smoothness: a) Curve with lower smoothness parameter value, b) Curve with used smoothness parameter, c) Curve with higher smoothness parameter.

Then for each peak the prominence was calculated. Peak prominence is described by *MathWorks*[47] as "The prominence of a peak measures how much the peak stands out due to its intrinsic height and its location relative to other peaks". The document presented in the website gives an example on how to calculate the peak prominence. Using the peak prominence, it was possible to find the peak that stands out the most. The

script only allowed any other peak if it passed a threshold of prominence that is defined by the highest peak prominence divided by a **factor**. In Figure 5.6, it is possible to see what happened if this factor was changed. In this example, the curve is rather smooth, so not that many new peaks were detected with the changes in the prominence factor, but this is not always the case. In some cases, a small difference in this factor may generate multiple incorrect peaks that will invalidate the final result.



(a) (b) (c)

Figure 5.6: Prominence factor: a) Peak detection using the default factor, b) Peak detection with higher factor, c) Peak detection with even higher factor.

For the analysis of the ramp instruction, there were two possibilities: it either had one peak or two peaks. Each peak indicated the presence of a genotype, which was explained in more detail in section 1.1.4. There were only three possible genotypes, CC, CT, and TT. Initially, if a peak was detected between 48ºC and 54ºC this would be a CC, if the peak was detected between 56ºC and 64ºC this would be a TT, and finally, if the peak was detected on both of these ranges, the result would be a CT. This was later changed by decision of STAB VIDA to 47ºC to 55ºC for a CC and 55ºC to 62ºC for a TT, with the CT having to have a peak in both of these ranges.

Having obtained the most prominent peaks it was a matter of analyzing where their position was within the curve and decide the result to give.

The results, much like the hold instruction analysis, were stored in a data structure. In this case, there were two lists; one stored the single peak encoding and the other the double peak encoding. An encoding was an object that stored the peak range limits and the result if the peak was between those ranges. The single peak encoding had one temperature range where the peak could exist and one result. The double peak encoding had two temperature ranges and a result.

The script would first iterate through the double peak encodings and check if any of the peaks it found were between each of the temperature ranges in the encoding; if this was the case, then it would return the respective result. Once these were all checked, the script would advance to verify the single peak encodings. Here it looked for what encoding range contained any of the peaks it found, since it was established that it did not have two valid peaks, returning the encoding result when this condition was met. If a peak was detected but outside the ranges defined in the encodings, then it would be

ignored.

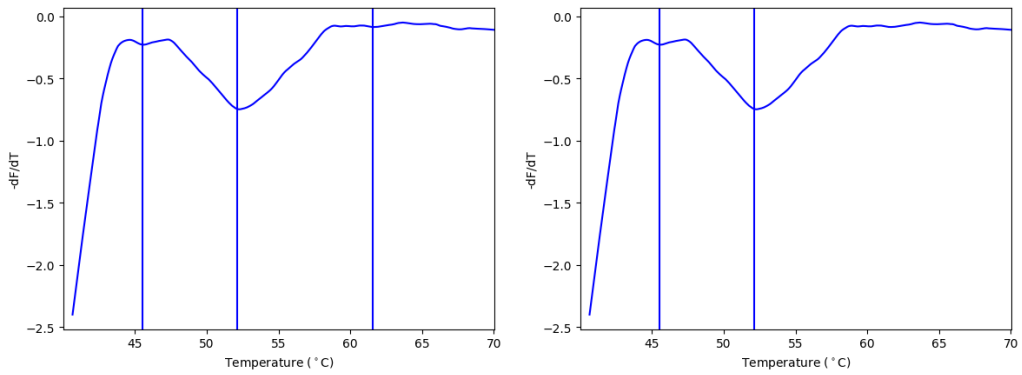
The script did not always find a peak in the curve, or it could find incorrect peaks. There were some complications when tuning the peak detection function that are worthy of mentioning.

5.1.3.3 Peak Detection Difficulties

The peak detection function went through multiple iterations until the final configuration was decided.

Initially there was an attempt to use the height of the peaks to detect them, but it was quickly noticed that this would not work since different assays generated fluorescence curves with different fluorescence ranges and scales. What needed to be measured was the intrinsic height of the peak; thus, the use of the peak prominence was appropriate. Even though this was the case, a small height threshold was kept since it helped remove noisy peaks from the equation.

Another addition to the function was the minimum width a peak had to have. With a carefully calibrated width, small one- or even three-point wide peaks were ignored (Figure 5.7), which helped in some cases where the smoothing was not enough.



(a) (b)

Figure 5.7: Width parameter: a) Peak detection using peak minimum width of 0, b) Peak detection with higher minimum width.

With the height threshold and the minimum width taken into consideration, the function would still return some peaks that were supposed to be ignored. Initially, a default peak prominence threshold was used, but this did not satisfy every case due to a problem similar to that of the peak height, an example of this can be seen on the results section 6.2 (Figure 6.4), as there was an important result that suffered from this problem.

This is where the highest peak prominence was used. Following the previous explanation, the peak with the most prominence was used to define a threshold that every other peak's prominence had to comply with for it to be considered. This way, this threshold was relative to the most prominent peak and not just a default value.

The combinations of all these strategies greatly reduced the number of false peaks detected by the algorithmm, see Figure 6.5 of the result section 6.2, but in some rare cases, these peaks were still considered by the algorithm. If this happened, the user had to redo the analysis with slightly tuned parameters. This was not a big concern because the script was only meant to be used by STAB VIDA employees for demos and tests until the final application was ready and was not meant to be commercialized by STAB VIDA.

5.1.4 PDF Generator

The final component of the Python script dealt with generating a PDF report for the user. The format used consisted of three sections. The first section displayed the protocol temperatures that were going to be used throughout the assay, Figure 5.8. This did not take into consideration the time to reach the temperatures before each instruction.

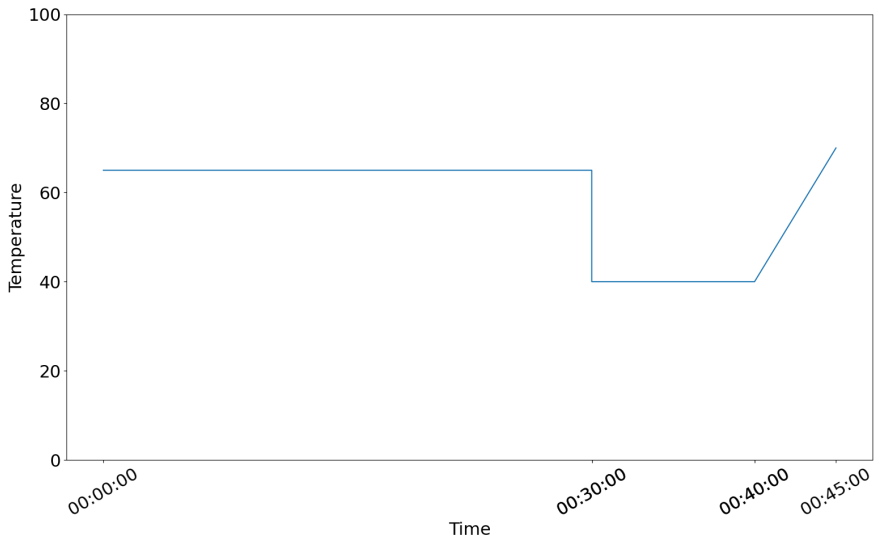


Figure 5.8: Image used in the PDF to illustrate the temperatures used throughout the assay.

The next section describes each instruction individually, indicating the value for every field of all the instructions present in the protocol.

Finally, the last section presents the plots with the detected peaks and the final analysis, Figure 5.9. The figure of the graph with the final result displayed a solid line for every peak detected and a dashed line for the range the peak was in. In the case of the image, the range was that of the CC which was 48ºC to 54ºC since this example used the initial version of the peak ranges.

Every component of the report made use of a Python module created just for the purpose of generating this PDF. It was built on top of a library for Python called *PyFPDF* [48]. This library allows you to position elements like text lines, images, and objects on a specific location by using coordinates and the width and height of the component. For

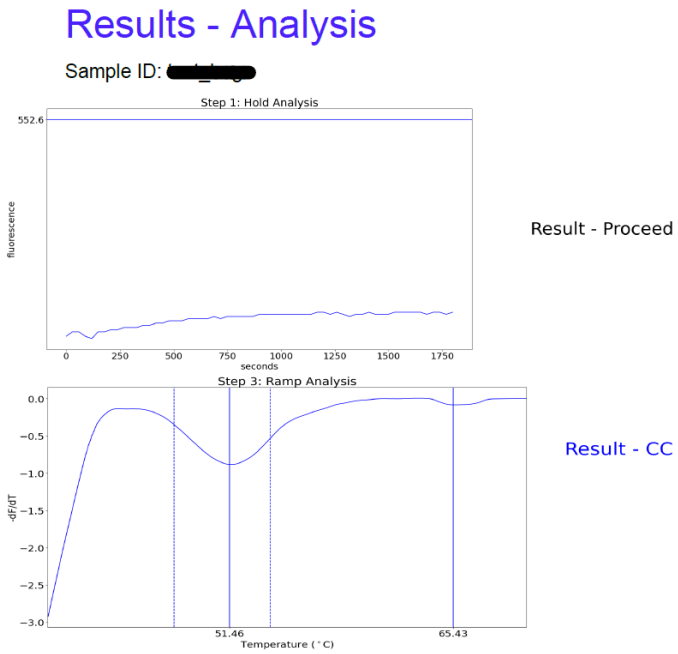


Figure 5.9: Analysis portion of the PDF report.

example, to place a text line, the function the library provided took as arguments the text, font, font size, width and height of the text cell, alignment, and the color of the text, using RGB values. For images, it was simpler, as it only needed the image, its width and height, and the coordinates of the top left corner of the image.

This library, even if not simple to use, proved to be very useful as it allowed precise placement of the elements in the PDF, especially images since sometimes the text lines would be somewhat misplaced. With enough fine tuning, the Python script was capable of generating a full assay report for the user with the protocol description and the final result.

5.1.5 Python Script Conclusion

The Python script was only made to be used for a short time. Its only purpose was to help STAB VIDA's staff develop the lactose intolerance test using the already-made DVP device and show how the process works. It was not meant to be commercialized or even be available to the public.

With the satisfactory results obtained from the script, which will be discussed in the next chapter, STAB VIDA demonstrated the concept of the device executing the lactose intolerance test, and it was a matter of developing the mobile application to go along with it.

The script had one final purpose throughout the development of the application. Any time a result had to be validated, it was compared to the result that was obtained from

the script, as this was already proven to work.

5.2 Application to Execute Melt Assay

The application was the last component to be developed during the thesis. It involved less work than the Python script since it was an upgrade to an already existing application. This being said, it still had some complications, and some changes had to be made. This section will present the work done in the application and the nuances that affected it.

5.2.1 State Management

The first step was to add the new state management components to the ones that the device already contained. These are the states in blue in the general schema State Module, Figure 3.1.

Without this, the application could not recognize the states and therefore could not redirect to the correct page. This was because, as explained in the state of the art, the application associated a component with each state the device was in. These components were responsible for displaying in the application the correct pages to handle the behavior of the pocket relative to each state. Only after adding the new states to the state management component could the development process advance to creating the different components for each state and improving the Bluetooth module to handle the new features the device had.

The application's main service, called *assay-session* was responsible for controlling the display of different pages throughout a single assay. Every assay was associated with a session. The session performed a variety of tasks, including observing device behavior and deciding what to do next based on the signals received, initiating an assay by gathering the necessary information, storing data segments, and issuing commands to various sections of the application.

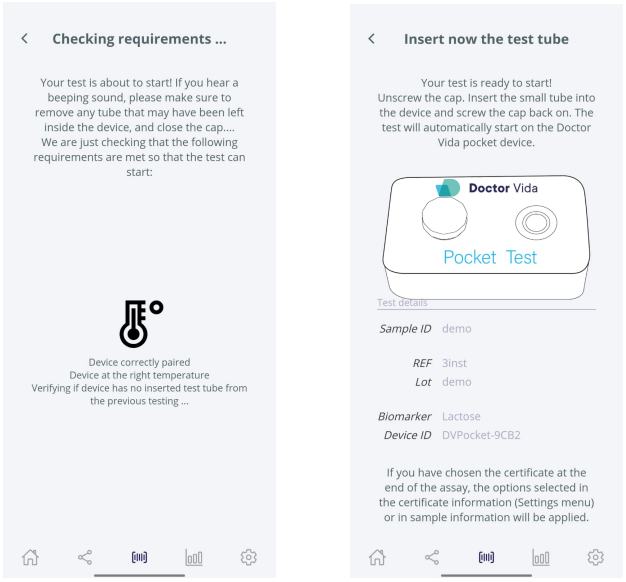
The session's first job was to obtain the assay information from the user. This includes what device to use, the sample information, and the protocol information. This part was kept equal to what already existed; the only new thing was that there existed a new protocol format that had to be parsed in a new way. In order to not change what was already being used, the new protocol incorporated the old fields, giving a *null* value to the ones that did not matter. This way, the old parsing method could be reused, adding only the new fields that previously were not accounted for.

After the user confirmed that he wanted the assay to go forward, the assay session would begin preparations to execute the assay. This included creating instructions from the protocol, much like the DVP device did, calculating the number of entries, and creating the data structures to store the assay data obtained from the device. The application already knew if the protocol it was about to execute was the melt protocol or the old

COVID-19 protocol. At this point only the melt protocol is relevant, so it will be assumed that this was the protocol used.

The final step of the assay setup involved subscribing to multiple device characteristics, as will be explained in the next section.

Once this was complete, the application actively listened for notifications from the device and decided what to do based on them. Each state in Figure 4.12 starting with the **Preheating State** had a corresponding screen. This state was the first to display a page on the application after the assay started. The **Build Instruction State** was the actual first state of the device that the application recognized, but since it only prepared the device for the assay, it was decided not to add a screen as it would only be on display for a second at most and would not bring any new information to the user.



(a) (b)

Figure 5.10: Application Screens: a) DoctorVida application assay preheating, b) DoctorVida application screen for waiting the tube.

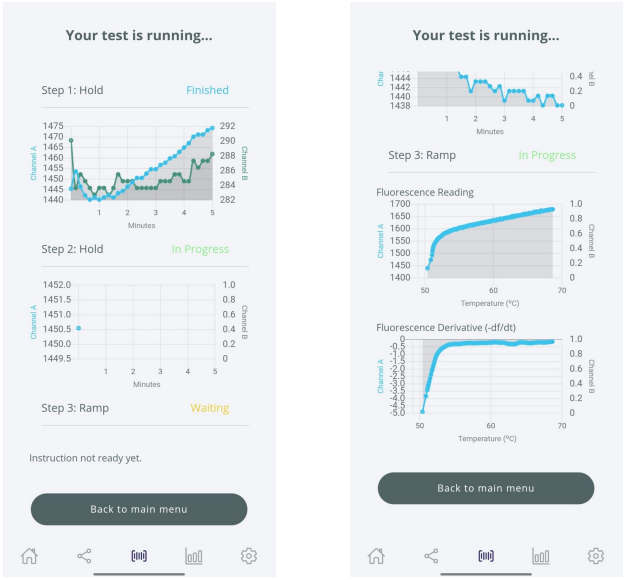
The **Preheating State** application page, Figure 5.10a, was displayed when the device was warming up to the temperature of the first instruction. This was the same as the old preheating screen.

Once the temperature of the first instruction had been reached, the assay session would detect that the device was no longer in **Preheating State** and advance to the next state.

This was the **Wait For Tube State** since the **Ready State** was ignored by the application because it was only used in the firmware side for printing some informative text and logging. The only connection that this state had with the application was that it waited for a signal indicating when to advance to the next state.

The next state, which was the previously mentioned **Wait For Tube State**, had one

function. This state waited for the user to insert a new tube. The application would display a screen giving instructions to the user on how to proceed, Figure 5.10b. As soon as the device detected the new sample tube, it advanced to the next state. This was the **Set Temperature State**, but since the device had already been preheated to the correct temperature, this state was skipped and the first instruction started almost immediately.



(a) (b)

Figure 5.11: Application Screens: a) DoctorVida application hold graphs, b) DoctorVida application ramp graphs. Both of this images are taken from the same application page.

The application screen for the **Set Temperature State**, **Hold Instruction State** and **Ramp Instruction State** was the same and consisted of a set of graphs, one for each temperature hold instruction and two for each temperature ramp instruction (Figure 5.11). The colors of the graph lines followed the same principle as the Python script, where they would have the color of the excitation source LED, making it easier to distinguish the fluorescence readings.

The hold instruction only required a single graph to display the evolution of the fluorescence levels throughout its duration. On the other hand, the ramp instruction made use of two graphs, one for the same purpose as the hold instruction graph and another to display the processed data, with normalization, derivative, and smoothing applied, just like in the real-time and data analysis of the Python scripts.

Every instruction was displayed on this screen, even the ones that had not yet been executed. This meant there had to be some distinction between the progress of each instruction. For this, it was decided that next to each instruction there would be a text indicating if it had finished executing, was currently executing, or if it was going to be executed in the future (see Figure 5.11a). This, along with the progress bar displayed at the top of the screen, gave the user a better notion of the progress of the assay and how

the test was going.

During each instruction, the application would request data segments at the pace defined in the protocol. Every time it obtained data from the device, it would store it locally in the designated data structure and send the new information to the backend where it could be accessed using the API.

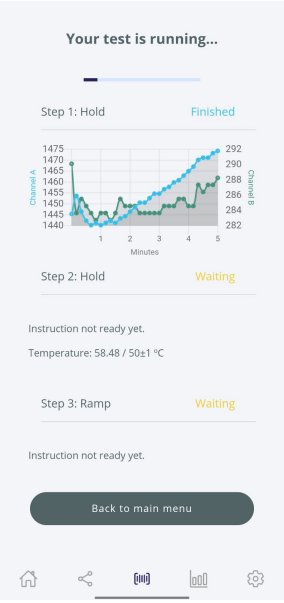


Figure 5.12: Application page with the graphs and the temperature display in real-time as the device is setting the temperature for the next instruction.

Every time an instruction was completed and there was still more to execute, the device would enter the **Set Temperature State** to set the correct initial temperature for the next instruction. The application would remain on the same page with the graphs of each instruction, but during this state it displayed below the next instruction title the current temperature of the device (Figure 5.12), which updated every second. This helped the user understand how long it would take to reach the expected temperature.

Finally, once the assay was completed, the device advanced to the **Assay Finish State**, this allowed the application to finish requesting any segments that it might need and/or resend to the backend any segments that did not make it to the data base.

Once every data entry was stored in the backend, the application was ready to analyze the data and return a result to the user. This entailed developing a service specifically for this purpose, as well as a new screen that could display the various graphs and results associated with the melt procedure (Figure 5.13). This will be discussed in a future section in this chapter.

To complete the assay, the application would store the result in the database and send the final segment to the device; this would reset the device back to the **Init State** where it waited for a new protocol. It would then store the assay data segments in a CSV in the file system of the mobile phone in case the raw data was needed for other purposes.

The application's final screen would be the results screen, and at that point the user knew the assay was completed. Using the same application and device, a new assay could be run, and the previous result was saved in the result list screen so that it could be accessed again later.

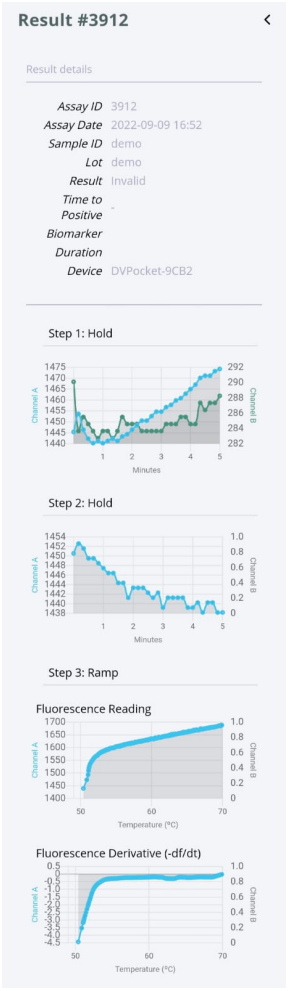


Figure 5.13: Full extent of the result page, normally the user would have to scroll down to fully see this page. The protocol used was just for demonstration, thus the Invalid result.

5.2.2 Bluetooth Module

The Bluetooth service in the application handled every aspect of communication with the device via Bluetooth (Figure 3.1). Since the device had up to three communication permissions in the form of read, write and notify, the application had to deal with each of them to be able to fully exchange data with the device. The read and write communication permissions came with a callback that would be triggered once the characteristic was called using either the read or write.

The *write* permission allowed the application to send information to the device, which would handle it in a particular way using its own characteristics callbacks. An example of

this was the writing of the protocol parameters. First the application had to convert the protocol information into bytes in order to send them via Bluetooth. This was done for each field in the protocol, with each of the bytes of each field being appended to the end of the byte buffer. Only when the protocol was completely converted to bytes could the application send this information to the device, which would trigger the callback for the write permission of the characteristic responsible for handling the protocol parameters.

Another example was the request for the data segments. This involved using a characteristic that had both write and read permissions. The application would request a particular data segment, writing to the device the order of said segment. This would trigger the write callback of that characteristic that would prepare the information to respond to the application and store it as the value of the characteristic, explained in section 4.2.2. Once this was completed the application would request to read the value of the same characteristic, which would contain the data segments requested previously.

The *notify* permission was the most complicated to handle since it involved listening to the device and actively taking action once a signal was received. But some of the most important functionalities of the application relied on this type of communication. Examples of these notifications are the state change notifications, the progress notifications, and the temperature notifications.

The way the application approached this was by using Observables from the RxJS library[49]. This allowed for asynchronous responses to signals from the device. An Observable is a data structure that can receive values using the *next()* function and emit them to any observers. An observer is a consumer of values produced by the Observable it subscribed to and will execute a callback upon detecting that the Observable has emitted a new value. This is useful since the main code of the application can declare that it will observe an Observable and continue the execution of the main process knowing that, when a new value is emitted by the Observable, an asynchronous piece of code will execute in response.

In reality, the application used the Subject component, which functions just like the Observable but allows for multiple observers to subscribe and listen for emissions of values. Using this, it was possible to asynchronously listen for signals emitted by the device.

The application created a Subject for each characteristic that produced notifications, and every time the device notified a new value, the Subject would call the *next()* function using this value, effectively triggering the callbacks from every listener of the Subject.

A good example of this was the state Subject. The Bluetooth component would create the Subject and every time a new assay session started the assay, it would subscribe to that Subject. Whenever there was a state change, the device notified the new state code. This notification would be detected by the application, and the Subject would emit the new state value. This in turn would cause the callback of the subscription to execute. In this case, the callback executed a switch that would run a piece of code depending on the state code received. This was used for setting flags and updating variables in the assay

session; in the case of the state Subject it was used to decide what was the correct page the application should display.

A simpler example was the temperature Subject, here the application would just store the new temperature value notified by the device.

The Bluetooth component was arguably the most important component to be developed and had to guarantee that the application was able to follow the development of the device's processing of the assay. This is why some problems arise if the application is taken out of range of the device, which were already resolved by the previous version of the application.

5.2.3 Result Analysis Module

The final step before finishing an assay was to calculate the result to give the user. For this, there was an attempt to translate the process that was developed for the Python script into the backend so it could provide the application with the final results, as was done with the initial application. The problem was that the backend used Symfony, which is a PHP framework, and there were no libraries that could provide the same analysis of the temperature ramp instruction as the Python libraries used in the script.

It was decided that until there were changes to the backend to be able to run Python code in a Python environment, the processing and calculation of the final result would be done in the application and stored in the backend.

The temperature hold analysis was simple to translate from the Python script as it did not involve any external libraries. In fact this analysis could have been done in the backend straight from the start but it was decided to keep it in the application not to fragment the analysis and later translate the full process to the backend.

The temperature ramp instruction had some problems that required adapting the analysis algorithm. The first step of processing the data was identical to the Python script. The application would normalize, calculate the derivative, and apply smoothing to the ramp instruction data, but when it came to detecting the peaks, there were some complications. All the tested libraries for typescript[4] failed to replicate the results obtained in the Python scripts as they did not provide the same functionalities and were a much simpler form of peak detection, not viable for the usage that was intended in the ramp analysis.

This prompted STAB VIDA to change the approach, and it was suggested that the peak detection be changed into a simpler form that would ignore the problems the typescript peak detection libraries were suffering. The new analysis method would use a predefined threshold to indicate if there was a peak or not. The script would save the point at which the curve crossed the threshold the first and second time, and this would count as a peak. By calculating the middle point of these two points, the highest point of the peak was

[4]Ionic with Angular used typescript as the development language

obtained, and this was used in the analysis, Figure 5.14. The ascending portion of the line that was always present at the beginning of the graph was ignored.

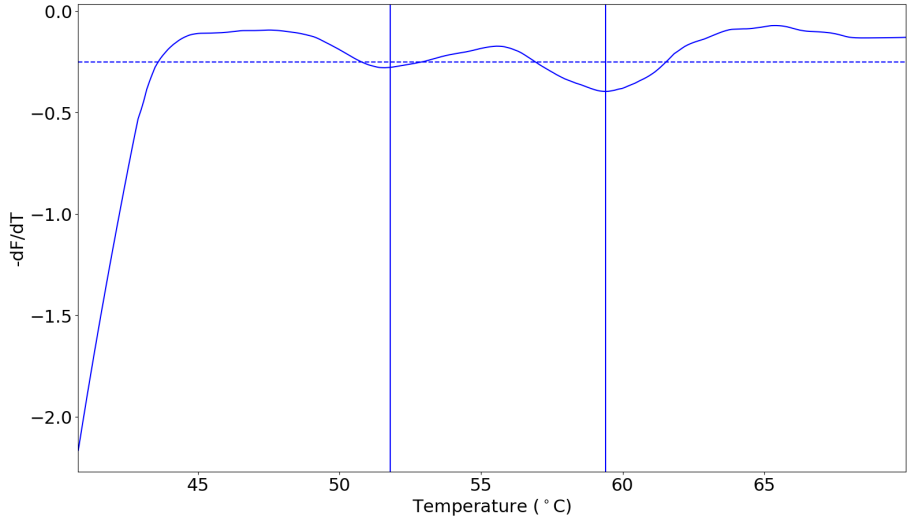
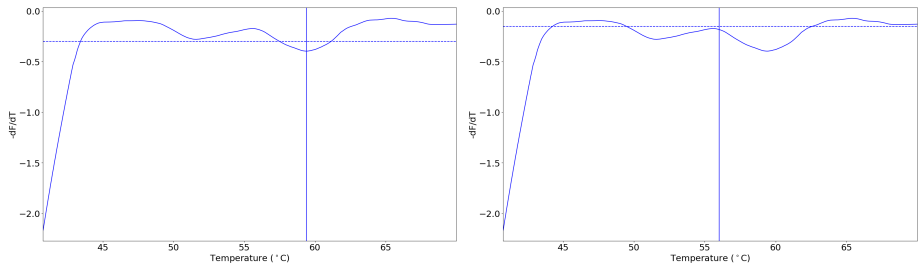


Figure 5.14: Example of a correct analysis made by the application algorithm.

The problems with this method were easy to spot. If a peak never crossed the threshold, it would not be accounted for, which could affect the outcome (see Figure 5.15a). Another problem would occur if there were two peaks; if the valley between them never crossed the threshold, only one peak would be saved, Figure 5.15b.



(a) (b)

Figure 5.15: Application analysis problems: a) Example of an incorrect analysis made by the application due to one of the peaks not being detected, b) Example of an incorrect analysis made by the app due to the valley between peaks not being detected.

This method worked most of the time and was good enough for an initial prototype of the application that was used by STAB VIDA representatives in an exposition. The final product should have the analysis in the backend, where it has a Python environment running a version of the initial Python script dedicated to analyzing the data obtained from the device. This was not worked on during the thesis as backend development was outside the scope.

5.2.4 Application Conclusion

The application developed during the thesis was the final objective that was proposed by STAB VIDA. Although it still has some details that need work, the final application could execute the full melt procedure. Starting by obtaining the protocol from the backend, it would send it to the device, thus starting the process on it. The application would then track the progress while displaying real-time graphs of the data obtained from the device. By the end of the assay, the application would calculate and display the result page for the user.

In addition to the initial objectives that were achieved, there were also new ones that were suggested during development, such as the temperature characteristic for tracking the temperature of the device and displaying it in the application.

All the work done in the application did not prevent it from executing the previous protocol for COVID-19; this way there was retro-compatibility with devices that only possessed the old firmware.

The final application was used by STAB VIDA personnel on multiple occasions, providing various results that are going to be discussed in the next chapter.

6

Results and Discussion

This chapter intends to present the successful results obtained and the problems that were noticed throughout the thesis.

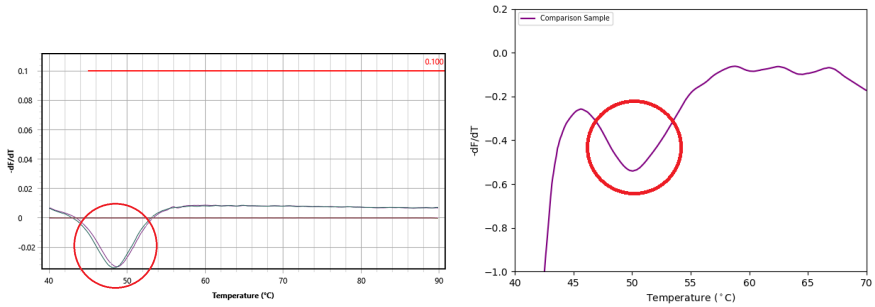
The results that are discussed in this chapter are split into two main subjects, the device results and the application results. The first is more of an overview since most of the results were displayed in the firmware chapter 4. It made sense to do this since it supported the information presented there. On the other hand, the Python and application results are mostly presented here; nonetheless, some of the graphs and results obtained might be displayed in the software chapter 5, where they were used to exemplify some aspects of the software.

6.1 Pocket Device

The testing of the DVP device firmware was a very anticipated part of the thesis; it would tell if the work that was invested into the firmware would really pay off. Some tests were previously made in some device prototypes designed to execute specific melt protocols, but with the new firmware, it was possible for the user to define its own protocol and execute an assay on the device.

There were some initial tests to confirm that the device could in fact produce results comparable to those of a laboratory machine designed for this purpose. Figure 6.1 shows a comparison of the results obtained in a laboratory machine configured to run LAMP tests and the DVP device. There were minor variations in the results, but these were to be expected since the sample was used twice for this comparison, therefore the enzymes had more time to amplify the DNA in the second assay. Furthermore, the sensors and firmware were different, which might affect the result. Despite all this, it is possible to see that the result given by both machines was mostly the same, where the negative peak in both cases is around the same temperature value of 50ºC.

The pre-processing made by the software of the laboratory machinery was different from that of the software used by STAB VIDA to produce that graph, but nevertheless it is possible to see that the peak occurred in the same area, which was the only requirement



(a) (b)

Figure 6.1: Comparison of results obtained: a) Result obtained in a laboratory machine prepared to execute LAMP assays, b) Result obtained from the pocket device for the same sample.

for the analysis to match.

The firmware could only be tested by using either the Python script that was created soon after the firmware completion or the mobile application, which came later in the thesis development. For a few weeks, the Python script was used by STAB VIDA staff to test and tune the reagents needed for the lactose intolerance assay, therefore, it makes sense to approach this topic first.

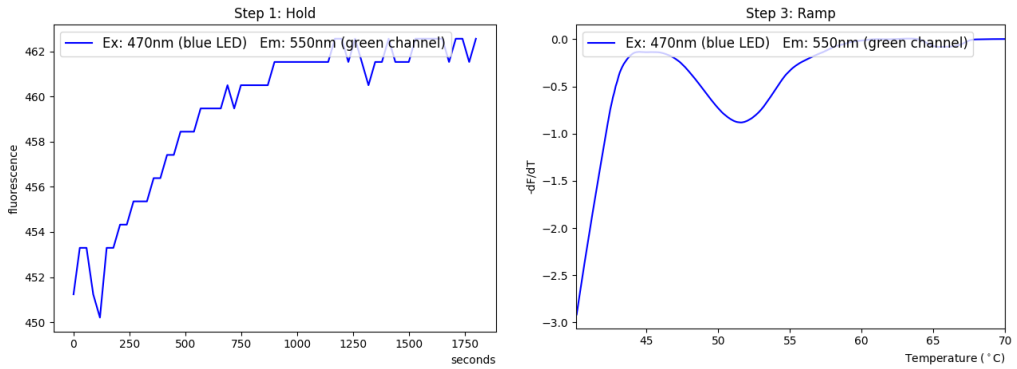
Most of the result were presented in the firmware chapter (see chapter 4) since it allowed to visually support the information presented there, therefore they are not presented in this section.

6.2 Python Results

The Python development was brief and enabled STAB VIDA to start testing the device. This was requested since there was going to be a presentation of the device to a partner company and some form of visual component was needed to enable the participants to see the progress of the assay. The first result obtained using the Python script used a sample provided by one of STAB VIDA's employees.

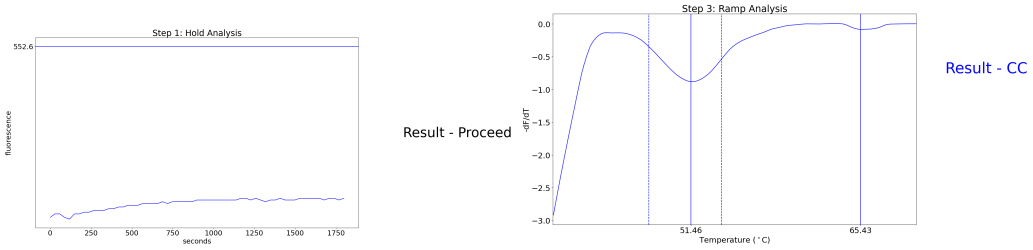
The assay duration was roughly one hour and fifteen minutes. During this time two graphs were shown displaying the progress of the assay (Figure 6.2) and this gave a trained professional in the biochemistry area a rough idea of what the result would be even before the analysis made by the script.

Once the assay was finished, these two plots were saved in the file system, and two new ones were created showing the result of the analysis for each particular instruction (Figure 6.3), with the result of the last instruction also being the final result of the assay. The analysis of the first hold was set to return *Proceed* every time, since what STAB VIDA was interested to see was the ramp instruction analysis, explained in section 5.1.3.1.



(a) (b)

Figure 6.2: Assay graphs: a) Real-time graph of the first temperature hold instruction, b) Real-time graph of the temperature ramp instruction.



(a) (b)

Figure 6.3: Assay graphs with the analysis on the right side: a) Analysis graph of the first temperature hold instruction, b) Analysis graph of the temperature ramp instruction.

From what is possible to see in the figure, the result was a CC, meaning that the subject suffers from low levels of lactase production. The peak position is the only factor the script takes into consideration when deciding the result in the lactose intolerance protocol, since in this case the peak was between 48ºC and 54ºC[1] the result calculated by the script was a CC. This result was later confirmed by DNA sequencing analysis.

When it came to the demonstration, a small error occurred during the analysis of the temperature ramp instruction, which produced an incorrect result. The result could be deduced to be a CC by just looking at the graph, but the script incorrectly detected a negligible peak in the 56ºC to 64ºC range[1] and assumed there were two peaks in the expected zones, thus outputting the CT result, which was incorrect, see Figure 6.4.

This problem was caused by a complication when detecting the peak, explained in section 5.1.3.3. This problem was later resolved when the analysis portion of the script was revised and improved. Once this was done, the data obtained in the demonstration, which previously originated the wrong result, was once again tested with the data that

[1] Explained in section 5.1.3.2 the ranges were changed, and the initial version of the Python script used a different range.

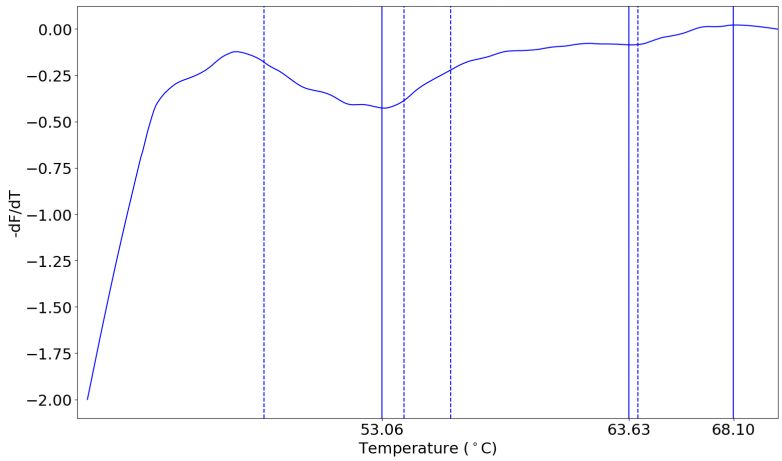


Figure 6.4: Ramp analysis with the incorrect peak detected in the 56ºC to 64ºC range (63.63ºC), changing the result to a CT.

was stored, delivering the correct analysis that time around, Figure 6.5. Fortunately, this problem was not very impactful since it was indicated that the analysis was still being improved and what really mattered was the presence of the peak in the graph.

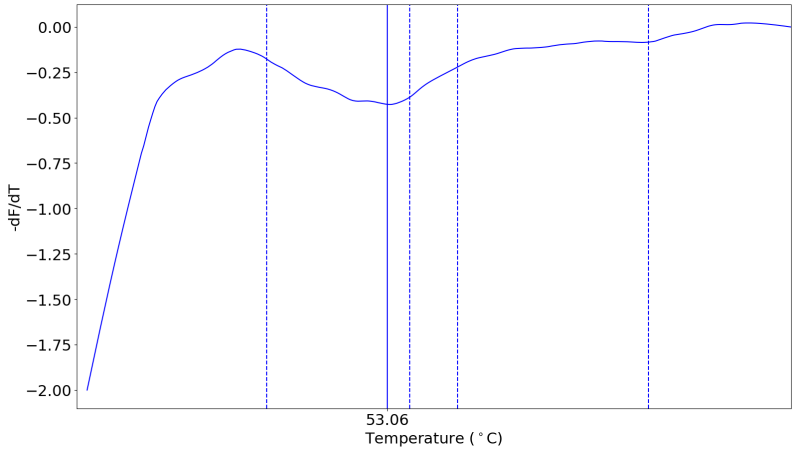
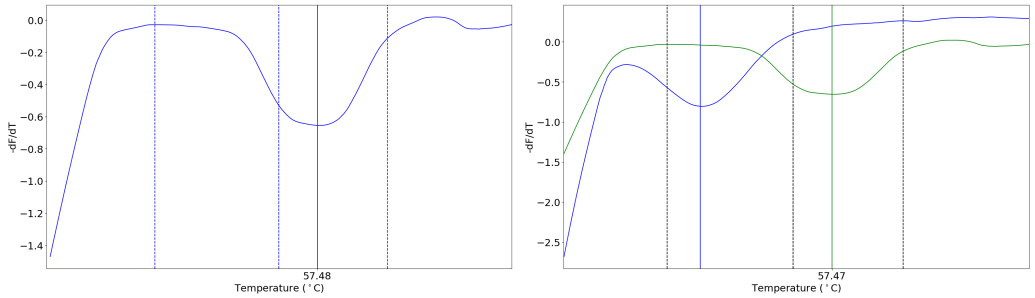


Figure 6.5: Ramp analysis where only the correct peak is detected at 53.06ºC.

Besides the CC result it is interesting to discuss the other two results. The other single peak result was the TT. In this case, the peak occurred between 55ºC and 62ºC, this way there was exclusivity between the CC and TT peaks, preventing ambiguous results, Figure 6.6.

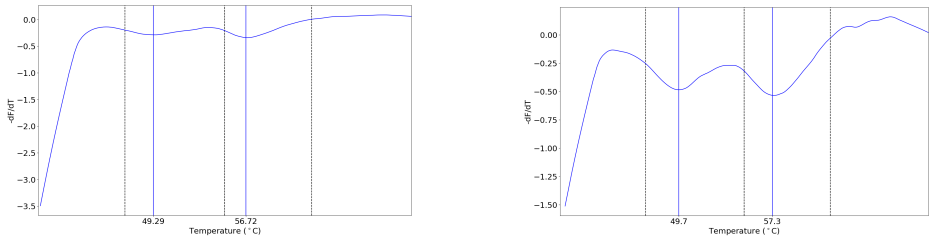
The only double peak result that could be obtained from the lactose intolerance test was the CT result, which possessed a peak in both of the intervals referenced in the previous two results (Figure 6.7).

This were the three possible correct results, but even after the peak detection parameters were improved, there were still some invalid or incorrect results. These included the



(a) (b)

Figure 6.6: Assay graphs: a) Ramp analysis with a TT result, b) Comparison of the CC (47-55ºC) and the TT(55-62ºC) peak ranges.

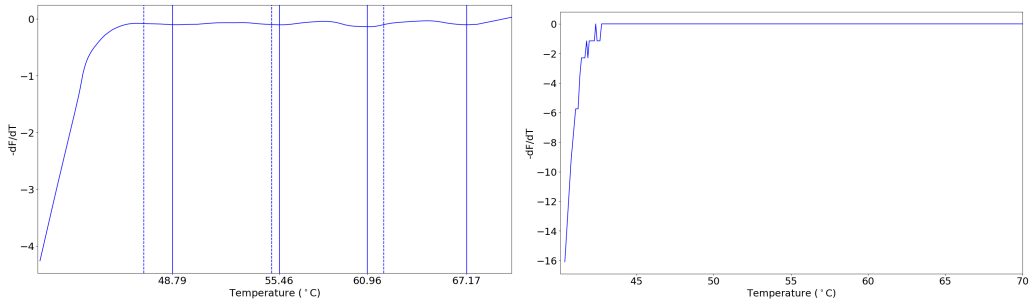


(a) (b)

Figure 6.7: Assay graphs: a) Graph with a CT result with less noticeable peaks, b) Graph with a CT result with the most substantial peaks.

fluorescence readings not having substantial peaks (Figure 6.8a), the data producing an invalid curve (Figure 6.8b) or a peak that was not detected by the analysis due to some problem with the data, see Figure 6.9.

Unfortunately, this could be caused by multiple reasons that may or may not be related to the informatics part of the process. The DNA may not have been amplified enough, or the sample may have been incorrectly taken, or even the reagents might have expired. This was more common when the sample was a CT because the peaks were usually smaller due to there being two types of nucleotides instead of one, as was the case with the CC and TT. On the other hand, some of the mistakes were caused by the software, mainly the peak detection component, which was the component most prone to errors.



(a) (b)

Figure 6.8: Assay graphs with incorrect analysis: a) Ramp instruction analysis with negligible peaks detected, b) Invalid fluorescence readings that generated a flat line.

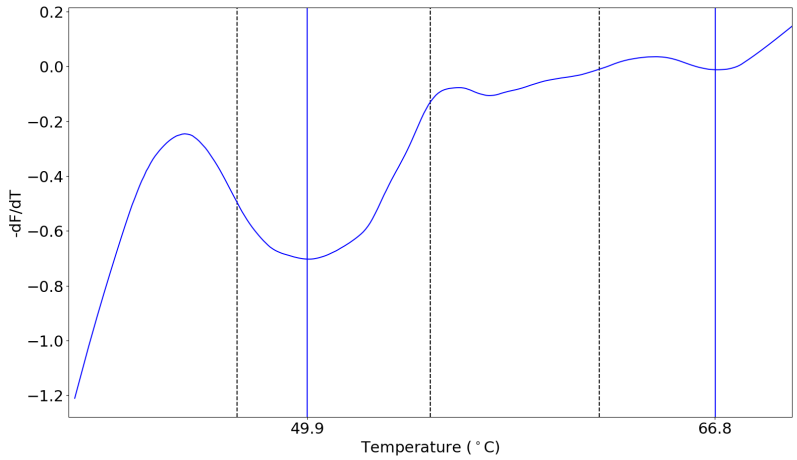


Figure 6.9: Ramp analysis with a missing peak that originated a CC result when it should have been a CT (according to laboratory analysis).

6.3 Application Results

Upon completing the Python script, there was a period of a few weeks where it was improved and used for testing multiple samples. This was a period of correcting eventual bugs both in the firmware and the script. Then it was decided by STAB VIDA that the efforts would be focused on developing a new version of the already existing application to be compatible with the device, thus being able to execute melt protocols. This was discussed in chapter 5, but what matters most for the results is the analysis, which was changed in the application development due to software differences, explained in section 5.2.3.

This changes to the analysis might generate different results than those that would be obtained with the Python script. This was considered acceptable by STAB VIDA until the analysis that was made in the Python script could be transposed to the application backend.

There were several assays executed in the application as it was important to test the

analysis since the application was going to be presented at an international technology fair alongside the device and other STAB VIDA products. The amount of assays executed were not enough to make a conclusive statistical reasoning, but the outcome of the ones that were executed indicated that the results could be trusted.

Each possible outcome was tested several times to ensure that the application could produce the correct result for each of them. The CC result (Figure 6.10a) was the first to be tested and also the most frequently obtained result since it was the most common among STAB VIDA employees, which were the main test subjects for testing the application. The TT result was obtained less frequently, Figure 6.10b.

Both the CC and TT results had few mistakes when analyzed since the peaks were clear and the analysis was simpler than the Python script using only a threshold to detect peaks, which had the upside of not detecting incorrect peaks that could influence the result.

(a) (b)

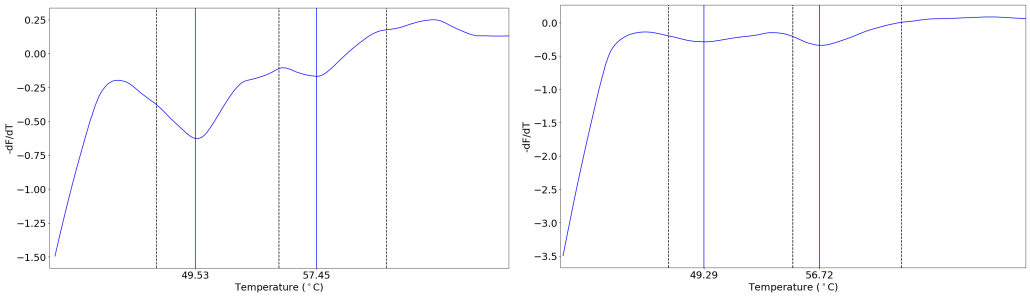
Figure 6.10: Application result page: a) Ramp analysis with a CC result made by the application, b) Ramp analysis with a TT result made by the application.

The CT result proved more error-prone. This had to do with the height of the peaks in comparison to that of the threshold, which sometimes caused one of the peaks to not be detected, changing the outcome of the analysis. Two examples of this can be seen in the graphs in Figure 6.11 which should have received the CT result but were attributed with different results instead. By taking the raw data that was stored in the mobile device at the end of these assays and executing the analysis on the Python script, it was possible

to see that the correct results should have been a CT, Figure 6.12. This was also later confirmed by DNA sequencing were the CT result was obtained, confirming the analysis made by the Python script.

(a) (b)

Figure 6.11: Application result page: a) Ramp analysis with an incorrect CC result when it should have been a CT result, b) Ramp analysis with an incorrect TT result when it should have been a CT result.



(a) (b)

Figure 6.12: Assay graphs from Python script: a) Analysis of the assay data from Figure 6.11a with the correct CT result from the Python script, b) Analysis of the assay data from Figure 6.11b with the correct CT result from the Python script.

6.3.1 Feedback and Conclusion

The application and the device were taken to an exposition at an international technology fair where passing individuals could take a lactose intolerance test guided by the STAB VIDA personnel present on site. For the duration of the fair, some problems and suggestions emerged. The problems were mainly related to the communication between the application and the backend, which was frequently taking longer than usual and would suffer from connectivity issues. This is outside of the scope of the thesis, but in short, it had to do with the method used for sending data to the backend, which used to function normally with the COVID-19 test but failed more often with the melt test, mainly due to the amount of data, which was many times larger. The other problems that the application had were related to the analysis, where one of the results generated an invalid result when in reality it was a CC.

There were also some suggestions made by the on-site team that were later implemented. The most relevant was the temperature tracking component that can be seen on the graph page during the assay on the application. This involved creating a new characteristic on the DVP device and a corresponding Observable on the application side to keep track of the changing temperatures, described in section 5.2.2.

The feedback given by STAB VIDA employees about the application described both the good and the bad aspects. For the good aspects:

- The graphs on the Assay Progress screen gave useful information about how each instruction was progressing.
- The colors displayed by the device's LED proved very helpful to understand the state the device was in.
- On the application progress screen, the use of colors to show the different parts of each instruction was helpful.
- The real-time graphs of each instruction were the main point of interest as they provided visual feedback of the assay's progress.

There was an attempt to fix or at least mitigate the downsides described by STAB VIDA employees, but some were still cause for complaint. Some of the downsides described are the following:

- One of the main problems that was not possible to fix during the thesis was the time it took for the device to cool down. Most of the time, cooling down took 20 to 30 minutes, but on some occasions this could go on for almost an hour.
- The application occasionally lost track of the progress of the device and would not update the graphs or advance to the results screen.

- The upload of the assay data took a long time and sometimes failed on the first attempt, requiring a retry from the beginning.

The technology fair served as the first real-life scenario of using the new application and device, and it provided useful feedback to improve the application, which happened for the remaining duration of the thesis. By the end of the thesis, the application was as described in section 5.2, it still suffered from connectivity issues and will still be subject to changes as improvements and new functionalities are going to be added in the future.

On the device side of things, there were not that many changes once the firmware was finished; they were mostly small ones requested by STAB VIDA. By the end of the firmware development, the changes that were made revolved mostly around the new states and new protocol management, with little changes to the functionality of the PID controller, due to there being no significant improvements from the tests that were executed, explained in section 4.1.3; these tests included both manual and Ziegler-Nichols (see section 2.6.2) tuning to ensure that there were no missed configurations that could improve the temperature control. The same tests demonstrated that the device was unable to reach temperatures above 80ºC reliably, which could influence the results. This limited the assays the device could execute.

The final device was capable of executing any type of assay that involved temperature control and fluorescence reading as long as it complied with the device limitations; this included the COVID-19 test, which was the procedure the initial device was designed for. By the end of the thesis, STAB VIDA was already discussing new types of tests to execute on the DVP device, since the lactose intolerance test showed promising results.

7

Conclusion

STAB VIDA proposed the development of firmware for a device that had to be able to control temperature in a way that enabled it to execute assays with complex protocols requiring multiple temperature variations while maintaining a small, transportable form factor.

This was based on a device designed and manufactured by STAB VIDA for the COVID-19 test, which required only a constant temperature. It had to be able to take accurate fluorescence readings despite the fluctuating temperatures, as well as retain all of its original features. Bluetooth was used to communicate with any peer with whom the device was paired. Thus, it was able to transmit the data to the processing-capable software.

The device was only half of the thesis; it was also proposed the development of software capable of communicating with the device and tracking its progress, presenting the data obtained to the user, and providing an analysis of the assay. Initially, this was accomplished by a Python script entirely developed during the course of this thesis, and later by a mobile application built on top of the initial application that worked with the initial device, both of which were created for the COVID-19 test.

Upon completion of development, STAB VIDA employees were able to carry out a variety of tests utilizing the pair device-application. These were lactose intolerance tests and sought to identify the gene responsible for this condition. The obtained results demonstrated that the device was capable of consistently and accurately detecting the gene, and they corresponded with the laboratory analysis.

The accomplishments of the device's firmware and application validated the development efforts and inspired new ideas and possibilities for the device. By the conclusion of the thesis, STAB VIDA was already considering expanding the device by improving and adding new features that would enable the execution of new types of assays.

7.1 Future Work

At the conclusion of the dissertation, STAB VIDA considered multiple enhancements that would be advantageous to the device. It was known that the performance of the device was hindered by several factors, the cooling time being the most significant. This was a major complaint of the users, since the average cooling time from 65ºC to 40ºC was approximately 30 minutes.

With this issue in mind, one of the future enhancements STAB VIDA may invest in is a cooling system that reduces some of the total assay time, which would be a significant improvement. This would not necessitate any modifications to the application, as it would only impact the duration of the test.

Another change STAB VIDA was considering was to add WI-FI functionality to the device and enable it to communicate directly with the backend. This would allow the user to run assays on a device while not being close to it, but also raises some complications that STAB VIDA would have to solve.

At the conclusion of the thesis, STAB VIDA had begun developing a new prototype per client request. This new device will feature modifications to its hardware and firmware. It will add new hardware features that will enable the device to execute two assays concurrently, with the intention of testing the assay sample and a control sample simultaneously. It will use the firmware developed during the thesis as a foundation, but it will be modified to accommodate the hardware additions. Because this will result in a completely redesigned device, it is expected that the application will also undergo significant changes.

Many changes can be applied to the DVP device as it is simple to access and change the hardware components if done by a trained professional. This means that in the future, there could be various iterations of the initial device with different purposes to fit a specific need. The device developed for this Master's thesis was an example of this.

Bibliography

[1] *Usage Statistics of Content Languages for Websites*. 2021. URL: https://covid19.who.int (cit. on p. 1).

[2] *Nucleic Acid Amplification Tests (NAATs)*. 2021. URL: https://www.cdc.gov/coronavirus/2019-ncov/lab/naats.html (cit. on p. 2).

[3] B. Ferreira. "Rapid tests for Covid-19: what are the differences?" In: (2020). URL: https://www.hospitaldaluz.pt/en/health-wellness/rapid-tests-covid-19-what-are-differences (cit. on pp. 2, 3).

[4] C. A. et al. "A molecular test based on RT-LAMP for rapid, sensitive and inexpensive colorimetric detection of SARS-CoV-2 in clinical samples". In: (). URL: https://rdcu.be/cGTPF (cit. on p. 3).

[5] D. M. Swallow. "Genetics of Lactase Persistence and Lactose Intolerance". In: *Annual Review of Genetics* 37.1 (2003). PMID: 14616060, pp. 197–219. DOI: 10.1146/annurev.genet.37.110801.143820. eprint: https://doi.org/10.1146/annurev.genet.37.110801.143820. URL: https://doi.org/10.1146/annurev.genet.37.110801.143820 (cit. on p. 4).

[6] *PCR Troubleshooting*. 2021. URL: https://www.bio-rad.com/en-pt/applications-technologies/pcr-troubleshooting?ID=LUSO3HC4S (cit. on p. 5).

[7] G. Doria et al. "An isothermal lab-on-phone test for easy molecular diagnosis of SARS-CoV-2 near patients and in less than 1 hour". In: *International Journal of Infectious Diseases* 123 (2022), pp. 1–8. ISSN: 1201-9712. DOI: https://doi.org/10.1016/j.ijid.2022.07.042. URL: https://www.sciencedirect.com/science/article/pii/S1201971222004362 (cit. on p. 8).

[8] L. Leonardi, G. Patti, and L. L. Bello. "Multi-Hop Real-Time Communications over Bluetooth Low Energy Industrial Wireless Mesh Networks". In: *IEEE Access* 6 (May 2018), pp. 26505–26519. ISSN: 21693536. DOI: 10.1109/ACCESS.2018.2834479 (cit. on pp. 9, 11, 12).

[9] C. Gomez, J. Oller, and J. Paradells. "Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology". In: *Sensors (Switzerland)* 12 (9 Sept. 2012), pp. 11734–11753. ISSN: 14248220. DOI: 10.3390/s120911734 (cit. on p. 10).

[10] T. Instruments. *BLE-Stack User's Guide*. URL: https://software-dl.ti.com/lprf/simplelink_cc2640r2_latest/docs/blestack/ble_user_guide/html/ble-stack-3.x-guide/index.html (cit. on pp. 10, 11).

[11] A. Kevin Townsend Carles Cufí and R. Davidson. *Getting Started with Bluetooth Low Energy by Kevin Townsend, Carles Cufí, Akiba, Robert Davidson*. URL: https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch04.html (cit. on p. 12).

[12] kolban. *BLE2902 Class Reference*. URL: http://www.neilkolban.com/esp32/docs/cpp_utils/html/class_b_l_e2902.html (cit. on p. 12).

[13] Yulong-Espressif. URL: https://github.com/espressif/esp-idf/issues/1087 (cit. on p. 13).

[14] D. Geevarghese. *BLE vs Wi-Fi: Which is Better for IoT Product Development?* 2018. URL: https://www.cabotsolutions.com/ble-vs-wi-fi-which-is-better-for-iot-product-development (cit. on p. 13).

[15] K. Shahzad and B. Oelmann. "A comparative study of in-sensor processing vs. raw data transmission using ZigBee, BLE and Wi-Fi for data intensive monitoring applications". In: *2014 11th International Symposium on Wireless Communications Systems (ISWCS)*. 2014, pp. 519–524. DOI: 10.1109/ISWCS.2014.6933409 (cit. on p. 13).

[16] URL: https://platformio.org (cit. on p. 15).

[17] SparkFun. URL: https://github.com/sparkfun/SparkFun_AS726X_Arduino_Library (cit. on p. 15).

[18] A. Tech. URL: https://github.com/AdysTech/PIDPWM (cit. on p. 16).

[19] Sparkfun. *Pulse Width Modulation*. URL: https://learn.sparkfun.com/tutorials/pulse-width-modulation/duty-cycle (cit. on p. 17).

[20] S. Solutions. *Understanding Duty Cycle and Phase Angle in Quadrature Sensors*. URL: https://sensorso.com/resources/understanding-duty-cycle-and-phase-angle-in-quadratic-sensors.html (cit. on p. 17).

[21] J. DiStefano. *Feedback and Control System*. 1989. ISBN: 0070170452 (cit. on p. 18).

[22] J. Maggay. "Module-3-Introduction-to-Feedback-and-Control-System". In: (). URL: https://www.scribd.com/document/529625795/Module-3-Introduction-to-Feedback-and-Control-System (cit. on pp. 18, 20, 21).

[23] A. Dourado. *Modelização de sistemas por equações diferenciais* (cit. on p. 19).

[24] D. Meadows. *Leverage Points Places to Intervene in a System* (cit. on p. 20).

[25] Electrical4U. *On Off Control Controller: What is it?* URL: https://www.electrical4u.com/on-off-control-theory-controller/ (cit. on p. 21).

[26] C. D. H. Williams. *Feedback and Temperature Control*. URL: http://newton.ex.ac.uk/teaching/CDHW/Feedback/index.html (cit. on pp. 21–24).

[27] G. C. Goodwin, S. F. Graebe, and M. E. Salgado. *CONTROL SYSTEM DESIGN*. 2000. ISBN: 0139586539 (cit. on pp. 22, 23).

[28] K. J. Astrom and L. Rundqwist. "Integrator Wind-up and How to Avoid It". In: () (cit. on p. 23).

[29] M. Gräber. "Practical PID tuning guide". In: (2021). URL: https://tlk-energy.de/blog-en/practical-pid-tuning-guide (cit. on pp. 24, 29, 30).

[30] *P, I, D, PI, PD, and PID control*. [Online; accessed 2022-02-17]. Dec. 2021 (cit. on p. 26).

[31] G. Ellis et al. *Control System Design Guide A Practical Guide*. URL: https://www.sciencedirect.com/book/9780123859204/control-system-design-guide (cit. on pp. 27, 29, 30).

[32] K. H. Ang, G. Chong, and Y. Li. "PID control system analysis, design, and technology". In: *IEEE Transactions on Control Systems Technology* 13 (4 July 2005), pp. 559–576. ISSN: 10636536 (cit. on p. 28).

[33] P. Woolf. *CHEMICAL PROCESS DYNAMICS AND CONTROLS*. URL: https://books.google.pt/books/about/Chemical_Process_Dynamics_and_Controls.html?id=Op87vwEACAAJ&redir_esc=y (cit. on pp. 30, 31).

[34] A. S. A. El-Hamid et al. "Comparison Study of Different Structures of PID Controllers". In: *Research Journal of Applied Sciences, Engineering and Technology* 11 (6 Oct. 2015), pp. 645–652. ISSN: 20407459. DOI: 10.19026/rjaset.11.2026 (cit. on p. 31).

[35] J. F. Smuts. *Process Control for Practitioners*. 2011. ISBN: 0983843813 (cit. on p. 31).

[36] J. F. Smuts. *PID Controller Algorithms*. URL: https://blog.opticontrols.com/archives/124 (cit. on p. 31).

[37] *Serial or parallel PID, which structure to pick?* URL: https://www.acsysteme.com/en/multimedia-resources/serial-or-parallel-pid/ (cit. on p. 32).

[38] Microsoft. URL: https://docs.microsoft.com/en-us/cpp/cpp/struct-cpp?view=msvc-170 (cit. on p. 55).

[39] ESPRESSIF. *Watchdogs*. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/wdts.html (cit. on p. 69).

[40] H. Blidh. URL: https://bleak.readthedocs.io/ (cit. on p. 74).

[41] Python. URL: https://docs.python.org/3/library/struct.html (cit. on p. 74).

[42] matplotlib. URL: https://matplotlib.org/stable/api/_as_gen/matplotlib.animation.FuncAnimation.html (cit. on p. 76).

[43] Y. Liu. *PyHRM - High Resolution Melt Analysis in Python*. URL: https://github.com/liuyigh/PyHRM/blob/master/PyHRM.ipynb (cit. on p. 79).

[44] S. Seabold and J. Perktold. "statsmodels: Econometric and statistical modeling with python". In: *9th Python in Science Conference*. 2010 (cit. on p. 79).

[45] NIST. *NIST/SEMATECH e-Handbook of Statistical Methods*. 2012. URL: http://www.itl.nist.gov/div898/handbook/pmd/section1/pmd144.htm (cit. on p. 79).

[46] Scipy. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.find_peaks.html (cit. on p. 80).

[47] MathWorks. URL: https://www.mathworks.com/help/signal/ug/prominence.html (cit. on p. 80).

[48] M. Reingart. URL: https://pyfpdf.readthedocs.io/en/latest/index.html (cit. on p. 83).

[49] RxJS. URL: https://rxjs.dev (cit. on p. 90).

I

Annex 1

I.1 Equations

$$
\begin{aligned}
& F = MA \\
& \Leftrightarrow u - B\dot{y} = M\ddot{y} \\
& \Leftrightarrow M\ddot{y} + B\dot{y} = u \\
& \Leftrightarrow M\mathscr{L}[\ddot{y}] + B\mathscr{L}[\dot{y}] = \mathscr{L}[u], \\
& \text{considering } \mathscr{L}[y(t)] \rightarrow Y(s) \text{ and } \mathscr{L}[U(s)] \rightarrow U(s) \\
& \Leftrightarrow M(s^2Y(s) - sy(0) - \dot{y}(0)) + B(sY(s) - y(0)) = U(s) \\
& \Leftrightarrow Ms^2Y(s) - Msy(0) - M\dot{y}(0) + BsY(s) - By(0) = U(s) \\
& \Leftrightarrow Y(s)(Ms^2 + Bs) = Msy(0) + M\dot{y}(0) + By(0) + U(s) \\
& \Leftrightarrow Y(s) = \frac{M(sy(0) + \dot{y}(0)) + By(0)}{Ms^2 + Bs} + \frac{U(s)}{Ms^2 + Bs}, \\
& \text{considering initial conditions are null} \\
& \Rightarrow \frac{Y(s)}{U(s)} = \frac{1}{Ms^2 + Bs}
\end{aligned}
\tag{I.1}
$$

$$
\begin{aligned}
& C = G(R - CH) \\
& \Leftrightarrow C = GR - CGH \\
& \Leftrightarrow C + CGH = GR \\
& \Leftrightarrow C(1 + GH) = GR \\
& \text{Since: } T = \frac{C}{R} \\
& \Rightarrow T = \frac{G}{1 + GH}
\end{aligned}
\tag{I.2}
$$

$$
\begin{aligned}
& C = G(R + CH) \\
& \Leftrightarrow C = GR + CGH \\
& \Leftrightarrow C - CGH = GR \\
& \Leftrightarrow C(1 - GH) = GR \\
& \text{Since: } T = \frac{C}{R} \\
& \Rightarrow T = \frac{G}{1 - GH}
\end{aligned}
\tag{I.3}
$$

$$
u = K_P e + K_I \int e dt + K_D \frac{de}{dt} \Leftrightarrow
$$

$$
u = K_P \left(e + \frac{1}{T_I} \int e dt + T_D \frac{de}{dt} \right) \Leftrightarrow
$$

Considering:

$$K_I = \frac{K_P}{T_I}$$

$$K_D = K_P T_D$$

Now considering the following equivalences: (I.4)

$$W = u$$

$$T_S - T_O = e, T_S = \text{set value and } T_O = \text{the output value}$$

$$P = K_P$$

$$I = \frac{1}{T_I}$$

$$D = T_D$$

$$W = P\left((T_S - T_O) + I \int (T_S - T_O) dt + D \frac{d}{dt} (T_S - T_O) \right)$$

I.2 Definitions

I.2.1 Definitions relative to the Feedback Control Block Diagram 2.8

- **Definition 1**: The plant (or process or controlled system) *g2* is the system, process, or object controlled by the feedback system.
- **Definition 2**: The controlled output *c* is the output variable of the plant, under the control of the feedback control system.
- **Definition 3**: The forward path is the transmission path from the summing point to the controlled output *c*
- **Definition 4**: The feedforward (control) elements *g1* are the components of the forward path that generate control signal *u* or *m* to the plant.
- **Definition 5**: The control signal *u* (or manipulated variable *m*) is the output signal of the feedforward elements in *g1* applied as input to the plant *g2*.
- **Definition 6**: The feedback path is the transmission path from the controlled output *c* back to the summing point.
- **Definition 7**: The feedback elements *h* establish the functional relationship between the controlled output *c* and the primary feedback signal *b*.
- **Definition 8**: The reference input *r* is an external signal applied to the feedback control system, usually at the first summing point, in order to command a specified action of the plant. It usually represents ideal (or desired) plant output behavior.
- **Definition 9**: The primary feedback signal *b* is a function of the controlled output *c*, algebraically summed with the reference input *r* to obtain the actuating (error) signal *e*, that is, $r \pm b = e$.
- **Definition 10**: The actuating (or error) signal is the reference input signal *r* plus or minus the primary feedback signal *b*. The control action is generated by the actuating (error) signal in a feedback control system.
- **Definition 11**: Negative feedback means the summing point is a subtractor, that is, *e*=*r*-*b*. Positive feedback means the summing point is an adder, that is, *e*?*r*+*b*.