A Generic Sensor Framework for Parallel and Continuous Data Processing

Master Thesis
to obtain the academic degree of
Master of Science
in the Master’s Program
Business Informatics
STATUTORY DECLARATION

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.
This printed thesis is identical with the electronic version submitted.

Traun, 12.07.2018
Table of Contents

1. Chapter 1 – Introduction ........................................................................................................... 7
2. Chapter 2 – Current Situation and Related Work ................................................................. 8
   2.1. OSGi ............................................................................................................................. 9
   2.2. AsTeRICS .....................................................................................................................11
   2.3. Internet of Things (IoT) ...............................................................................................13
3. Chapter 3 – A Generic Sensor Framework ................................................................. 17
   3.1. Architecture and modules .............................................................................................17
      3.1.1. Recorder module ..................................................................................................19
      3.1.2. Remote-Control .................................................................................................22
      3.1.3. Core module ........................................................................................................23
      3.1.4. Driver module ......................................................................................................25
      3.1.5. sensorMetaWear module ......................................................................................30
      3.1.6. sensorCortrium module .......................................................................................31
   3.2. Hardware devices ............................................................................................................35
      3.2.1. CortriumC3 ...........................................................................................................35
      3.2.2. MetaWearCPro ....................................................................................................36
   3.3. Configuration files ...........................................................................................................37
   3.4. Setup and data acquisition process ..................................................................................46
      3.4.1. App start-up ...........................................................................................................47
      3.4.2. Device initialization ...............................................................................................48
      3.4.3. Connection process ...............................................................................................56
      3.4.4. Data logging ..........................................................................................................61
   3.5. Conclusion ......................................................................................................................64
4. Test Results ............................................................................................................................66
   4.1. Scenario .........................................................................................................................66
      4.1.1. Test procedure .......................................................................................................67
      4.1.2. Device configuration .............................................................................................67
   4.2. Results ............................................................................................................................68
      4.2.1. CortriumC3 body temperature ...............................................................................68
      4.2.2. CortriumC3 device temperature .............................................................................69
      4.2.3. CortriumC3 respiratory rate ..................................................................................70
      4.2.4. CortriumC3 accelerometer data .............................................................................71
      4.2.5. MetaWearCPro accelerometer data ......................................................................72
      4.2.6. MetaWearCPro barometer data .............................................................................77
4.2.7. MetaWearCPro gyroscope data ............................................................................. 78
4.2.8. Combined analysis ................................................................................................. 80
5. Chapter 5 – Discussion, Conclusion, and Future Work ............................................. 85
   5.1. Discussion and Conclusion ...................................................................................... 85
   5.2. Future Work ............................................................................................................. 86
Abstract

Being able to live self-reliantly and independently also with advancing age is one of the basic human desires, especially at an older age because in this case it is oftentimes a matter of pride. How to technically contribute to this aspect is the major motivational factor for this thesis which merely uses the SENEX project as its thought-provoking impulse.

A well-thought-out design and implementation of a generic sensor framework for parallel and continuous data processing are what is needed to construct some sort of body surveillance system which monitors and records crucial vital signs and other sensor data in an unobtrusive way. This way, valuable conclusions can be drawn which, in the best case, lead to precious insight and intelligent inferences to better understand the human body.

The focus of this framework lies upon its generic nature, simple configuration, and uncomplex extensibility. In this thesis, related work aspects, the framework itself including detailed architecture and implementation facets as well as process flow analyses, a personal study, and critical concluding comments are presented. It does neither encompass nor disclose any confidential SENEX characteristics or SENEX-related test results. However, it gives a deep understanding of the framework, the way it functions, and what can be achieved with it. In my case, I show the advantage of connecting several distinct sensors to determine a specific motion sequence over only having the data of one single sensor.
Acknowledgement

At first, I would like to thank my supervisor and my university for the opportunity to be part of the very intriguing SENEX project and contribute to a worthy cause.

Further, I want to thank my family, friends, and girlfriend for being there for me and supporting me throughout the entire process.
Chapter 1 – Introduction

The purpose of my thesis is to show the implementation and usage of a generic sensor framework for parallel data recording implemented in Java for the Android platform. This framework was developed by a workgroup at Johannes Kepler University in Linz of which I was part to contribute my share. In this thesis, I put a non-exclusive focus on the sensor devices which I implemented but I do not own any of the code.

The creation of this framework was part of a project called SENEX [1] whose purpose is to find out whether it is possible and technically viable to detect and identify moments or states of confusion related to initial stages of dementia. The latter comprise people who are still able to manage their everyday life without external assistance but are prone to short-term memory losses and other symptoms of incipient dementia. Although they can still live self-dependently, these short moments of confusion can entail dangerous situations in the outside world, e.g., when going shopping and forgetting about it on the way which leaves the respective person confused about their location and potentially frightened even if it is usually merely a temporary condition.

The goal is to enable people affected by dementia to participate in public life for as long as possible with technical aid but without requiring the daily support of another person. As such a support system cannot be solely based on intra-body signs or indicators partly due to technical limitations and partly because, for example, brain waves are not helpful to determine moments of confusion, the SENEX system concentrates on kinetic and visual indicators. Relevant technical appliances include various body-attached sensors and eye-tracking cameras. Since an affected individual is supposed to rely on our guidance system based on the presented framework, it must work independently as well as react automatically to critical situations without requiring any user interactions. The type of assistance such a system can provide is not part of the SENEX project, as the focus lies on the feasibility and practicability of detecting said situations. However, the SENEX project does include a clinical study with authentic test subjects. This study is not a component of my thesis.

The respective SENEX parts form the key question for my thesis which is whether and how it is feasible to retrieve, collect, and process continuous data package updates from various sensor devices and how to design and implement a generic framework which allows to easily configure and add numerous sensor devices of diverse types. These devices can but do not have to be attached to a human being. In general, the processing part can include various methods and analysis steps and is certainly not limited to any procedures or approaches performed to interpret my personal test case results. An aspect on which I put a major focus in this thesis is the generic nature of our framework which allows to connect different sensor devices with minimal effort. The related work section deals with selected and already existing products, solutions, and approaches. Its purpose is to state the underlying problem and introduce a common ground which allows for a comparison to our own approach and implemented solution in order to point out differences, similarities, as well as advantages, disadvantages, and possible improvements. The latter topics are also covered in the last chapter of this thesis. Furthermore, and to round off my thesis, I conducted a personal study to test the created framework and a selected range of sensor devices including a detailed analysis of the results.
2. Chapter 2 – Current Situation and Related Work

The first main chapter of my thesis deals with already existing approaches, products, and solutions related to our framework. It is not my intention to give a complete overview of related products, list all available approaches, or similar. The purpose of this chapter is to show a selected range of existing solutions, products, or at least approaches to solve the problem of communicating with various sensor devices and processing their output. Before I go on with this, I explain and describe the initial situation including the problem we want to solve, as it is an important connection to this related work part. However, our personal solution to this problem is presented in the next chapter. Additionally, I want to note that the linked devices do not necessarily have to be sensor devices but can also be cameras, keyboards, printers, etc. The reason sensor devices have been chosen at this point already is to make this section better comparable to our framework described below. The opposite is true for our framework, as it could communicate with other devices than sensor devices as well but was not created for that purpose which is why I focus on the latter.

Figure 1 above includes two diverse sensor devices, A and B, which require two separate driver libraries, driver A and B, which are obviously not the same and do not have anything in common regarding reusability. Additionally, each of them relies on some sort of configuration with the same characteristics, i.e., not the same and not reusable. Both sensor devices send data packages which need to be processed to retrieve interpretable output of some sort which does not require further specification at this point. The following problems arise:

- How can we communicate with the different sensor devices without writing a separate application for each device especially because they rely on different driver libraries?
- How can we configure those sensor devices in one place?
- How can we collect and process the data packages from all devices in one place to retrieve a combined and interpretable output?

The solution to this problem is a central device communication and data processing unit which addresses all the core questions mentioned above. A corresponding visual and many more details are presented in the next chapter.
2.1. OSGi

This first sub-section deals with OSGi or more specifically with the OSGi framework which is a well-known approach in the field of Java programming in connection with Internet of Things (IoT) and associated topics, such as home automation, healthcare, and automotive development. I want to emphasize once more that this is neither supposed to be a complete overview or description of the entire OSGi principle nor a universal solution to all problems in this field for all technologies. Therefore, I only give a general introduction and pick out interesting details which are beneficial to this thesis and which support the understanding of the underlying issue. All external information in the first part of this sub-section is taken from the official OSGi website [2] and the official PDF description of its most recent release – 7 [3]. OSGi stands for Open Services Gateway initiative which was the original name of the current OSGi Alliance non-profit organization. “OSGi provides a vendor-independent, standards-based approach to modularizing Java software applications and infrastructure” [4] for local and network usage. OSGi including its framework puts a heavy focus on modularity, components, and reusability to reduce software complexity, maintenance efforts, errors, and overall costs. Furthermore, this increases agility, adaptability, and the longevity of the entire system. It also supports the development of extensive projects and enhances testing efficiency. Below, the layered architecture model is shown in Figure 2. It gives a first insight into how OSGi is structured. Further details are provided below.

![Figure 2 - OSGi layers (taken from [5])](image)

The Java VM is a crucial part and runs on top of the native operating system which means that this framework can be used on all platforms which are able to run Java. The execution environment limits the methods and classes available based on the platform. The modules layer specifies how code can be exported and imported by a bundle. The life-cycle layer is responsible for installing, starting, stopping, updating, and uninstalling bundles at the module layer. In order to connect bundles, which are OSGi components created by developers, the services layer is required. Security aspects are considered on all OSGi-specific layers including the execution environment. To sum it up, components are packaged in bundles, which are JAR files, and communicate through services which are shared Java objects. Bundles are administrated by life-cycles and handled within modules which means that other bundles must explicitly import the required code parts designed for exporting through services, as sharing entire bundles is neither
possible nor intended by the framework. The following image in Figure 3 provides additional clarification.

![Figure 3 - OSGi layer interaction (taken from [3])]

It shows how bundles register, unregister, as well as get, unget services to communicate with other bundles, how services and bundles are managed by life-cycles and how the latter ones work with modules, how bundles load classes, i.e., import code from modules containing other bundles, and how they get executed in their respective execution environment. Based on the descriptions so far, in an even more abstract way, and to make it comparable to our underlying problem, the following conclusions can be drawn visualized in Figure 4. The module and life-cycle layers are fundamental principles on which the higher-level components are based. Thus, they are also illustrated that way in said figure below.

![Figure 4 – Abstract OSGi framework]

At this point, I need to state a crucial difference between the OSGi framework and our own, as the first one is designed to support inter-component communication while ours is not meant to be used to establish a communication between sensor device A and sensor device B, as all sensor devices function independently from each other. Nevertheless, and with appropriate modifications to the framework, it would be possible to enable inter-sensor device communication as well, but it is not part of the SENEX specification.

This means that in order to compare both frameworks, bundles do not only represent sensor devices but also our control unit which is further specified in the next chapter. The service
The abstract elements device class A and B represent OSGi bundles which can instantiate an AbstractDevice class – only in theory, of course, as an abstract class cannot be used this way, but it hides the more sophisticated device class concept which is explained later – of their own type which resembles an OSGi object. DeviceProxy knows all about its respective device. Note that each device has its own DeviceProxy, i.e., OSGi service, which has been simplified here. DeviceController is aware of all its DeviceProxy instances and reacts according to the abstract control unit’s commands. This delineates on a general level that the two approaches resemble each other, can be linked, and follow the same underlying principle. However, they certainly differ regarding security, modularity, and development status.

2.2. AsTeRICS

The information in this section is a combination of the official AsTeRICS website [6] and its official PDF quick start guide [7] as well as a personally conducted interview with a Johannes Kepler University employee who is involved in its creation and continuous development process. AsTeRICS stands for Assistive Technology Rapid Integration and Construction Set and is
specifically designed for assistive technologies in various fields which range from game control over input device emulation to embedded devices targeted towards users with disabilities. The AsTeRICS project is open-source and developed by several partners located in the European Union and funded by the European Commission. Furthermore, the project coordinator is situated at Johannes Kepler University in Linz, Austria.

A set of currently 290 plugins forms the heart of the AsTeRICS concept which includes sensors, data processors, and actuators. Consequently, software applications, such as Skype, Gmail, or media players, as well as hardware devices, e.g., mice, keyboards, joysticks, printers, webcams, RFID-readers, etc., are supported and integrated. Since the concept is based on the OSGi framework, modularity realized through plugins is a key aspect. In addition, AsTeRICS Configuration Suite (ACS) is provided which represents a graphical editor relying on AsTeRICS Runtime Environment (ARE) to enable a broader user audience to use their application based on drag-and-drop activities without the need of programming knowledge. However, if specific modifications or requirements exist, it is possible to add said functionality through self-programmed plugins which is a guided process supported by a plugin wizard. Furthermore, a complete GUI editor is included as well which, for example, lets the user build their own smart home application via drag and drop and connect it with selected plugins to control security cameras, blinds, power sockets, or even a smartphone. In general, it is platform-independent but as found out in the interview, it is not designed to and does therefore not run on Android or iOS smartphones but would work on Windows phones. This is one of several vital reasons why it was not chosen to be used as part of the SENEX project. Additional reasons include a very limited Bluetooth support, complicated, very tedious, and probably impossible connections of proprietary hardware devices which do not support certain standard protocols or IP-based communication. Finally, the open-source nature might have been another hindering aspect for the SENEX board. Nonetheless, a diagram including an explanation of a potential use case is shown below in Figure 6.

It is a simple scenario including sensors, processors, and an actuator which shows AsTeRICS' basic concept outlined above in action. At first, the mouse position on the screen is determined by the horizontal and vertical nose coordinates. Second, the x and y values of the eye tracker output are processed using a mathematical evaluator which generates a single output value representing the user's eye focus on the screen. If this value reaches a specified threshold, a left mouse click is triggered at the current mouse position determined by the nose. The graphical editor includes a property configurator to define all necessary values, events, and actions, which are all not part of this diagram.

Overall, AsTeRICS provides a simple and attractive way to build applications in conjunction with assistive technologies. It is facile to use, well-documented, and has a broad range of application areas. Everything can be designed, configured, and run from within the AsTeRICS suite. However, it is not the right tool for our purpose for the above-stated reasons.
2.3. Internet of Things (IoT)

Internet of Things is a very broad topic which could easily fill several theses. Thus, the purpose of this sub-section is neither to give a complete overview of what IoT comprises and what not nor to explain its history and advancement, but to give a general introduction and definition in order to find a connection to our developed solution – or to conclude that there is none – backed by examples. Smart homes, for instance, are an aspect which is usually heard in conjunction with IoT. However, it is merely one of many and not relevant here.

To find common ground, a definition of IoT is required and shall be given at this point. It is based on these three papers [8], [9], [10], which do not assume a significant amount of prior knowledge. Crucial elements are its mostly wireless nature, a pervasive and connected presence, autonomous and smart behavior, as well as interaction and cooperation within the system and between all participants. Human interaction can be part of it but is neither mandatory nor necessarily intended. All sources agree that IoT is one of the currently dominating topics with a vast impact on everyday life especially considering the near future. The following components below, which represent merged content of paper [9] and [10], constitute the core of IoT.

Additionally, it is important that all relevant devices can be uniquely identified, which usually happens via IP and URN (Uniform Resource Name).

- RFID (Radio Frequency Identification)
- WSN (Wireless Sensor Networks)
- Middleware
- Cloud computing for data storage, analytics, and more
- Application software and visualization
RFID technology is commonly utilized in RFID tags which do not require any battery, can be read by RFID readers, and can store more data than conventional barcodes. However, there are also active RFID tags which provide their own battery supply to initiate communication with a reading device. Mixed products are available too.

WSN consists of several sub-components which encompass hardware, middleware, secure data aggregation, and a communication stack. WSN can further function as alternative to RFID tags but both can coexist and cooperate if desired. A wireless sensor network typically consists of several nodes, i.e., sensor-equipped devices. Each of them comprises sensor interfaces, a power supply, and processing as well as transceiver units. The communication stack enables the nodes to talk to each other, to the central hub or base station, and connects the network to the Internet. The middleware guarantees platform independence. Secure data aggregation is essential to avoid a system failure due to one or more node failures and to prevent intrusions.

IoT applications and services can become very complex. To reduce this complexity, additional middleware can and should be used to gain a higher level of abstraction and hide unnecessary details. An example for an open-source middleware platform for sensors is Global Sensor Networks (GSN) which facilitates the development of sensor services.

Cloud computing is naturally important for all kinds of data storing, processing, analytics and analysis, as well as smart follow-up actions carried out by other systems, applications, or similar. As these kinds of resources are usually not available locally, cloud service providers step in. The data values to process come from sensors, tags, etc., and can be sent as well as received via gateways or other application components depending on the concrete implementation.

Application software is essential, as it enables human-to-device and device-to-device communication and interaction. These applications usually run on separate devices which communicate via middleware with, for instance, wireless sensor networks, push data to the cloud, as well as receive results and act accordingly. Note that there is nothing like the one IoT example which covers all potential system designs. Below, an abstract visualization of the IoT concept is given in Figure 7. It is followed by another diagram displaying an abstract IoT scenario used to show and explain why our solution is not an IoT system and what would be needed to make our framework part of one.
Figure 7 shows four of the five core components of IoT, as the middleware components act as intermediaries between the other ones. Note that there are no specific applications but rather application fields shown at the top. The same idea applies to the WSN part of the image.

As alluded above, Figure 8 shows an abstract IoT scenario in which the user is working on a desktop computer with a specific application software installed. This application communicates with a cloud service which provides a database and a processing engine. The internet icon in the cloud places emphasis on the online IoT nature.

On the other end, a wireless sensor network in conjunction with several RFID tags is in place. These sensors and tags provide data to the cloud via one or more gateways. Sensors and tags can communicate with each other to exchange necessary information. Additionally, the user can
utilize the application to control said sensors and tags via the cloud to, for example, change configurational values or ask for additional information. The data provided by the RFID/WSN system undergoes some processing tasks in the cloud’s processing engine. The purpose of this procedure is to turn non-user-interpretable input into comprehensible output which is then further processed and visualized by the application on the client-side. The last missing aspect which has not been considered yet are the smart background tasks. The background tasks icon stands for all kinds of autonomous actions taken by the system which run in the background based on the data provided to the cloud. This is an essential aspect of IoT. Useful cloud data is sent to other data centers, applications, service providers, or similar. These third-party elements take intelligent actions based on their input. Some examples for such actions encompass the provision of hardware elements, spare parts, or additional cloud storage, the scheduling of appointments, or calling assistive service hotlines. The potential options are countless.

What our framework is missing to become an IoT system are smart online components which handle data storage, data processing, take autonomous actions in the background, and function as links between the application, i.e., our recording device, and the sensors. Moreover, our sensors are not meant to communicate with each other which is not an IoT requirement but what IoT was, amongst others, designed for. In order to become part of the IoT world, our framework would need fundamental restructuring, redesigning, and reprogramming. The sensor devices would have to communicate with a cloud service which, itself, communicates with our smartphone application. In fact, our smartphone application would not be responsible for recording anymore but for visualizing and handling the processed input data coming from the cloud. Therefore, the cloud service would have to process its own input data coming from the gateways first and take according actions in the background plus send meaningful information back to the smartphone. For our purpose, such actions could be automated emergency calls or notifications to external facilities or a family member of the affected person. Such an action could also be to alarm a guidance system attached to the affected person without any human interaction required. If I take this one step further, the user’s smartphone could be replaced by said guidance system or even left out completely if no human interaction is required. As a matter of fact, these are valid options for future development discussed in more detail at the end of this thesis.
3. Chapter 3 – A Generic Sensor Framework

The purpose of this section is to analyze our implementation of a generic sensor network used for parallel and continuous data processing. Before I go into detail as to how we realized this solution, I briefly address the most relevant technologies involved in the creation process. The initial situation including the problem we want to solve has already been delineated in the previous chapter and is not repeated here. After this introductory segment, the framework’s modules and components including an abstract solution for said problem is presented and explained in depth.

The technologies, software, hardware, and platforms used to realize our project comprise various sensor devices by different manufacturers, Bluetooth, Android and Java, Subversion as version control system in connection with TortoiseSVN as client, Eclipse and IntelliJ as IDEs, and smartphones of diverse brands for testing.

3.1. Architecture and modules

Before I begin to describe the actual realization and implementation of the framework in detail, I talk about the previously alluded abstract visualization of our solution. Below, said drawing is shown in Figure 9 which contains some entities of the abstract problem in Figure 1 in the previous chapter but has been significantly extended. First, I further specified the sensor devices, as each of them encompasses at least one sensor which usually sends regular data updates, e.g., an ECG sensor device which receives heart rate values from one of its integrated sensors. Second, I introduced a Recorder entity which contains a control unit which is responsible for the configuration and control of all sensor devices including their integrated sensors. Additionally, it receives their data updates and provides interpretable output as needed. Third and most importantly, I added a generic interface which can communicate with various sensor device types and understands several types of sensor device configurations. This interface is part of the control unit but the latter one does not need to know how to communicate with the sensor devices, as it simply calls the respective interface functions. Naturally, the generic interface needs to have access to all required driver libraries because there is no generic way of sending the same commands to different sensor devices on an implementation level. The external configurational input which is provided by the control unit needs to be read, processed, and accordingly sent to the respective sensor device as well. Overall and with a glance to the following implementation details, this means that we need a separate configuration file for each sensor device as well as a driver class. This class uses the respective driver library and connects the sensor device with the generic interface. This way, the control unit can communicate with all sensor device types using the same commands. This also enables the usage of configuration files which all follow the same structure. Although each sensor device needs its own configuration file, once implemented, modifications can be performed using the main configuration file only. Configuration files are explained in detail later in this chapter. In the diagram (Figure 10) which follows the one discussed at this point, the Recorder entity is split into SENEX-Recorder, core, and driver.
As stated above, the created framework consists of three fundamental modules and one additional module for each sensor device. The three modules are called: SENEX-Recorder, core, and driver. The identifier for each sensor device module uses the product’s name including the word “sensor” as a prefix. The two relevant sensor modules for this thesis are: sensorCortrium and sensorMetaWear. Both device types together with the functionality they provide are described in the subsequent sections as well.

Figure 10 shows the dependencies between the mentioned modules. However, this is a mere overview illustration which becomes more detailed and explicit below. Moreover, it does not represent the exact code dependencies but rather the logical connections between the different parts of the project. The following sections use this diagram – referenced as core overview diagram – as a starting point and, at first, briefly describe each module separately followed by an extensive combined analysis with the help of several flow charts which guide through the entire application process. The goal is to show how these modules including their content interact with each other in order to gain a complete understanding of the presented framework’s features and functionality.
Senex-Recorder represents the actual Android application from which the whole recording process including the setup of the sensor device connections can be managed. The dotted and almost transparent connection between this module and driver in the core overview diagram is owed to the fact that Senex-Recorder merely requires some helper classes and interfaces. The entire driver module, however, is primarily essential for the core module and its relation to the sensor devices. In hindsight, the driver code parts which are required by Senex-Recorder as well as core could have been moved to the latter right away. Such improvements as well as limitations of the entire application is dealt with in the final chapter. Naturally, Senex-Recorder depends on the core module which requires sensor devices to work with and of course the driver to know how to talk to them. Since driver does not – with a few minor exceptions – contain any concrete implementation of functionality, the connection is also visualized using a dotted but as it is crucial to core not transparent line.

3.1.1. Recorder module

The Senex-Recorder module comprises several classes and enums. To explain the CommandTypes enum, I need to mention that a second Android application was created. It enables another person to remote-control the main application on the main device carried by the test person, i.e., the one with the device sensors affixed to their body. This was done because it would be hard to operate the main mobile phone directly, as it is supposed to be attached to the test user for reasons of connectivity and stability. The communication between both mobile phones is realized using a private wireless local area network and said CommandTypes enum simply lists all available commands which can be propagated to and are accepted by the main application. However, the remote-control application is not part of this thesis and is not important for any of the upcoming sections, as it neither influences the behavior nor the functioning of the overall process but can rather be seen as an additional feature. Nonetheless, I include a screenshot and short description in the next sub-section to present a brief outline.

The MessageTypes enum goes hand in hand with the command types and is excluded for the same reason. Naturally, the implementation of the communication between both application devices stays disregarded as well.

The FileRecorderService class is not only crucial for the logging part but for the entire process and implemented as an Android ServiceConnection to ensure it is always up and running. The class is responsible for the correct log file setup, logging itself, i.e., writing to output files, several CPU and wireless LAN settings. It is also partially accountable for the device state management. Additionally, the connection and disconnection to and from the sensor devices run via FileRecorderService because it needs to take care of the affected logging process. The same is true for the calibration as well as the start, pause, resume, and stop recording processes.
Nevertheless, the actual interaction with the sensor devices is handled in the controller which is part of the core module and is described later. Figure 11 visualizes the intra-module dependencies for the Senex-Recorder module. This is especially important for the subsequent bigger picture.

Before we get to the main application file, SenexActivity, I want to briefly outline the DeviceButton class. The UI of the application visualizes the device setup using a button for each sensor device explicitly defined in the configuration files. The latter ones are explained in detail later. Each button provides functionality to interact with the respective sensor device. Moreover, the button also shows the sensor device’s current state as well as battery status. Each action and status are additionally supported with the use of colors and animations. Figure 12 shows the SENEX recorder application which runs on the recording Android device itself. Some parts of it are self-explanatory, the rest comprises: the pink battery bar positioned below its corresponding device name, the test subject ID at the top, and a menu for further options, including remote control communication and sensor device configuration, in the upper right corner. The connect/disconnect button connects/disconnects all sensor devices at once. If required, then each device can be separately connected by tapping the respective yellow device button. The start button is only available if all sensor devices have been successfully connected and turns into a pause button thereafter. Accordingly, the stop button can only be triggered once the recording has been started.

The already alluded main Android application lives inside the SenexActivity class. It represents the central controller for the entire setup and is used to manage all sensor devices. It provides the following functionality:

a) FileRecorderService initialization and management
b) Android permission management
c) Storage directory initialization
d) Loading of the main configuration file
e) Setup of the root profile as well as the SENEX profile
f) Main log file initialization
g) Generating the sensor device buttons on the screen of the Android device according to the sensor devices configured in the main configuration file
h) Ensuring Bluetooth availability and functionality
i) Handling the button interaction to connect to, disconnect from, start, pause, resume, stop, and calibrate each sensor device
j) UI updates
Note that the calibration process will not be explained any further because it simply defines basic parameters for each sensor device and/or does a short recording to gain reference data prior to the actual recording. It might also switch on a LED or similar depending on the respective sensor device. A successful calibration process is fundamental to start the recording process, but it does not contribute to the description and understanding of the presented framework, thus, it is omitted.

Everything regarding the points a) to j) which requires additional explanation is explained in the following sections, as this part is only intended to succinctly outline all modules.

Finally, I briefly want to address the AndroidManifest.xml file. Besides several permissions which are required to run the application and other common settings, it is especially important to register the Service classes. Android Service provides several benefits such as background and long-running tasks. It is possible to bind to it, e.g., from the main Android context, which establishes a close relationship between the respective Service and the actual application context. With the right configuration, this ensures that the Service instance will be available as long as the application needs it. Additionally, it can also survive the shutdown of an application. This is commonly used for messenger services, push notifications, etc. To create a Service component, the respective class merely needs to extend the Service class and implement its onBind(...) method to allow another class to bind to it. Optionally, <intent-filter> elements can be added to the Service declaration in the manifest file which allows certain events to trigger, i.e., run a Service instance independently from the rest of the application. A WLAN state change is an example for such an event. However, these filter options are not required for our application.
The reason it is necessary to include Service classes in Android’s manifest file is that otherwise the system cannot see nor run them. For my personal test setup, only the FileRecorderService and MetaWearBleService – provided by MbientLab’s own library – need to be declared.

3.1.2. Remote-Control

As already alluded, this sub-section is a mere overview of the SENEX-remote-control application which is used to run on another Android device to remotely control the SENEX-Recorder application on the recording device. Its purpose is not only to control the other application but also to calibrate the sensors, log incidents, and track a pre-defined path of events, i.e., a specific route through a building or similar, which the test-subject is supposed to follow. Figure 13 shows the corresponding UI. If the button saying “Ausgangsposition”, i.e., initial or starting position, is pressed, it means that the subject is ready and in position. Additionally, this event is logged and the next one will appear on the button. This procedure is useful to track and combine the recorded data with the test subject’s actions in retrospect. The upper right menu serves procedural settings as well as server and client connections. The server is another external device only used to provide different types of configuration files. Its status can be checked at the bottom. It is not a required architectural element as these files can also be stored directly on the device. Yet, it offers valuable possibilities for further enhancements and additional features. The client indicator represents the connection between the remote control and the recorder. The pause/start button triggers the respective button on the recording device. The stop button appears after the configured procedure has been completed. The yellow device indicator lights show that all devices have been connected but not yet calibrated. After the calibration process, they turn green and start flashing slowly while recording. The button with the question mark on it is designed to be tapped by the test subject’s guide in moments of confusion, i.e., when the patient could potentially suffer from memory loss or a similar state of confusion connected to their dementia.
3.1.3. Core module

The core module encompasses three classes and five interfaces of which three are only used for the three classes, one each. For reasons of simplicity and coherence, Figure 14 explains the intra-module dependencies.

In the diagram, only the most important relationships – because non-inter-interface ones – are shown. Note that the uses relationship either indicates a direct class property and/or field and/or interface parameter relationship which means it merely denotes that a connection between the two respective files exists. Inter-interface relationships are covered implicitly through the other
dependencies and are, therefore, not shown in the diagram. The following paragraphs describe each file including the class-interface dependencies.

The DeviceController class which acts as device hub implements the IController interface. It is implemented as Singleton and keeps a list of all DeviceProxy instances to be able to control and manage them. The IController interface dictates to implement all methods required to interact with DeviceProxy instances. With these methods, DeviceController can initialize itself as well as initialize, connect to, disconnect from, start, pause, resume, and stop all devices (DeviceProxy instances). Additionally, checks can be performed to know whether a device has already been initialized and whether it is ready. A getter to retrieve all DeviceProxy instances exists as well.

DeviceProxy can be regarded as an intermediary between DeviceController and a specific device, e.g., a CortriumC3 device. DeviceProxy always represents exactly one device but DeviceController can hold several DeviceProxy instances. Moreover, it provides additional features such as initialization tasks and event handling through listeners and event queues. It also knows its device’s sensors and configuration. The IDeviceProxy interface provides the same methods as listed for IDeviceController above plus additional getters, a reset device interface method, and naturally also the interface methods for the registration and deregistration of the device listeners.

Besides this, DeviceProxy class provides two static nested classes which represent a DeviceStateChangedEvent event and a SensorValuesChangedEvent event. They occur in two methods which are inherited from two interfaces from the driver module. Their exact usage is explained later but their existence is important because they are used for the event queues to work off device and sensor events. Both types of events are routed via DeviceProxy. Device events, on the one hand, are directly propagated to the respective listeners. A potential listener can either be DeviceButton to update the UI or FileRecorderService to appropriately react to a specific event, e.g., a disconnection or another error. Sensor events, on the other hand, are first propagated to the respective SensorProxy which then calls its own listeners to continue further. The code which takes care of these listener registrations is located in FileRecorderService which registers its static nested class DataToFileWriter at the respective SensorProxy. This means that there is exactly one writer, i.e., one log file, for each sensor. DataToFileWriter implements the ISensorListener interface which explains the uses relationship between this interface and SensorProxy. All this is explained in detail later in this thesis.

Consequently, SensorProxy acts as an intermediary between the respective DeviceProxy and one of its sensors, e.g., a heart rate sensor of the CortriumC3 device. SensorProxy always represents exactly one device sensor but DeviceProxy can hold several SensorProxy instances. The ISensorProxy interface provides several getter methods and naturally also the interface methods for the registration and deregistration of the sensor listeners. SensorProxy also implements a method for handling previously mentioned propagated SensorValuesChangedEvent events from DeviceProxy. The rest about SensorProxy has already been explained above.
3.1.4. Driver module

The driver module comprises four classes, one abstract class, one enum, and four interfaces. To begin with, Figure 15 visualizes the intra-module dependencies which will be outlined and briefly explained in the subsequent paragraphs.

For the sake of dependency completeness, note the following:

- The IDevice interface uses the Profile class, but this relationship is implicitly covered through the abstract class AbstractDevice.
- The ISensor interface uses the ConfigOption class, but this relationship is implicitly covered through the BasicSensor class.
- The IDevice interface uses the ConfigOption class, but this relationship is implicitly covered through the abstract class AbstractDevice.
- The abstract class AbstractDevice uses the ISensor interface, but this relationship is implicitly covered through the BasicSensor class.
- The IDevice interface uses the ISensor interface, but this relationship is implicitly covered because the abstract class AbstractDevice implements the IDevice interface and uses the BasicSensor class which implements the ISensor interface.
- The ISensor interface uses the IDevice interface, but this relationship is implicitly covered because the BasicSensor class implements the ISensor interface and uses the abstract class AbstractDevice which implements the IDevice interface.

![Figure 15 – Driver intra-module dependencies](image)

ISensorValueChangedListener and IDeviceStateChangedListener are the two interfaces mentioned before which are implemented by the DeviceProxy class of the core module to handle the two types of events called DeviceStateChangedEvent and SensorValuesChangedEvent. DeviceProxy is also the only class which implements these interfaces. We already know that the class to which DeviceProxy reports back is FileRecorderService. To retrieve the data from a sensor device, the DeviceProxy registers itself as listener at AbstractDevice because each concrete device implementation, e.g., CortriumC3 or MetaWearCPro, extends the abstract class AbstractDevice. This means that IDevice, AbstractDevice, and the actual device implementation
class combined represent a specific device. In other words, a sensor device uses its abstract superclass to report back to its very own DeviceProxy which then notifies FileRecorderService.

AbstractDevice implements the IDevice interface which provides: several getters, String constants, device as well as sensor listener registration and deregistration methods, a setter method to set the device configuration for a specific device, and logically all basic methods to connect to, disconnect from, start, pause, resume, stop, and reset a device. Additionally, a check can be performed to know whether a device is ready which is called by the respective DeviceProxy instance. The IDevice interface also comprises an enum called DeviceState which lists all available device states. Here are all valid ones excluding the ones related to the calibration process: \textit{UNKNOWN, DISCONNECTED, CONNECTING, CONNECTED, STARTRECORDING, RECORDING, RESUMING, PAUSING, PAUSED, STOPPING, STOPPED, DISCONNECTING, RESETTING, ERROR}. Furthermore, the static nested class DeviceProperty is also part of the IDevice interface. DeviceProperty is always made up of ConfigOption and an according value. The ConfigOption class is explained below. DeviceProperty instances are part of those methods which inform their listeners about device specific events and used in the concrete device implementations only. The usage enfold battery and device state updates.

Said methods to inform the listeners are already implemented in the abstract class AbstractDevice and called from the specific device implementation class using the \textit{super} keyword. Sensor-related events have their own inform-methods and utility classes described later in the paragraphs about the BasicSensor class and ISensor interface.

The AbstractDevice constructor accepts an instance of the Profile class which represents (a part of) a separate configuration file which is specific for each device. Then, it reads the device’s name, description, manufacturer, and version from the Profile. Afterwards, it creates a separate sub Profile for each device option, such as \textit{btaddr} (Bluetooth address), \textit{device_state}, or \textit{battery}, with the help of various utility methods which are part of Profile. The exact definition of the Profile class and how it works can be found below. However, the returned sub Profile instances for the key \textit{options} would be:

- options.btaddr.
- options.device_state.
- options.battery.

Notice the `.` Operator at the end of the key which signals that there is at least one more hierarchy level to come. If we investigate the Bluetooth address further, this information can be retrieved:

- options.bt addr . propName=BT-Addr.
- options.bt addr . propDescription=Bluetooth address of the device
- options.bt addr . propType=TEXT

Subsequently, the constructor goes through each of these Profile instances and puts each key, e.g., \textit{btaddr}, together with a newly created ConfigOption into a HashMap\langle String, ConfigOption\rangle for easy access.

Afterwards, another HashMap to comprise all available sensors of a specific device is created. This time, however, the value is an instance of ISensor, i.e., a BasicSensor. The respective sub Profile instances are retrieved via the key \textit{sensors}. Depending on the selected sensors, this can result in the following for a CortriumC3 device:
sensors.ecg1. (electrocardiogram sensor one)
sensors.ecg2. (electrocardiogram sensor two)
sensors.ecg3. (electrocardiogram sensor three)
sensors.resp. (respiratory sensor)
sensors.bodyTemp. (body temperature sensor)
sensors.deviceTemp. (device temperature sensor)
sensors.accel. (accelerometer sensor)

So instead of ConfigOption, a BasicSensor instance is created and put into the HashMap together with the according key, e.g., ecg1. This way, DeviceProxy can effortlessly check for available sensors and retrieve their Profile on demand.

As previously alluded, an instance of the Profile class represents a set of configuration properties and also enables a hierarchical structure. Additionally, sub Profile instances can be retrieved which are again instances of the Profile class and have access to the same properties. This implementation does not allow to get only specific parts of the entire configuration file, e.g., all the properties for one specific sensor of a device. Instead, it merely adapts and updates the respective qualifiers because the Profile class is a system of qualifiers. All Profile instances have access to the same data, but the access differs. Instead of writing sensors.ecg1.sensorName to retrieve the sensor's name, I can simply access it via the appropriate sub Profile instance using the key sensorName. Together with the HashMap<String, ConfigOption> instance in AbstractDevice, I can retrieve the ConfigOption for the Bluetooth address using the key btaddr. At that point, ConfigOption already loaded the required data from the previously passed Profile instance which can be accessed via various getter methods of ConfigOption.

Along with the configuration files, Profile entries are represented and stored using qualifiers which consist of key sequences separated by '.' characters. Therefore, no key can contain such a character. Plus, keys are case-sensitive.

Although I go into detail about configuration files in their own section below, here are a few exemplary configuration properties to deliver further insight and to show the nesting of hierarchy levels using the '.' operator:

```
deviceName=Cortrium C3
deviceDescription=BLE-Beacon with various sensors!
options=btaddr,device_state,battery
sensors=ecg1,ecg2,ecg3,resp,bodyTemp,deviceTemp,accel
sensors.ecg1.sensorName=ECG1
sensors.ecg1.sensorOptions.frequency.propValue=250
sensors.ecg1.sensorOptions.frequency.propType=INTEGER
```

The entire Profile class is based on the java.util.Properties class which is used to store the keys and values. The Java class already provides the basic functionality, but Profile makes the data access easier and enables hierarchical structuring through different qualifiers and various Profile instances, whereas the Java class can only implicitly reflect hierarchies using '.' characters in the respective keys. The Profile class does not require or even allow such characters as part of the key, since the Profile's qualifier, which represents the higher and more general hierarchy levels, knows about the structure. For every sub Profile, the incoming key gets appended to the existing qualifier creating another and lower hierarchy level.
The already addressed ConfigOption class requires an instance of a Profile to be constructed. Each ConfigOption keeps a reference to its underlying Profile and extracts the following information with the help of specific keys: propName for the name of the respective property (e.g., “Device Battery Level”), propDescription for a textual description of the given property (e.g., “The current level of the battery”), propType describing the property’s type of data representation (e.g., “INTEGER”), propValue for the property’s (initial) value of type propType, and propOptions comprising all possible and valid options for propValue (e.g., 0, 20, 40, 60, 80, 100).

Apart from its core functionality, the ConfigOption class offers standard getter and setter methods but also a method to retrieve sub settings by key. It returns a new HashMap<String, ConfigOption> containing ConfigOption instances which are hierarchically lower than the parent and directly below the provided key. For a MetaWearCPro device and its LED, this would mean the following: At first, the sensor options are retrieved by providing the key options to the root profile. This yields, amongst other, the following qualifier: options.led. Assuming this got stored in ConfigOption, we can now call getSubSettings(String key) to retrieve said HashMap containing ConfigOption instances whose underlying Profile instances are based on these qualifiers, respectively:

- options.led.settings.color.
- options.led.settings.riseTime.
- options.led.settings.pulseDuration.
- options.led.settings.repeatCount.
- options.led.settings.highTime.
- options.led.settings.highIntensity.
- options.led.settings.lowIntensity.

Naturally, the same results can be obtained without using any ConfigOption instances. In fact, the described method solely relies on its Profile instance reference and associated utility methods.

Lastly, ConfigOption also keeps a reference to a device’s OutputStream if applicable. However, this is only relevant for sensor devices which stream their data to our application. Such data is logged to .ts files. A small video camera attached to the participant’s head would be an example for this use case.

The BasicSensor class represents a sensor and is based on the ISensor interface. The latter one defines several constants, getter and setter methods, as well as two static nested classes, i.e., ValueDescription and SensorValue. ValueDescription is composed of a value name and a value type. The first one is a String value and the second one of type EDataType. EDataType is a separate class and also used to store propType in ConfigOption. It comprises all valid data formats, i.e., BYTE, SHORT, INTEGER, LONG, FLOAT, DOUBLE, TEXT, STREAM, UNKNOWN, and various helper methods used to convert between bytes and given data types. An exemplary ValueDescription instance could carry the following information: °C as value name and INTEGER as value type. Both can be easily retrieved via two separate getter methods. Naturally, these values are and have to be defined in the respective configuration file. These sensor data types are important for FileRecorderService to know which type of file to create for logging, e.g., .bin for binary input or .ts for data streams.

SensorValue, on the other hand, consists of a sensor ID and a corresponding value. The ID is of type byte and the value of type byte[][][]. Bytes are required because the kind of data which uses
SensorValue logs to above-mentioned binary files and it avoids retrospective conversion. Naturally, this can be handled in many different ways. Such SensorValue instances are used to convey the recorded data from the sensor device via SensorValuesChangedEvent events, which are worked off in the responsible DeviceProxy instance, to their respective DataToFileWriter which then logs the received data. At DeviceProxy, the SensorValue instances are split into ID and value again to retrieve the corresponding SensorProxy from the DeviceProxy instance’s SensorProxy Array and pass on the values. The SensorProxy Array keeps track of all available sensors and will be explained in further detail where feasible.

The BasicSensor class implements all its interface methods and among them, there is one which is particularly important, as it allows the general main configuration file, which defines all devices, Bluetooth addresses, and more. The device-specific configuration files always define all necessary properties but in case some of them need to be updated for a certain test scenario, this can be done by overriding device-specific configuration file settings (properties) with the ones taken from the main configuration file. To do so, the latter file simply needs to define the same properties and the rest is handled in the code. The exact implementation is described in the setup and data acquisition process section.

The constructor of the BasicSensor class, however, does the following: it keeps a reference to IDevice, i.e., AbstractDevice, i.e., the concrete device class, to which it belongs, a reference to its ID which is simply a counter variable passed in from AbstractDevice, and a reference to its own Profile also passed in from the corresponding AbstractDevice instance. Using instance, I naturally refer to the instance of the respective device class which extends AbstractDevice. With the keys sensorName and sensorDescription, BasicSensor retrieves its name and description, e.g., “Body Temperature” and “Body temperature in °C”, from the associated Profile instance. Subsequently, the key sensorOptions is used to obtain a list of sub Profile instances based on the respective BasicSensor, i.e., sensor. Below is an exemplary list of such resulting Profile qualifiers:

- sensors.bodyTemp.sensorOptions.frequency.
- sensors.gyro.sensorOptions.range.
- sensors.magnet.sensorOptions.power_preset.
- sensors.baro.sensorOptions.standbyTime.

Since these Profile instances are once again wrapped in ConfigOption instances, all properties provide a value for the same keys as already mentioned above: propName, propDescription, propValue, propOptions, propType. The ConfigOption instances are then, one by one, put into a HashMap<String, ConfigOption> together with their corresponding keys, e.g., frequency, range, power_preset, or standbyTime.

Next, all sensor value names are retrieved from Profile using the key valueNames. Depending on the specific sensor, this could return only one String value such as “temp” for the thermometer sensor or an Array containing several ones such as “x”, “y”, and “z” for the accelerometer sensor. Accordingly, the corresponding sensor value types are obtained using valueTypes as key. For the first example given, this would return “FLOAT”. For the second one, “FLOAT”, “FLOAT”, “FLOAT” would be returned as a String Array. Both, the value types and the value names, are then used to create a List of ValueDescription instances and since ValueDescription is composed of String and EDataType, the value name is
used as String and the value type is converted into EDataType. Ultimately, this List is used by the FileRecorderService to create proper log files as described above.

The only important utility method in the Helper class is the one to create a deep copy of an object. Why this is relevant is explained in the section about the setup and data acquisition process.

3.1.5. sensorMetaWear module

This sub section deals with the MetaWearCPro class. I intend to describe its main functionality, however, not in detail yet, because everything mentioned so far is brought up again and connected in the setup and data acquisition process section. Further information about both devices and what they are capable of can be found in the section about configuration files. In this case, there is no need to show a diagram to visualize the direct relationships between the MetaWearCPro class and all previously addressed classes, as it extends AbstractDevice which implements IDevice and no class requires to know at compile time that such a concrete device implementation even exists. This is due to the following lines in DeviceProxy's initialize method with KEY_DEVICE_DRIVER being "deviceDriver":

```java
String driverClazz = configuration.getString(KEY_DEVICE_DRIVER);
remoteDevice = (IDevice) Class.forName(driverClazz).newInstance();
```

At this point, driverClazz references the path to the respective device implementation class which it extracts from the main configuration file. For the MetaWearCPro device, the path looks like this:

```java
at.sew.senex.daq.sensors.mbientlab.MetaWearCPro
```

At runtime, the respective class can be instantiated and since it is treated as IDevice which is described above, there are no new dependencies to show in a diagram.

The class holds a reference to its Android Context because it is required for MbientLab’s proprietary library for the MetaWearCPro device, and Android’s BluetoothManager as well as ServiceConnection. The latter is implemented by the MetaWearCPro class to be able to interact with the sensor device and to define what happens on connection to and disconnection from the board with board simply referring to the main control unit of the sensor device. This term is introduced and used by MbientLab.

Since MetaWearCPro extends AbstractDevice which does not implement all IDevice interface methods, it has to take care of this. Therefore, it implements the following methods from IDevice: connect, start, pause, resume, stop, and reset. It also overrides or rather extends some methods from AbstractDevice to adapt them to its specific needs. These methods are: setDeviceConfig, disconnect, isReady, and informSensorListeners. However, all of them use the keyword super to make use of the corresponding code in AbstractDevice in addition to device-specific refinements, e.g., setDeviceConfig needs to perform LED configuration updates.

Furthermore, the MetaWearCPro class keeps references to the device board itself as well as all required modules. Each module represents a specific sensor and is provided by MbientLab’s library. Additionally, there is a separate settings module and some sensor modules support multiple use cases such as the MultiChannelTemperature module which delivers device and body temperature data. The device class also keeps track of the device’s state of type enum
DeviceState, which is implemented as AtomicReference to ensure it is thread-safe, and naturally of its current battery level and its Bluetooth address.

In general, this device class, without considering its abstract superclass, is responsible for handling the entire interaction with the physical sensor device including the configuration and management of its sensors, and knowing its device state and overall status. Naturally, the sensor management incorporates working off incoming sensor data which is delineated in-depth later. The main part of AbstractDevice is to set up the configuration Profile instances and other configurational properties which are all device-specific. In addition, it keeps track of IDeviceStateChangedListener and ISensorValueChangedListener listeners. Moreover, there are two private and two inherited methods I want to mention:

```java
notifyBattery(float value);
ConfigOption batStat = super.getDeviceConfigOption("battery");
super.informDeviceListeners(System.nanoTime(), new DeviceProperty(batStat, EDataTypes.floatToBytes(value)));
notifyDeviceState(DeviceState newState);
ConfigOption devState = super.getDeviceConfigOption("device_state");
super.informDeviceListeners(System.nanoTime(), new DeviceProperty(devState, EDataTypes.textToBytes(newState.name())));
```

The first one is used to set the current battery level to the new value, to retrieve the battery ConfigOption in line two, and to call its inherited method in line three which informs the respective listeners stored in AbstractDevice. `notifyDeviceState(...)` updates the AtomicReference to the device’s current state with the passed DeviceState value, retrieves the `device_state` ConfigOption in line five, and then calls its inherited method in line six to inform AbstractDevice’s listeners.

Note that although the keyword `super` is used, in this very case, it does not make a difference whether it is there or not and could even be replaced with `this`, as the called methods are only available in the AbstractDevice class. It would make a difference if MetaWearCPro also implemented these methods, as `this` and no-keyword would then refer to this class instead of AbstractDevice. For clarity and to prevent any future confusion, `super` has been added.

The method `informSensorListeners(...)` is equally important but is called directly from other methods without any utility methods in between. Furthermore, instead of a DeviceProperty instance, a SensorValue instance is passed to the abstract superclass to propagate recorded sensor data.

### 3.1.6. sensorCortrium module

Since the CortriumC3 class is on the same level as the MetaWearCPro class, i.e., simply another sensor device implementation which also extends AbstractDevice, I only list the things that this class does differently or rather what is specific to CortriumC3.

First of all, there is no need to implement ServiceConnection, as the Cortrium Developers chose a different approach and provided a class called ConnectionManager through which users can interact with the device using various methods. These methods cover scanning, discovering, and connection/disconnection events. Also interfaces to handle incoming sensor data are provided and further explained later. As a side note, instead of the MetaWearCPro board, here, it is simply called CortriumC3.
Secondly, CortriumC3 implements the same IDevice methods as MetaWearCPro. It also extends the same methods from AbstractDevice except for setDeviceConfig because it does not require any additional behavior.

Everything mentioned in connection with the methods notifyBattery(…), notifyDeviceState(…), the two superclass methods, and informSensorListeners(…) regarding the MetaWearCPro device applies to the CortriumC3 device as well.

In order to recapitulate the previous sections and to provide a concise and visual summary, Figure 16 to Figure 19 visualize the initialization, connection, DeviceStateChangedEvent, and SensorValuesChangedEvent procedures. Note that I left out some interfaces and utility classes for clarity and simplicity because the diagrams’ purpose is to make the main process flows more apparent and easier to grasp. Nevertheless, all main components are present.

The first diagram shows the initialization and setup, the second one the connection process. The third and the fourth one focus on handling, propagating, and working off DeviceStateChangedEvent and SensorValuesChangedEvent events, respectively. Everything alluded in these diagrams is picked up again, explained in great detail, and illustrated in the setup and data acquisition process section.

This diagram below visualizes the initial application start and setup process and also the first part of the connection process because at the current implementation state, FileRecorderService is started as part of the connect method which immediately tries to connect to all devices. However, this might and should be changed for reasons of structural coherence, as the mere creation of FileRecorderService is not tied to the connection process. The same logic applies to the initialization of DeviceController and all subsequent actions. Therefore, they are separated from the rest and already included above and not in the one below which solely concentrates on the actual connection between the recording device and the sensor devices. The reason it has been implemented the way it is now is also because the user might want to change the main configuration file after the application start-up which would not be possible if everything had been initialized at this point already. Nevertheless, I regard the initialization of FileRecorderService, DeviceController, etc. as a separate task throughout this thesis.

The green box marks the starting point. SenexActivity creates a FileRecorderService instance and performs the previously listed tasks from a) to h) which includes the interaction with the Profile class as indicated. FileRecorderService then sets up the respective log files, .ts files for streaming and .bin files for binary input. It also initializes the DeviceController instance which creates as many DeviceProxy instances as needed to which FileRecorderService registers as IDeviceListener. Each DeviceProxy creates its required number of SensorProxy instances but also instantiates the actual device-specific class, e.g., CortriumC3, in the respective device module. Said class creates a Profile instance based on its very own configuration file. Afterwards, it calls its super class, i.e., AbstractDevice, which stores the configuration and creates a BasicSensor instance for each sensor of the device. Since DeviceProxy implements the ISensorValueChangedListener and the IDeviceStateChangedListener interfaces, it can register itself as device and sensor listener at its corresponding AbstractDevice. Additionally, it creates as many ConfigOption instances as required for each sensor.
The connection diagram in Figure 17 also has its starting point at SenexActivity. From there, the FileRecorderService instance is called which registers a separate DataToFileWriter instance at each SensorProxy, as DataToFileWriter implements the ISensorListener interface. FileRecorderService then uses its DeviceController instance to call each DeviceProxy’s connect(…) method.

What is not visible in the diagram is the fact that DeviceProxy also sets up a newSingleThreadExecutor instance which works off DeviceStateChangedEvent events taken from the respective event queue. Similarly, a newFixedThreadPool executor instance is created to work off SensorValuesChangedEvent events taken from the respective event queue. Both executor services implement Android’s ExecutorService interface.

Finally, DeviceProxy calls the connect(…) method of its corresponding device class, e.g., CortriumC3. Said device class performs further setup steps and then initializes the actual connection process between the controller device, i.e., smartphone, and the physical device attached to the participant’s body.
The last two diagrams about this specific topic cover the two cases in which either the sensor device’s state changes (Figure 18) or a sensor pushes new data values (Figure 19). Both flows originate from the respective device class, i.e., the sensor device itself, because this is where the state change takes place or the data values are produced, respectively. The propagation of information looks and is, in fact, quite similar except for the target class and an additional step performed at SensorProxy. Device state change are ultimately handled by the FileRecorderService class, whereas new sensor values are pushed through to the DataToFileWriter instance in charge. For the latter case, DeviceProxy needs to pass on the values to SensorProxy because it does not know about its listeners, as they are managed solely by SensorProxy. The corresponding SensorProxy instance can then notify its assigned DataToFileWriter which logs the new sensor values to the correct file. Regarding device state changes, the FileRecorderService is responsible to react accordingly. Note that the recently mentioned executor services and event queues are part of DeviceProxy and essential to receiving and distributing DeviceStateChangedEvent and SensorValuesChangedEvent events, respectively.
3.2. Hardware devices

Here, I want to describe the two devices used in my test setup as well as their features including the data they can record and provide before I go into detail about how and in what way they can be configured. As already alluded, the two devices are called MetaWearCPro and CortriumC3.

3.2.1. CortriumC3

The CortriumC3 device (Figure 20) is manufactured by Cortrium ApS which is located in Copenhagen, Denmark. The device supports Bluetooth 4.0 Low Energy (BLE) and live streaming to cloud services. Additionally, it offers a RESTful API as well as an Android and iOS SDK. For complete offline usage, an SD card can be inserted as well. To achieve the best
results, the device needs to be attached to the participant’s (shaved) chest as shown in image XY below. The internal battery supposedly lasts a minimum of 24 hours when recording to the internal SD card. However, it lasts shorter when recording via BLE. Cortrium also offers a smartphone application which enables the user to interact with the CortriumC3 device, retrieve or livestream recorded information as well as visualize it. The information is taken from [11].

Under normal conditions, it can provide the following metrics:

- Electrocardiography (ECG) consisting of three channels/electrodes
- Respiratory rate
- Body surface temperature
- Device temperature
- Accelerometer with three axes, i.e., x, y, and z

![Figure 20 - Digital representation of the CortriumC3 device (taken from [11])](image)

The above list comprises only the directly measured metrics. According to their documentation, the CortriumC3 software kit is able to yield the following derived metrics from above-listed direct metrics:

- Heart rate
- Heart rate variability
- Heart rate recovery
- Core temperature estimate
- Body position/posture
- Fall detection
- Sleep analysis
- Calorie count
- Step count

### 3.2.2. MetaWearCPro

The MetaWearCPro device (Figure 21) is manufactured by Mbientlab Inc. located in San Francisco, CA, USA. This device supports Bluetooth 4.0 Low Energy only but offers software kits for Android, iOS, macOS, and Windows supporting several programming languages such as Java, C#, Swift, Python, JavaScript, and C++. Additionally, Mbientlab offers a smartphone application to interact with their products and which supports similar features as the Cortrium app. On average, the button cell battery lasts approximately 1.5 weeks, but this may differ based on the respective use cases. Image XY shows a picture of the device including the required battery type to show the proportions. The information is taken from [12], [13], and [14].
Besides an RGB LED indicator, the MetaWearCPro device encompasses the following sensors:

- Thermistor temperature sensor
- Accelerometer sensor
- Gyroscope sensor
- Barometric pressure sensor
- Magnetometer sensor
- Ambient light sensor

Moreover, the altitude level can be retrieved as well but it is derived from the barometric pressure sensor, thus not part of the list above.

Overall, one CortriumC3 device and four MetaWearCPro devices were used. The entire test setup including selected results can be found in the section about my test results. Below paragraphs deal with the mentioned configuration files. The most essential information about the configuration files is that there is one main configuration file and one additional configuration file for each device as mentioned in the driver module section. Here, the complete test setup configuration for both devices is shown, explained, and commented on. Furthermore, I explain how device-specific configuration file properties can be overridden by the main configuration file.

### 3.3. Configuration files

To begin with, the main configuration file – henceforth also referred to as main config – is analyzed step by step. Nevertheless, configurational settings which are not relevant to the general understanding or do not provide any additional insights are omitted. This applies to all three configuration files.

Here are the first five lines:

```plaintext
senex.outputFolder=/data
senex.configName=Test
senex.network.SSID=Senex-1
senex.network.password=xyz
senex.devices=cortrium,leftArm,rightArm,leftLeg,rightLeg
```

The first one simply refers to the folder on the smartphone to which the log files should be written. The second one states the arbitrary name of the current main config which is useful to choose between several files in the smartphone application. The two network related lines, i.e., network ID and password, are important for sensor devices which do not connect via BLE but...
use a private wireless LAN hotspot set up by the smartphone. However, such devices are not part of my personal test setup. The last line lists all available devices connected to my specific setup, i.e., one CortriumC3 and four MetaWearCPro devices. The values after the ‘=’ character constitute the keys for the respective sensor settings as explained in the driver module part concerning the Profile class.

The next lines show the Cortrium settings taken from the main config. The CortriumC3-specific as well as the MetaWearCPro-specific configuration files – henceforth also referred to as device config – are both presented at a later point in this section.

```
senex.devices.cortrium.deviceDriver=at.sew.senex.daq.sensors.cortrium.CortriumC3
senex.devices.cortrium.options=btaddr,adapter
senex.devices.cortrium.options.adapter.propValue=BLED112-0
```

As alluded, all four lines feature the keyword `cortrium`. The first line points to the location of the device-specific implementation class which is essential for the DeviceProxy to be able to create an instance of the driver class. The second line shows the available options which are further specified in the last two lines. While the Bluetooth address is obvious, `adapter` refers to the smartphone’s, i.e., the recorder’s, specific Bluetooth adapter ID to which this particular sensor device is ought to connect.

Similar settings exist for the MetaWearCPro device but naturally the device driver class, the Bluetooth address, and the adapter ID are different.

The next line specifies the available sensors as part of the respective device. For CortriumC3, these are:

```
senex.devices.cortrium.sensors=ecg1,ecg2,ecg3,resp,bodyTemp,deviceTemp,accel
```

No further CortriumC3-related configurations are necessary or desired in this case. The sensors are further defined in the device config.

Regarding the four MetaWearCPro sensor devices, only one set of configurational settings is shown. However, it is representative for all four devices and the differences are limited to varying identifiers, separate Bluetooth addresses as well as adapters, and device-specific class paths all of which already addressed above. All other settings are equal for each MetaWearCPro device. Below, I extracted the most interesting because most meaningful lines from the main config:

```
senex.devices.leftArm.sensors=accel, gyro, magnet, baro
senex.devices.leftArm.sensors.accel.sensorOptions=frequency
senex.devices.leftArm.sensors.accel.sensorOptions.frequency.propValue=100
senex.devices.leftArm.sensors.gyro.sensorOptions=frequency
senex.devices.leftArm.sensors.gyro.sensorOptions.frequency.propValue=100
senex.devices.leftArm.sensors.baro.sensorOptions=oversamplingMode,standbyTime
senex.devices.leftArm.sensors.baro.sensorOptions.oversamplingMode.propValue=HIGH
senex.devices.leftArm.sensors.baro.sensorOptions.standbyTime.propValue=
```
62.5
senex.devices.leftArm.sensors.magnet.sensorOptions=power_preset
senex.devices.leftArm.sensors.magnet.sensorOptions.power_preset.propValue=ENHANCED_REGULAR

Line one defines that the MetaWearCPro device for the left arm should use its accelerometer, gyroscope, magnetometer, and barometer sensors. Additionally, four of them are further specified. The accelerometer allows to manually set its sampling frequency (line two) and it will try to send as many updates per second as specified by propValue in line three. The same is true for the gyroscope and its frequency setting in the lines four and five. Line six shows that the barometer’s oversamplingMode as well as standbyTime are to be overwritten. The first one is set to “HIGH” in line seven and the latter one to “62.5” in line 8. standbyTime specifies the minimum standby time which must not be undercut. Setting oversamplingMode to “HIGH” produces better and smoother output values with fewer errors.

Line nine shows that there is a power_preset option available for the magnetometer, but line ten specifies its value which basically means it should be set to a higher performance level than average and replaces the sampling frequency setting which is not available for this sensor.

This is all about the main config, the following paragraphs deal with the two device configs and start with the CortriumC3 device of which the first few lines are as follows:

deviceName=Cortrium C3
description=BLE-Beacon with various sensors!
manufacturer=Cortrium
version=0.1
options=btaddr,device_state,battery
sensors=ecg1,ecg2,ecg3,resp,bodyTemp,deviceTemp,accel

Line one defines the device’s name which is extracted by AbstractDevice when a new device is created. Similarly, the deviceDescription, the manufacturer, the version, the options, and the sensors are all extracted as part of AbstractDevice’s constructor. The most essential information is held by the values belonging to options and sensors which are further specified below starting with btaddr and device_state. However, note that the ECG measurement functionality is split into three separate sensors.

options.btaddr.propName=BT-Addr.
options.btaddr.propDescription=Bluetooth address of the device
options.btaddr.propType=TEXT
options.device_state.propName=Device State
options.device_state.propDescription=The current state of the device
options.device_state.propType=TEXT
options.device_state.propOptions=INIT,DISCONNECTING,DISCONNECTED,CONNECTING,CONNECTED,STARTRECORDING,RECORDING,RESUMING,PAUSING,PAUSED,STOPPING,STOPPED,RESETTING,ERROR
options.device_state.propValue=DISCONNECTED

The lines one, two, and three set the name, description, and data type of the Bluetooth address property. The actual value is defined in the main config as previously explained. This is important for the configuration overwriting part which is explicated afterwards. Lines four through eight are about the device’s state. Besides name, description, and data type, all valid options as
well as the initial value are listed. Note that there would be two more states related to the calibration process which are left out for above-stated reasons.

Subsequently, the battery device option is configured. As usual, the name, description, and data type are set (first three lines). This sensor device only delivers integer battery level updates and is initialized with zero (line four) which is immediately updated upon connection.

```java
options.battery.propName=Device Battery Level
options.battery.propDescription=The current level of the battery
options.battery.propType=INTEGER
options.battery.propValue=0
```

Next, the sensors are configured starting with ECG sensors are described. However, only the first one is shown because the other two feature the same configuration. Below, this configuration including its only sensor option is presented:

```java
sensors.ecg1.sensorName=ECG1
sensors.ecg1.sensorDescription=The first out of 3 electrode sensors
sensors.ecg1.valueNames=ecg1Val1, ecg1Val2, ecg1Val3, ecg1Val4, ecg1Val5, ecg1Val6
sensors.ecg1.valueTypes=INTEGER,INTEGER,INTEGER,INTEGER,INTEGER,INTEGER
sensors.ecg1.sensorOptions=frequency
sensors.ecg1.sensorOptions.frequency.propName=Frequency
sensors.ecg1.sensorOptions.frequency.propDescription=Frequency of sampling in Hz
sensors.ecg1.sensorOptions.frequency.propValue=250
sensors.ecg1.sensorOptions.frequency.propOptions=250
sensors.ecg1.sensorOptions.frequency.propType=INTEGER
```

Besides the sensor name, ECG1, and its description in the first two lines, line three and four define the values delivered by the sensor device. `valueNames` lists six parameters which are part of each data package sent on each ECG sensor update. `valueTypes` defines the corresponding data types for each parameter. This information is extracted by and stored in the BasicSensor class integrated in a ValueDescription instance as described above. Ultimately, however, this information is used by FileRecorderService to know which type of log file to create. Currently, these configurational settings are not used anywhere else in the code. However, it could certainly be used to dynamically convert the received data into bytes, i.e., this line

```java
EDataTypes.intToBytes(ecgData.getRawEcg1Samples()[0])
```

which fetches the first of six values from the incoming ECG1 data package could be replaced with a dynamic check to know which data type is expected and then call a helper method to convert the given data of the given type into bytes without – regarding the code – knowing its actual data type beforehand.

Line five indicates that there is only one sensor option which is called `frequency`. Line six and seven are only included for the sake of completeness. Line nine shows all valid options for `frequency` and line 8 defines the value to be used and which must be part of `propOptions`. The order in which these settings are defined does not matter. The last line declares the data type for this sensor option to which `propValue` and `propOptions` must comply.

The next sensor to be described is responsible for the respiration rate which is measured in breaths per minute (bpm) as float value. Again, there is only one sensor option which is
frequency. As before, the sampling rate allows only one pre-defined value which is set to 41.67 and, therefore, represents a float value as well.

```java
sensors.resp.sensorName=Respiration
sensors.resp.sensorDescription=Respiratory rate in breaths per minute
sensors.resp.valueNames=bpm
sensors.resp.valueTypes=FLOAT
sensors.resp.sensorOptions=frequency
sensors.resp.sensorOptions.frequency.propName=Frequency
sensors.resp.sensorOptions.frequency.propDescription=Frequency of sampling in Hz
sensors.resp.sensorOptions.frequency.propValue=41.67
sensors.resp.sensorOptions.frequency.propOptions=41.67
sensors.resp.sensorOptions.frequency.propType=FLOAT
```

The device and body temperature are both in degrees Celsius, return INTEGER values, and allow a sampling of 0.16 Hz:

```java
sensors.bodyTemp.sensorName=Body Temperature
sensors.bodyTemp.valueNames=°C
sensors.bodyTemp.valueTypes=INTEGER
sensors.bodyTemp.sensorOptions.frequency.propOptions=0.16
sensors.deviceTemp.sensorName=Device Temperature
sensors.deviceTemp.valueNames=°C
sensors.deviceTemp.valueTypes=INTEGER
sensors.deviceTemp.sensorOptions.frequency.propOptions=0.16
```

Note how valueNames is mostly important to know how many values a data package contains and to be mapped with valueTypes. However, the actual name does not matter to the application but is important for the user to know what kind of information to expect.

The last sensor which is part of the CortriumC3 device and requires an explanation is the accelerometer whose name and description are defined in the first two lines. As mentioned before, it consists of 3 axes which are usually called x, y, and z (line three). All of them are float values as indicated in line 4. The meaning of each axis and how to interpret them is part of the chapter about my test results where everything about the Accelerometer is explained in detail.

This sensor also only has one additional option, i.e., frequency, which allows one value: 41.67

```java
sensors.accel.sensorName=Accelerometer
sensors.accel.sensorDescription=3 axes accelerometer
sensors.accel.valueNames=x, y, z
sensors.accel.valueTypes=FLOAT,FLOAT,FLOAT
sensors.accel.sensorOptions.frequency.propOptions=41.67
```

The second and last sensor device to describe is the MetaWearCPro chip. As it has a lot in common with the CortriumC3 device, I rather but not exclusively focus on the differences of which the first few lines already show some:

```java
deviceName=MetaWearCPro
deviceDescription=BLE-Beacon with various sensors!
manufacturer=mbientlab
version=0.1
```
Besides *deviceName* and *manufacturer*, *options* and *sensors* changed as well, as the MetaWearCPro device features an LED in addition to the Bluetooth address, device state, and battery level. Furthermore, it offers various (other) sensors which include an accelerometer, a gyroscope, a magnetometer, a barometer, an altimeter, a temperature as well as an ambient light sensor. However, latter one was not used for my individual test and is also not listed in the main config which overrides these settings. Similarly, the temperature sensor was omitted because the CortriumC3 device already delivers this data. The altimeter was left out as well, as it is not a separate sensor, but it is based on the barometer which was included and, therefore, provides all necessary data. *btaddr*, *device_state*, and *battery* are the same as before. The LED option, however, offers various settings:

```
options.led.settings=color,riseTime,pulseDuration,repeatCount,highTime,
  highIntensity,lowIntensity
options.led.settings.color.propName=LED-Color
options.led.settings.color.propType=TEXT
options.led.settings.color.propOptions=RED,GREEN,BLUE
options.led.settings.color.propValue=
options.led.settings.riseTime.propName=LED-Riserime
options.led.settings.riseTime.propDescription=Rise time of the LED in ms
options.led.settings.riseTime.propType=SHORT
options.led.settings.riseTime.propValue=0
options.led.settings.pulseDuration.propName=LED Pulse Duration
options.led.settings.pulseDuration.propDescription=Pulse duration time
  of the LED in ms
options.led.settings.pulseDuration.propType=SHORT
options.led.settings.pulseDuration.propValue=1000
```

The first line shows all valid LED settings which are further specified in the lines below. However, some are left out because most of them are quite similar. The color property allows three different values of data type string. Initially, the LED is off which is why *propValue* is empty. *risetime* describes the time in milliseconds which the LED needs to switch between its low and its high state. The exact brightness for each state can be defined via *lowIntensity* and *highIntensity*, respectively. Since this transition should happen instantaneously, the value of data type short is set to zero. *pulseDuration*, which is set to one second, defines the entire time it takes to go from low to high and back to low. *highTime* specifies the amount of time the LED spends in its high state. For instance, if this value is set to 500 ms and with the already configured settings in mind, it causes the LED to switch from low to high as fast as possible, stay there for half a second, and then take about half a second to go from high to low again. Setting *repeatCount* to -1 leads to a never-ending loop which means it does not stop automatically but repeats to pulse until another command tells it to do otherwise.

The first sensor to be configured is the accelerometer which, similarly to the CortriumC3 device, also features three float value axes using the identifiers x, y, and z. However, this sensor has two options, i.e., *frequency*, and *range*. Unlike CortriumC3, MetaWearCPro offers several frequency options of type float listed in line five. The second parameter specifies the accepted
range of data measurement in g (line eight) based on the options of data type short specified in line nine. A value of eight means all values between minus eight and eight are accepted, i.e., ultimately lead to SensorValuesChangedEvent events.

```java
sensors.accel.valueNames=x,y,z
sensors.accel.valueTypes=FLOAT, FLOAT, FLOAT
sensors.accel.sensorOptions=frequency, range
sensors.accel.sensorOptions.frequency.propValue=12.5
sensors.accel.sensorOptions.frequency.propOptions=0.78125, 1.5625, 3.125, 6.250, 12.5, 25, 50, 100, 200, 400, 800, 1600
sensors.accel.sensorOptions.range.propName=Measurement Range
sensors.accel.sensorOptions.range.propDescription=Range of measurement in g
sensors.accel.sensorOptions.range.propValue=8
sensors.accel.sensorOptions.range.propOptions=2, 4, 8, 16
sensors.accel.sensorOptions.range.propType=SHORT
```

The gyroscope features three axes, x, y, and z, of type float and frequency and range can be set as well. The latter one is measured in degrees per second and set to 1000 chosen from the possible options in line eight of data type short.

```java
sensors.gyro.valueNames=x, y, z
sensors.gyro.valueTypes=FLOAT, FLOAT, FLOAT
sensors.gyro.sensorOptions.frequency.propValue=50
sensors.gyro.sensorOptions.frequency.propOptions=25, 50, 100, 200, 400, 800, 1600, 3200
sensors.gyro.sensorOptions.frequency.propType=SHORT
sensors.gyro.sensorOptions.range.propDescription=Range of measurement in degrees/sec
sensors.gyro.sensorOptions.range.propValue=1000
sensors.gyro.sensorOptions.range.propOptions=125, 250, 500, 1000, 2000
```

The magnetometer measures surrounding magnetic fields as described in line one. The measurement unit is called Microtesla and is delivered as data type float (line three). This sensor provides one option called power_preset which can be set to one of the values listed in line six. This setting influences the update rate, the power consumption, and the measurement precision. It is set to the lowest possible option because this information is barely relevant for the project’s purpose.

```java
sensors.magnet.sensorDescription=Measures surrounding magnetic fields
sensors.magnet.valueNames=t
sensors.magnet.valueTypes=FLOAT
sensors.magnet.sensorOptions=power_preset
sensors.magnet.sensorOptions.power_preset.propValue=LOW_POWER
sensors.magnet.sensorOptions.power_preset.propOptions=LOW_POWER, REGULAR, ENHANCED_REGULAR, HIGH_ACCURACY
sensors.magnet.sensorOptions.power_preset.propType=TEXT
```

The thermometer measures the surrounding temperature and delivers values of the data type float. No further options are available. This sensor was not part of my test. The barometer, on the contrary, offers several sensor options and measures the surrounding pressure in Pa of data type float. filterMode defines of how many samples the resulting average value should consist. If
set to one, then each value gets sent. In this case, it is set to four which means not every value gets delivered but always the next four incoming values are aggregated and used to calculate the average value which then gets sent to the API. The way `oversamplingMode` works is explained as part of the main config because the latter one overwrites this very value here. The same applies to `standbyTime`.

```java
sensors.baro.sensorName=Barometer
sensors.baro.sensorDescription=Measures surrounding pressure in Pa
sensors.baro.valueNames=Pa
sensors.baro.valueTypes=FLOAT
sensors.baro.sensorOptions=filterMode,oversamplingMode,standbyTime
sensors.baro.sensorOptions.filterMode.propValue=4
sensors.baro.sensorOptions.filterMode.propOptions=1,2,4,8,16
sensors.baro.sensorOptions.filterMode.propType=SHORT
sensors.baro.sensorOptions.oversamplingMode.propValue=LOW_POWER
sensors.baro.sensorOptions.oversamplingMode.propOptions=SKIP,ULTRA_LOW_POWER,LOW_POWER,STANDARD,HIGH,ULTRA_HIGH
sensors.baro.sensorOptions.oversamplingMode.propType=TEXT
sensors.baro.sensorOptions.standbyTime.propValue=1000
sensors.baro.sensorOptions.standbyTime.propOptions=125,250,500,1000,2000
sensors.baro.sensorOptions.standbyTime.propType=SHORT
```

Since the altimeter is part of the barometer, there are no additional configurations required. The ambient light sensor measures the surrounding light in Millilux (mlux) of data type long. It was not part of my personal test, however, it provides three options: `gain`, `integration_time`, `measurement_rate`. `gain` controls the data range, `integration_time` specifies the measurement time of each cycle, and `measurement_rate` defines the data update rate. All three are of data type short.

```java
sensors.als.sensorOptions=gain,integration_time,measurement_rate
sensors.als.sensorOptions.gain.propValue=4
sensors.als.sensorOptions.gain.propOptions=1,2,4,8,48,96
sensors.als.sensorOptions.gain.propType=SHORT
sensors.als.sensorOptions.integration_time.propValue=100
sensors.als.sensorOptions.integration_time.propOptions=50,100,150,200,250,300,350,400
sensors.als.sensorOptions.integration_time.propType=SHORT
sensors.als.sensorOptions.measurement_rate.propValue=50
sensors.als.sensorOptions.measurement_rate.propOptions=50,100,200,500,1000,2000
sensors.als.sensorOptions.measurement_rate.propType=SHORT
```

All sensor-specific information is taken from the official MetaWear Android documentation [15].

The next part deals with the way the main config can overwrite the device config settings. We already know that, during the initialization process, SenexActivity creates the main Profile instance based on the main config file in the beginning and that each device-specific class creates its own Profile instance based on its device config file. To overwrite the device config (Profile), SenexActivity first needs to pass down the main config (Profile) instance via DeviceController to each DeviceProxy instance which then splits it into device options and
sensor-related ConfigOption lists. Both are passed on to their respective device class instance on connect.

However, the initialize method performs another key step. It creates a copy of all device config entries which are also in the main config, i.e., the ones which should be overwritten by the main config, but with the value from the main config. Then, it stores this list to be used later. This step is similarly performed for each sensor with the difference that the respective configurational settings are passed to the SensorProxy constructor and not stored at DeviceProxy. In addition, the device config is used at AbstractDevice’s constructor to know about all available sensors and to create the respective BasicSensor instances.

For device and sensor config, several checks are performed to ensure the validity of the main config and its properties. This also guarantees that the device config only contains valid entries and that neither invalid ones can be added, nor valid ones overwritten. Naturally, this is based on the assumption that the device config is correct and valid per se, as the main config is the more precarious ones because it can and should be easily modifiable by the user.

On connect, said device and sensor settings are propagated to the sensor device instance where they are used to call two specific methods defined in AbstractDevice to update the device and sensor config. In order to pass the sensor settings copies, they are retrieved from the respective SensorProxy instances, because DeviceProxy does not know them as previously mentioned. Note that the latter one is part of the device config file but treated separately because each sensor only needs to know its own configuration.

The method to overwrite the device-related settings merely goes through the list of previously stored duplicates, i.e., the entries from the main config, and replaces the ones with the same key in the device config without touching the rest. This only overwrites the general device settings but not yet the config related to its sensors. To still have access to the original settings, a deep copy is created beforehand. The described scenario is true for the CortriumC3 and MetaWearCPro device. However, the latter one performs additional LED configurations based on the main config which needs to be done right away.

The method to overwrite the sensor-related settings first updates the list of selected sensors which can contain all available sensors or fewer but not more. Moreover, it calls another method at BasicSensor to finally overwrite the sensor config with the new values taken from the main config. This procedure uses the same logic as the one to overwrite the device config.

Figure 22 provides a visual representation of the overwriting process.
3.4. Setup and data acquisition process

Since all prerequisites have been described and clarified, the following segment deals with the application setup and data acquisition process focusing on the programmatic point of view. This means that I go through the entire application and explain how the various components interact with each other and what their purpose is. Furthermore, this part contains many visuals which illustrate several implementation details on a more abstract level which makes the underlying construct easier to grasp than mere code line citations. Naturally, I still cannot include every single detail which means that variable initializations and other minor and self-evident processes are left out. Before the first diagram is presented, the used elements shall be explained in Table 1.
3.4.1. App start-up

The first diagram (Figure 23) shows the application start-up once more but in greater detail than before. Note that the main difference between root and SENEX Profile is that the former one includes information about the general log file location and debug settings. SENEX Profile is the one which I commonly address as main config. The other one is insignificant to all other parts of my thesis but this one.

The App Start state indicates the launch of the application, whereas the End state indicates that the initial start-up has been completed, i.e., SenexActivity’s onCreate() is done, but not that the application has been closed. Furthermore, note that DeviceButton also implements IDeviceListener so it can react to DeviceState changes independently from its creator SenexActivity and the only other class which implements IDeviceListener FileRecorderService. At the time all DeviceButton instances are created, the UI setup is complete as well. This diagram does not yet include FileRecorderService and subsequent initialization processes.
3.4.2. Device initialization

As described and reasoned in the previous section, the first part of the connection process which, from a structural point of view, still belongs to the initialization context is again extracted from the connection methods and shown together with the remaining initialization tasks composing the initialization part.

The next diagrams illustrate said initialization part and due to its size, it is split into several ones which include the most relevant (abstract) class and interface methods as well even if not used thus far. Furthermore, not all relations between all elements can be shown every time they are used because it would add unnecessary complexity which could lead to confusion. However, all essential relations and elements are included. Afterwards, the connection process itself is visualized in-depth.

Since the big diagrams are split, they contain a circled letter in the upper left corner which describes the different areas in the big picture which is shown subsequently after each topic, i.e., initialization part, connection process, and data logging. Moreover, my comments in between the
diagrams are only complementary because most of the information is already integrated into the diagrams.

Diagram A (Figure 24) shows the first part of the Senex-Recorder module and continues where the previous one left off. SenexActivity starts FileRecorderService which initializes DeviceController in diagram B (Figure 25). The first two text boxes below FileRecorderService call the implementation of registerDeviceListener(…) in diagram D (Figure 27). The positive option of the decision node in A continues at the top left of diagram B. The negative one is continued at the top of diagram E (Figure 29) and creates a DataToFileWriter instance which is further specified in diagram B and implements the ISensorListener interface in diagram D. Additionally, ISensorProxy in D is used to create the respective SensorProxy instance in B. FileRecorderService as well as DeviceButton both implement the IDeviceListener interface which is also part of diagram E. C (Figure 26) only contains a small fraction of information which belongs to IDeviceProxy and SensorProxy. Also, E connects to the left side of B and B to the left side of D. F (Figure 28) connects to the left side of G (Figure 30) and continues the IDevice interface, while G connects to the top of H (Figure 31) and connects DeviceProxy with the respective device class and the latter one with AbstractDevice. The entire diagram can be found in Figure 32.

In summary, SenexActivity starts FileRecorderService which configures all devices, sensors, listeners, and log files. Then FileRecorderService hands over to DeviceController which actually creates the DeviceProxy instances. Each DeviceProxy instance then creates SensorProxy instances as needed as well as the specific device classes whose super class (AbstractDevice) takes care of further configuration and BasicSensor creation. The de facto end state is part of SenexActivity, as it initializes the described process. However, I connected it to <DeviceClass> because once all devices are created, the scenario ends from an abstract point of view. Note that to save space, diagram F comes before diagram E.
Figure 24 - Initialization A
Figure 25 - Initialization B

1. creates a StreamToFileOutputStream handling over the current IDeviceProxy instance, the sensor’s ID, a new and empty File for logging/recording using the sensor’s ID and the ending “.ts” and the defined storage path
2. assigns the stream to respective sensor so it gains access
3. calls super(File, true) with true meaning that new content should be appended to potentially existing content
4. saves a reference to the respective IDeviceProxy and the sensor’s ID

Figure 26 - Initialization C

1. saves a reference to the respective IDeviceProxy and the sensor’s ID
2. creates the output File with the given name at the given Path
3. creates a new FileOutputStream using the File instance and the instruction to append new content naturally. It keeps this stream instance to retain access

C

keeps all sensor-specific configuration values, its respective Sensor as well as DeviceProxy instance, and its listeners

IDeviceStateChangedListener

is part of

deviceStateChanged(long timestamp, final IDevice.IDeviceProperty... properties)
Figure 29 - Initialization E
Figure 30 - Initialization G

1. sets a reference to the device profile, i.e., the config of the respective device class
2. saves name, description, manufacturer, version, device options, and available device sensors
3. for each available sensor, a BasicSensor is created as Sensor (see other path)

Figure 31 - Initialization H

possible values: UNKNOWN, DISCONNECTED, CONNECTING, CONNECTED, CALIBRATING, CALIBRATED, STARTRECORDING, RECORDING, RESUMING, PAUSING, PAUSED, STOPPING, STOPPED, DISCONNECTING, RESETING, ERROR

DEVICESATE (E)

registerDeviceStateListener()DeviceStateListener

registerSensorValuesChangedListener(SensorValueChangedListener)

setDeviceConfig(ConfigOption ... property)

connect(Context androidContext, Logger logger, long currentSystemNanos, List<ConfigOption> deviceProperties, Map<String, List<ConfigOption>> sensorProperties)

start()

stop()
Figure 32 - Initialization overview
3.4.3. Connection process

Figure 33 - Connection process A
Figure 34 - Connection process B
Figure 35 - Connection process C

1. checks the availability of all selected sensors and makes sure the required data can be provided by the available sensors; no further configuration at this point

2. gets the Bluetooth address from the updated config (because only available in the main config file) to connect

1. sets up a reference to the current device state and battery level

1. compares the discovered device’s Bluetooth address with the one in the main config
2. if it matches, a reference to the CortriumC3 device is set and the scanning process is stopped

1. sets the device's mode to active and sets up the battery listener
2. further device and sensor configuration according to the main/device config (see MetaWearPro) should be performed here but not implemented at this point

overwrites standard device and sensor configuration with main config
The four diagrams above show the different sections which are part of the overview diagram below. The connection process can start once the application start-up and initialization tasks are completed. In diagram A (Figure 33), SenexActivity calls FileRecorderService's connectDevices method which forwards to DeviceController via IController and DeviceProxy via IDeviceProxy. DeviceProxy then sets up event queues and ExecutorService instances to handle sensor device data updates which is described in the next section about data logging. Afterwards, an AsyncTask instance is created and executed to register listeners for general device as well as sensor updates and to connect to the respective device. This is done by calling the corresponding device’s connection method which is in the upper right corner of diagram B (Figure 34). Depending on the exact device implementation, the final connection tasks differ. The MetaWearCPro device is described in B and D (Figure 36), while CortriumC3 is almost entirely described in C (Figure 35). In both cases, the connect method takes care of initialization the connection process including the setup of listeners, callbacks, and services dependent on the specific device. This also entails the process of reading the respective Bluetooth address from the main config in order to search for the required sensor device(s). Once found, the connection is established, additional settings are applied, and a notifyDeviceState method call is triggered to inform all listeners, i.e., FileRecorderService and DeviceButton instances, about the device status update. How this notification task works exactly is explained in the next segment about data logging. The entire diagram can be found in Figure 37.

How the main configuration overwrites the device configuration has already been described in detail in the previous section and, thus, does not require further explanation at this point.
3.4.4. Data logging

The data logging or gathering process has already been explained in the previous section. Below, this process is depicted in greater code detail together with additional explanations and remarks. It is split into two separate diagrams and, as before, followed by an overview diagram which shows the entire process. The first diagram (A) in Figure 38 represents updates related to device state changes, while diagram B (Figure 39) illustrates sensor data updates.

At the bottom of diagram A, a sensor device sends data updates to its respective device class, i.e., the recording smartphone. In my case, this is either a MetaWearCPro or a CortriumC3 device. Depending on whether the update is battery or device state-related the left or right branch is chosen, respectively. In each case, the corresponding super class, i.e., AbstractDevice, is informed about the new data. AbstractDevice takes care of notifying all its IDeviceStateChangedListener listeners which are all DeviceProxy instances. In the DeviceProxy’s deviceStateChanged method, the new data is packaged into a DeviceStateChangedEvent object and passed on to the respective event queue which gets asynchronously worked off in the run method of its corresponding ExecutorService instance. Said method will call the deviceValuesChanged method for each IDeviceListener instance. The latter one can either be of class DeviceButton or FileRecorderService. In the first case, the user interface button gets updated to reflect the new device state or battery value. In the second case, FileRecorderService reacts accordingly, e.g., initialize a reconnection procedure in case of a disconnection and log the error.

If the device input is sensor-related, the respective sensor’s callback method is called followed by the notifySensorListeners method which calls its super method in AbstractDevice. AbstractDevice then calls the corresponding DeviceProxy’s sensorValuesChanged method. This happens for each ISensorValueChangedListener listener stored at AbstractDevice. As before, DeviceProxy hands over the new data to an event queue wrapped into a SensorValuesChangedEvent object. Similarly, an ExecutorService instance works off this type of events in its run method. However, and unlike before, an additional step is required at this point. Said run method calls the valueChanged method of the corresponding SensorProxy instance because only the latter knows about its listeners which are of type ISensorListener. For each of these listeners – usually only one, i.e., a DataToFileWriter instance, the sensorValueChanged method is called. In this last method, the data is properly formatted and written to the respective log file. Afterwards, the event is completed.
Figure 38 - Data logging A
Figure 39 - Data logging B
3.5. Conclusion

In this chapter, I analyzed our framework in detail including its purpose, the technologies used, its components and modules, two sensor devices including their configuration. Moreover, I presented the setup, data acquisition process, and logging. To underline the generic nature of the framework, I briefly explain the four steps required to integrate another sensor device below.

1. Add the driver package/library to the project source.
2. Add a new device class representing the new sensor device.
3. Add a new sensor device config file.
4. Edit the main config file to include the new sensor device; otherwise it would not be considered part of the application.

First of all, we need to add the Android library provided by the manufacturer to the project’s source folder. This is required for the next step, so we can create a new sensor device class which has access to the respective package methods to communicate with the hardware device. This class must extend the abstract class AbstractDevice which stipulates that a constructor accepting a Profile class instance, i.e., the device’s config file, as well as the connect, start, stop, pause, resume, calibrate, and reset methods are implemented. Third, and to have input for the constructor, a new config file is necessary. It needs to adopt the same structure as described in...
the sections above. Regarding all options and possible values, the device specification manual plays an essential part. Finally, the main config file needs to at least list the new device here:

```
se nex.devices=newDevice,…
```

and specify its driver, i.e., device class name, here

```
se nex.devices.newDevice.deviceDriver=newDeviceClass.
```

Overriding device specific configurational values in the main config is an additional but optional step.

The framework is generic because one does not have to touch any existing Java files, as the correct implementation – at least regarding the prescribed methods – is ensured when extending the abstract class AbstractDevice which itself implements the IDevice interface. All interface methods which are not covered by AbstractDevice must be implemented by all classes extending AbstractDevice. In addition, the respective manufacturer library and a new config file need to be added as well as the main config file adapted. This is everything required to incorporate another sensor device into the system.
4. Test Results

To show the created framework in action, an individual self-test was performed. The writer of this thesis acted as the test person as well. The purpose of this test is to demonstrate an exemplary set-up of selected sensor devices in connection with the framework. The Scenario section below gives details about how the experiment was exactly performed and which sensor devices were used. The Results section describes the logged data at an abstract level, i.e., no in-depth analysis of the received sensor values, as it is not essential or conducive to the test’s objective.

4.1. Scenario

The sensor devices which were chosen to be part of the test are: one CortriumC3 and four MetaWearCPro devices. For details about the devices itself and which data they can provide, see chapter three.

The CortriumC3 device was placed on the test subject's chest to measure data appropriately. Both hands and ankles were equipped with one MetaWearCPro device each. Since the actual data output and its concrete meaning is of no relevance to the thesis, no device calibration was performed. The collected data can, nevertheless, be interpreted in a reasonable manner to provide proof of concept. Figure 42 shows a possible device setup but includes more than what was used for this specific test. The sensor devices used in this case are marked with a red circle, i.e., the ones with the white ribbon.
4.1.1. Test procedure

In the test scenario, I simply walked down and up a stairwell (Figure 41). The generated output is discussed later on. It is important to note that it was a usual stair descent and ascent process. I did not use the hand rails for the descent part of the stairwell, i.e., my arms were dangling casually at waist level. For the ascent part, the hand rails were used for about 50% of the stairs, i.e., arms bent about 45 degrees to the side. Before descending and ascending, respectively, there was a very brief break in which the participant rested in a relaxed and upright position with their arms stretched out orthogonally to the ground. As a matter of course, the test person had to perform a 180 degree turn between the descent and ascent process.

4.1.2. Device configuration

The specific CortriumC3 as well as the MetaWearCPro configuration files including the way they work has already been described in the Modules section of Chapter 3. Additionally, the main configuration file was dealt with. This segment, however, deals with the specific device sensor setup used for the above-mentioned scenario which is necessary to explain the logged data and form a connection between configuration input and data output. Therefore, it only talks about the main configuration file particularly designed for this purpose. The device-specific configuration files are identical to what was previously described.

Regarding the CortriumC3 device, it tells the device to deliver all sensor data except for accel_raw, as the acceleration data in byte form is not required. Apart from that, it only sets the correct
Bluetooth address as well as to which adapter to connect. The log files encompass the following data: ECG1, ECG2, ECG3, respiratory rate, body temperature, device temperature, and acceleration.

In terms of the MetaWearCPro device, the following is true for all four devices. Naturally, it also sets the correct Bluetooth address and which adapter to use. Additionally, it excludes the temperature sensor, as it is not essential and already part of CortriumC3. Thus, the sensors which log data are: accelerometer, gyroscope, magnetometer, barometer, and altimeter. Apart from that, only the frequency of data sampling for accelerometer, gyroscope, magnetometer, and barometer are adjusted.

4.2. Results

The test was completed successfully in almost exactly one minute and all required data was recorded. The short rests, which are part of the one minute, in the beginning, in between when changing direction, and in the end, took approximately equally long. This means that the time was equally distributed with respect to the ascent and descent of the stairwell. To prove the experiment’s success, some sensor data is visualized below including fundamental information and explanations about how it needs to be interpreted. Furthermore, a connection between the scenario and its results is drawn. The incorporated sensor values comprise: CortriumC3 body temperature, CortriumC3 device temperature, CortriumC3 respiratory rate, CortriumC3 acceleration data, MetaWearCPro acceleration data of the left arm and leg as well as of the right arm and leg, MetaWearCPro barometer data of the left arm, and gyroscope data of the right arm and left leg. The barometer output hardly varies for all four devices, so only one is selected for the analysis. The gyroscope data was limited to two devices, as the most interesting values are provided by the accelerometer sensor of which all four outputs are presented below.

In the end of the Results section, a combined data analysis approach is presented to show the joint value of such recorded information. The more sensor values are added up and linked, the more valuable the overall scenario’s expressiveness and significance becomes.

For reference, the test was carried out on October 24th, 2017.

4.2.1. CortriumC3 body temperature

The body temperature measured by CortriumC3 is depicted in Figure 43. As it has been attached to the chest, the recorded values are slightly lower than what is usually referred to as the normal body temperature. Although several peaks and lows can be observed instead of a steadily increasing line, on average, it stays at 35.50 degrees Celsius. On the one hand, one could argue that the values are not very reliable and that they fluctuate too much. On the other hand, however, the recorded period is very short, the fluctuations are still within a reasonable limit, and the average value delivers a credible result nonetheless. Assuming that the values are very precise and correct, one could argue that the temperature slightly dropped as I went down the stairs, as the device is placed on the body and exposed to ambient temperature. However, another peak can be observed after 00:25 which indicates even higher values than at the start but does not correlate with the actual position on the stairs. In fact, at this point, I was at the bottom of the stairs. This means, in return, that the temperature does not seem to be dependent on the ambient temperature or at least only to a very limited extent. It could also mean that the body temperature increased faster than the ambient temperature was able to influence it.
Overall, it is yet very unlikely that the ambient temperature had any effect on the electrodes attached to the body. It is more likely that it influenced the device temperature shown in the next chart and is discussed there.

The body temperature diagram merely shows the results of the recording as well as the accuracy of the values for a brief test period. This chart alone is meaningless to determine the process of ascending or descending stairs but becomes more useful in connection with other data.

4.2.2. CortriumC3 device temperature

Figure 44, in contrast, shows the device’s temperature. Here, it is clearly visible that the device’s temperature is adjusting to the one of the body. However, this happens in very small steps, since it is only over the course of one minute. Additionally, the device had been kept at room temperature and had had time to adjust to the subject’s body temperature before the actual experiment started. Although the adjustment process is very limited, it is still a steady increase within the recorded minute. The previously alluded effect of the ambient temperature cannot be perceived or indicated in this diagram. This is most likely due to the fact that the temperature sensor is built-in the device’s core part and therefore independent from quick and slight ambient temperature changes. Additionally, the device was worn on my chest but below my T-shirt which might have further prevented external influences. Logically, this also applies to the body temperature described before. Another aspect which can be considered is that the device itself produces heat as well while operating. This might also be the main reason for the rise in temperature or perhaps it is a mixture between both, the device’s operating temperature increase and my body’s heat. In this case, a longer test over more minutes or even hours as well as tests which include a from the body detached device would be interesting. However, this is not part of my thesis, as the main purpose is to show the operational readiness of our framework put into action. Nonetheless, it is important to ensure the validity and at least partial usefulness of the recorded results even if it is only for such a brief recording period. Thus, I expect the
temperature values to rise even further over time, but I do not think that they would go as high as my body temperature, since there would still be a small physical distance between my body and the device. Furthermore, the ambient temperature is usually much lower than the one of the body which means that the first one will, to a certain extent, compensate for the influence of the latter.

4.2.3. CortriumC3 respiratory rate

Since the respiratory rate was gathered as raw data and the conversion process is not part of this thesis, the values shown are not depicted as breaths per minute but another undefined unit. Nevertheless, a slight positive tendency towards shorter breathing cycles can be noticed, as the test person gets more exhausted during the process. The respective graph can be found in Figure 45. For the interpretation of this diagram, the actual measurement unit is extraneous as long as it correlates with my respiration. It can be observed that, in the beginning, my breathing rhythm is quite stable and starts to increase when I start descending the stairs. Then, it stays invariant on average regarding the break at the bottom of the stairs. On my way up, my respiration rate seems to marginally rise again before it levels off during the last break at the top of the stairs prior to the recording stop.

As for my personal condition during the experiment, I had time to calm down completely beforehand which means that the initial values can be regarded as my resting respiratory rate and the following ones can be seen in relation to it which allows for a comparative point of view. I experienced the ambient temperature as comfortable, therefore, no external difficulties affected my personal condition in a way which would influence the results shown below. Accordingly, this connotes that the recorded values are a pure result of ascending and descending the stairs without any other aggravating factors.
4.2.4. CortriumC3 accelerometer data

In Figure 46, we can see the accelerometer data from the CortriumC3 device. This is also the first diagram where the movement breaks are clearly visible and indicated by the very small amplitude in the beginning, middle, and end of the experiment. Additionally, it shows that the first and last break lasts about five seconds while the one in the middle is almost twenty seconds long. The ascent and descent process take approximately fifteen seconds each. Since the positive x-axis is pointed towards the sky – or ceiling in my case – and the negative one towards the ground, its values are centered around 1, whereas the values of the y-axis and z-axis are centered around 0. This is due to the fact that these values represent force of gravity (measured in g) applied to each respective axis. In the beginning, when the positive x-axis is pointing upwards, there is exactly one g required to keep the device from falling which is applied through my body. In other words, my body exerts one g on the device’s positive x-axis to keep it in position. The reason why the x value does not show an exact value of one in the beginning is because this value is also influenced by my body posture as well as the exact position on my chest. As I start moving down the stairs, all values begin to fluctuate. This can be explained as follows: each step down influences the x-axis in a way which produces values between 1 and 0.5 because the g-force applied to the device through my body is reduced to almost 50 %. This is only logical because otherwise I would not be able to step down. However, there are also values which range between 1 and 1.5 and even slightly exceed the latter occasionally. This behavior results from my body’s back-bouncing movement after reaching the next step. On my way up, the described behavior occurs in reverse order, i.e., taking a step up increases the g-force applied to the device’s positive x-axis and because one usually makes a greater step than what is actually required to reach the next level, one needs to slightly step down again to set the foot on solid ground. Consequently, this results in a lower g-force applied to the device’s positive x-axis. However, a negative value is never reached, since this postulates a quite fast step down. A similar behavior can be observed regarding the other two axes. The y-axis indicates movements to the left and right which produces values roughly ranging from -0.25 to 0.3. A
positive y-axis value indicates movement to the right, whereas a negative one represents movement to the left. Naturally, these side movements are rather limited when walking up and down some stairs and traced back to a not 100% straight walking path. This is also true for the z-axis which shows comparable values and indicates forward and backward movement. Forward movements usually occur when taking the next step while backward movements can be related to the short stops or bouncing movements one commonly performs when reaching the next level without stopping the overall ascent or descent.

4.2.5. MetaWearCPro accelerometer data

This section deals with the accelerometer data of the MetaWearCPro devices attached to the left hand and leg as well as the right hand and leg of my body. First of all, I analyze each of them separately before I undertake a combined approach to show the strength of connecting all four data streams. The analysis itself relates to the accelerometer data of the CortriumC3 device where applicable.

Figure 47 shows the respective data for my left hand. In fact, the sensor device is attached to the back of my hand as visible in Figure 42. This also means that already simple hand movements influence the accelerometer’s x-, y-, and z-axis. As before, the three breaks are clearly evident. Here, the y-axis is the one which is close to -1 in the beginning and the other two are between 0 and 0.5. Again, this depends on the exact position of my hand. However, the negative y-axis value of -1 indicates that the g-force is applied along the device’s negative y-axis which means that it points at the ceiling. The positive y-axis points at the ground and the x-axis to the front (positive) and back (negative) from my point of view. The positive z-axis points away from my body while the negative one exactly faces my body. At this point, it needs to be mentioned that both my arms are initially casually dangling from my body without any deliberate movements. As indicated by the values, the back of my hand points somewhat to the front (positive x-axis) and side (positive z-axis). Since no hand rails were used for walking down the stairs, the values of all three axis merely show a general
fluctuation related to the usual hand and arm movements when walking down some stairs. Every time the hand moves forwards, the positive x-axis is triggered. When it moves backwards, the negative x-axis experiences g-force. When the hand moves away from the body, the positive z-axis is affected and if it moves closer to the body, the negative one is influenced. Both of them, the x- and z-axis, only experience minor changes except for some outliers which could result from quick hand movements the way they happen if one briefly needs additional balance while walking. None of them, however, happened intentionally. The y-axis is the one which is affected the most because it is the one to which changes in height are applied. Every time I take a step down, the g-force required to keep the device at the same level is reduced, as it loses height. Naturally, the bouncing behavior mentioned beforehand when analyzing the CortriumC3 accelerometer data also influences the sensors on my hand. This is how the negative y-axis values become more than 1 and even 1.5 at some points.

At the bottom of the stairs, we can observe an interesting change of values which was not present, at least not in a significant way, for the CortriumC3 device, since I turned around in place and not into a certain direction. This movement could be analyzed using a gyroscope. However, the CortriumC3 device does not support this kind of sensor. For my left hand, however, this is different. The y-axis representing the height does not show major value changes. The z-axis indicates that my left arm is slightly moving away from the body during the turn-around. The x-axis is the one which is affected the most. At first, it shows the backward movement of my hand resulting in a negative g-force value before it switches to a positive g-force value on the x-axis because of the back-swing of my left arm. It needs to be noticed that the back-swing seems to have a greater impact than the actual turn-around. However, this is due to the fact that the turn-around starts at a positive value of about 0.34, not at zero, and almost reaches -1 at its lowest point. However, this is hardly visible on such a minified version of the diagram. Regarding the second break, no mentionable value changes can be observed and none are expected.

The ascent process, on the other hand, is very interesting again. During its first few seconds, it seems similar to the descent process. However, as I stretch out my arms to use the handrails for a while, all values are greatly influenced. The y-axis veers towards 0 because it now points almost horizontally to the side instead of up and down. Therefore, there is also no g-force applied to this axis to keep it in position, as the hand rails prevent movements in that direction. The same is true for the x-axis, since the swinging movements of my arms are limited as well. Nonetheless, the direction of the x-axis does not change its orientation. Consequently, as the values of the x-axis as well as y-axis move towards 0, the values of the z-axis move up to around 1, because they now indicate height instead of movements to the left and right. As I bring my arms back into the original position, the values also go back to as they were before. The third and last break shows no surprises.
In Figure 48, the accelerometer data for my right hand is visualized. Again, the device was positioned on the back of my hand and, also as before, the negative y-axis is the one which is close to -1 in the beginning but the other two are between -0.5 and 0.5 this time. Since the values for the right hand’s y-axis closely correlate with the ones from the left hand and because the y-axis is not influenced by the fact that the device is on my right hand compared to my left one before, the y-axis does not require further analysis in this case. The same is true for the z-axis. However, it is mentionable that the values are even closer to -1 this time which means that the hand is held in a slightly different position which causes the y-axis to be even more vertically aligned. As before, the back of my hand slightly points to the front (negative x-axis) and to the side (positive z-axis) in the beginning. This time, however, the positive x-axis is triggered when I move my hand and arm to the back, while the negative one is related to forward movement. This causes the x-axis of my right hand to show negative values as opposed to the one of my left hand and the graph basically illustrates mirrored values. Regarding the rest of the recording, the right hand does not show any mentionable differences relating to what was described before except for the inversed interpretation of the x-axis and some outliers, e.g., major amplitudes for all three axes before the final break.
The next diagram (Figure 49), depicts the accelerometer data for my left leg. The sensor device was placed on the outer side on my left ankle. As before, the negative y-axis points to the ceiling, while the positive one points to the floor. The positive z-axis is directed towards my left side and the negative one points to the ankle of my right leg. The x-axis reflects changes connected to forward (positive) and backward (negative) movements. Since the sensor was almost perfectly vertically aligned, the y-axis shows a value of -1 because in order to resist the force of gravity, an opposite and compensating force along the negative y-axis is required. My left leg’s y-axis values are fairly similar to the ones of my hands. This is only logical, as the height difference covered is the same. Naturally, there are also no abrupt value changes when walking upstairs because the use of the hand rails does not affect my legs. The amplitude range, however, is generally higher because I actively lift and lower my feet when ascending and descending which is not necessary for my arms and hands. It is also visible that the lowering process of my foot is less significant when walking upstairs compared to descending. As a matter of course, the foot lifting movement is more significant when ascending. During ascending and descending, the average value of the z-axis stays around 0 but experiences major shifts to the negative part of the axis, i.e., to the right side, when descending. This might be related to me walking exceptionally slow and therefore not completely straight. However, the values seem off in this case, as I cannot explain such high peaks. Moreover, they do not occur when ascending. Neither the y-axis nor the z-axis shows unexpected values when turning around at the bottom of the stairs. In contrast to the accelerometer values of my hand, there is hardly any visible indicator for the turnaround regarding the x-axis. A possible explanation could be a smaller turn radius. The x-axis is also the most interesting one regarding the indication of forward and backward movements. When descending, the positive x-axis is clearly and visibly affected, and it is important to notice the small backward movements as well which occur because one usually starts with a bigger step than what is necessary to reach the next stair in order to be sure to get over the ledge but then has to pull back to step onto the next stair in the best possible way, i.e., without getting to far ahead which would result in a loss of
balance. Moreover, the fact that ascending steps contain less unnecessary movements, i.e., they are more precise regarding forward and backward movement compared to initially stepping too far as described before, is also unambiguously reflected in the positive x-axis values. Shortly before the final break, which does not show any unexpected results, there is one high x-axis value which most likely indicates a regular step forward at the top of the stairs with some undefined movements before the recording stops. This behavior can also be found in and confirmed by the charts of the other sensors.

The sensor device for the right leg was placed on the outer side on my right ankle (Figure 50). The x-axis and y-axis stay the same as before, due to the way the device was placed on the ankle. This means that the negative y-axis points to the ceiling, while the positive one points to the floor. The x-axis reflects changes connected to forward (positive) and backward (negative) movements. However, this time the negative z-axis is directed towards my right side and the positive one points to the ankle of my left leg.

As the sensor was not perfectly vertically aligned this time, the y-axis shows a value which is slightly higher than -1 and, consequently, the other two axes are not perfectly aligned at 0 as before. In general, the recorded data shows comparable results. For descending, the positive x-axis shows higher average values and the negative one shows lower average values. For ascending, the positive values show a lower average compared to the left leg. This can be explained by a different movement behavior of my legs which seems plausible for usual walking, as it is unlikely that both legs are moved in the exact same way. One would need to conduct additional recordings to analyze the walking behavior of other people which is not part of or necessary for this thesis.

The z-axis does not show opposite values for my right leg compared to my left one because of the changed axis direction and because I seem to step marginally to the left and to the right, respectively, when descending. For the ascending process, the same behavior although much less significant can be observed. The y-axis does not indicate any mentionable differences.
4.2.6. **MetaWearCPro barometer data**

Usually, the barometric pressure drops with ascending and rises with descending. Here (Figure 51), it is the other way around and an explanation needs to be found. It is important to mention that the recorded data is correct, as other recordings – for the SENEX project – show anticipated values related to a change in altitude. This leads to the conclusion that the environmental circumstances differed in order to produce such results. As I do not know of any ventilation
system that is installed in that building which could significantly contribute to this monitored anomaly, another cause must be found.

First of all, the outside temperature needs to be determined to gain additional insight. I did not record the temperature inside the building, thus, I rely on the weather archive to define an approximate outside temperature. According to [16], it was about ten degrees Celsius. This, in turn, means that the heating in the building was definitely on and since warm air rises and cold air was coming in from the revolving door on the ground floor, i.e., close to the bottom of the stairs, a significant difference in temperature must have been present.

In accordance with the barometric formula and the assumption that the stairs are about four to five meters in height, it would need a minimum temperature offset of two degrees Celsius/Kelvin per meter to produce shown results. This is a potential explanation but whether it is fully plausible cannot be guaranteed, as a temperature difference of eight to ten degrees for a few meters even under special circumstances is very high. Nevertheless, it provides a supposition.

As explained and concluded, the barometric pressure starts at its highest value in the beginning of the recording. The first and last few seconds indicate that no steps up and down are made which is true, as I did not immediately descend the stairs with my first steps, but I did make some additional steps in the end when I reached the top again. With every step down, the temperature falls, and the pressure level drops accordingly. As I reach the lowest point, the air pressure settles at an average of about 99760 Pa. At the point where I turn around, it slightly drops again. A potential reason for that could be a minor temperature change due to the revolving door being moved but since my left arm moved away from it, this is rather unlikely. It could also relate to a measurement inaccuracy. As I walk up again, the temperature and pressure level rise and there is even an indication of me putting my left arm up to grasp the hand rail at about 00:46. At the top of the stairs, the barometric pressure shows almost the same values as in the beginning which seems very plausible.

4.2.7. MetaWearCPro gyroscope data

The last data charts I want to explain show the recorded gyroscope data of the MetaWearCPro devices on my right hand (Figure 52) and left ankle (Figure 53). Again, there are three axes. The x-axis represents the roll axis. The positive values relate to a left and the negative ones to a right roll. The y-axis equals the yaw axis. Here, a counter-clockwise turn around a virtual vertical line is indicated through negative values and a clockwise turn through positive ones. The pitch axis is represented by the z-axis. A pitch up is shown through positive values and a pitch down through negative ones.

The x-axis, i.e., roll movements to the side stay very quiet for the descending part. The turnaround at the bottom of the stairs shows a minor pitch to the left (positive x-axis), as I turned my body and especially my right arm and hand to the left and slightly outwards. There are no surprises in the intermediate break as well as for the first ascending part. This is because most of my hand movements while climbing stairs result in pitching. However, there is clear evidence of the moment I move my right hand to the hand rail at about 00:46. My hand moves up and to the right resulting in a left roll on the graph. Now, the x-axis still represents roll. However, the roll movements in this situation are more intense than before and, therefore, there is much more activity for this period of time. The amplitude at around 00:55 shows the movement I remove my hand from the rail and back to the right side of my body. The last high negative value shows unintentional movements at the top of the stairs as I finished my test and was about to stop the
recording. Thus, it is not important.
The y-axis, which represents yaw behavior, does not show any outstanding values until the turnaround at the bottom of the stairs which apparently caused an anti-clockwise yaw of my right hand. As I put my hand up to the hand rail, the y-axis and the z-axis switch places for the time my hand is up which means that the y-axis now indicates pitching. Since my hand drastically reduces balancing movements when using a hand rail, the amplitudes merely show a minor increase for the y-axis. The moment I remove my hand from the hand rail, a pitch to the front can be observed – as I am still moving forward while lowering my arm – before the value normalizes again. The outlier at the end can be ignored.
The third and last data stream to analyze is the z-axis representing pitching behavior. When descending, my right arm swings the most which can also be seen in the depicted graph. As the measurement unit is given in degrees per second, it is evident that my right arm takes more time for pitching up (rather low positive values) than down (higher negative values) when walking downstairs. This is dependent on the general walking style, speed, and arm position. Again, the turnaround, i.e., a slight front pitch, is clearly visible. When descending, the amplitudes are much smaller compared to the first part. However, for the hand rails part, the z-axis becomes the y-axis, i.e., yaw, for that period of time. Here, the values merely show standard behavior, as the hand is guided by the hand rail and can hardly yaw. Afterwards, no mentionable behavior was recorded.
Overall, it needs to be pointed out that the yaw axis values seem a bit too high for the movements I made but still relevant in terms of significance and expressiveness.

![Figure 52 - MetaWearCPro gyroscope data right hand](image)

The final chart in Figure 53 shows the gyroscope data of my left ankle. As before, the x-axis represents the roll axis. The positive values relate to a right and the negative ones to a left roll. The y-axis equals the yaw axis. Here, a counter-clockwise turn around a virtual vertical line is indicated through negative values and a clockwise turn through positive ones. The pitch axis is represented by the z-axis. A pitch up is shown through negative values and a pitch down through positive ones.
The roll axis shows many left rolls for which I do not have a proper explanation. However, it is most likely related to my personal descending behavior, although it still seems exaggerated. Furthermore, the values are given in degrees per second which means they got scaled to one second and can, consequently, show higher values than expected. Naturally, this is true for all gyroscope data. The right roll relating to the turnaround at the bottom of the stairs is apparent. During ascending, there are many rolls to the left as well as to the right for which I, as before, can only provide the same explanation. The yaw axis is very quiet regarding the descent and until the turnaround at the bottom of the stairs which is unequivocally demonstrated by the very high negative values. Apparently, the yaw axis is more active when ascending but since the hand rail does not affect the leg movement, nothing mentionable requires attention. The outliers at the end are unintentional and can be disregarded.

The pitch or z-axis shows roughly equal values for pitching up as well as down which seems plausible for descending. The turnaround at the bottom of the stairs is hardly visible because my left leg solely turns and does not require additional positioning. When ascending, my foot and ankle pitch up and down, but the down-pitches are carried out faster indicated by a higher absolute degree per second value. As always, the final break concludes the recording.

![Figure 53 - MetaWearCPro gyroscope data left ankle](image)

### 4.2.8. Combined analysis

Regarding the previous descriptions, I already bore in mind that I was climbing stairs and, therefore, I was able to analyze the charts individually. However, what if the recorded person had not been me and if I did not know what the recorded data is about in advance? For this purpose, I am going for a combined approach of above-shown diagrams. This means that I will pick out the same specific section of each diagram one at a time in the same order they are presented above – except for the barometer data which is presented last – and add up the contained information to complete the overall picture and to find out what the test subject, I, was doing. However, I postulate the knowledge of the sensor device types and their positions on the body including all axes directions, otherwise a meaningful interpretation of the recorded data is
not feasible. Below, the resulting chart sequence is shown which focuses on ascending the stairs between around 30 and 45 seconds after the beginning of the recording.

Judging from the body temperature in Figure 54, no specific information can be obtained except that the temperature marginally but insignificantly rises. However, it might indicate some body movement. The same is true for the device temperature chart (Figure 55) but here it is hard to determine whether the very slight rise in temperature is caused by the running device or by the body itself.

The added respiration information in Figure 56 does not contribute to a clearer picture either, as it does not show significant fluctuations. Figure 57 adds the first interesting insights. The data shows hardly any movements in the beginning but starts to indicate actions at around 0:40. As a reminder, the positive x-axis points upwards, the positive y-axis to the right, the positive z-axis to the front. This information is sufficient to show that the entire body performs significant up and down movements but only minor ones to the left and right as well as to the front and back. This could indicate some sort of jumping or similar.
Regarding the accelerometer data of the left hand (Figure 58), the positive x-axis points to the front, the positive y-axis to the ground, and the positive z-axis away from the body and, thus, to the left side. The data shows that the left hand – and possibly arm too – is definitely moving or swinging to the front and back but only marginally to the left and right. Similar to the x-axis in the previous chart, the y-axis shows up and down movements which confirms a jumping-like action. The diagram for the right hand (Figure 59) shows very similar y-axis values, as this axis is independent from the hand to which it is attached, i.e., the positive y-axis points to the ground again. The positive x-axis is triggered when the test subject moves their arm to the back, and the positive z-axis when it is moved to the right side and away from the body. As the positive z-axis points in the opposite direction compared to the left hand, and since the values are similar, this indicates that both arms were held in an analog position, i.e., dangling loose next to the body in the beginning. The inverted x-axis confirms this supposition. Later, however, swinging-like movements can be observed judging from the two axes x and y, because the height of the hand as well as its position – not to the side – changes.

Regarding the left leg (Figure 60), the positive x-axis points to the front, the positive y-axis to the floor, and the positive z-axis towards the left side and away from the body. The y-axis value of -1 shows that the sensor device is vertically aligned in the beginning, as the negative axis direction represents the gravity counterforce. Consequently, this means the test subject is either standing or at least that their legs are in an upright position as if the person was standing, sitting, or the like. Additionally, the y-axis values of the left and right leg correspond with the ones of the left and right hand which indicates that legs and hands were moved based on some sort of rhythmic behavior which excludes any steady position but suggests sports-like and/or moving actions.
The data of the right leg (Figure 61) confirms this theory and the fact that all four affected body parts were moved rhythmically.

The gyroscope data for the right hand (Figure 62) shows minor roll (x-axis) and yaw (y-axis) movements but more significant pitch movements. All of this affirms slight arm swings similar to walking behavior and, more importantly, does not contradict any of the data analyzed so far. The left leg diagram (Figure 63), on the other hand, shows greater amplitudes. The meaning of the three axes stay the same as before. What can be read from the data is that there are major pitch and roll movements and rather minor yaw ones. The latter one indicates that the foot mostly points straight to the front. The average roll values are centered around 0 but high degrees per second values are present as well which suggests some sort of balancing actions. The same applies to the pitch axis but it is also apparent that the axes x and z show similar and rhythmic amplitudes. Consequently, the foot is rhythmically pitched up and down as well as rolled to the left and right. Together with the accelerometer data of both legs which shows movements to the front and/or back, it, again, suggests walking behavior and rather forwards than backwards.

The final information added through the barometer data (Figure 64) of the test subject’s left hand shows increasing values which under normal circumstances is a sign of a loss in altitude. This can be caused by elevators, escalators, stairs, or else. Since we have additional and analyzed data above, we know that the loss in altitude is combined with forward walking behavior. Usually, this can only refer to a person walking down a staircase.

As a concluding remark for this combined analysis section, I need to mention that it is neither practical nor reliable to quickly determine what a person was doing by picking out data sections
the way I did simply because the solution is likely to be ambiguous and unreliable. Moreover, I would need additional sample data of another test subject without knowing what that person did to perform an unbiased analysis. Also, the resultant finding is incorrect because of the abnormal barometer recording. It would require more knowledge of the test case to correctly interpret those values.

It is also important to keep in mind that, if applicable, all three axes need to be read together because as the values of one axis changes, at least one other axis is affected as well due to the way a three axes system works. This also means that an axis could initially show movements to the front and back but then to the top and ground if changed by 90 degrees.

Nevertheless, such a swift inspection gives a first impression and valuable insight even if it demands further analysis and a closer look. For more professional and more viable findings, it is probably best to use or write a piece of software to support the data interpretation especially because the moving axes can easily lead to confusion and misinterpretation.
5. Chapter 5 – Discussion, Conclusion, and Future Work

This is the last chapter of my thesis which deals with a selected range of above-mentioned topics. The focus here lies on my personal opinion as well as conclusions which I can draw from what I have written so far. The last sub-section of this chapter addresses future work and suggestions to improve or expand the presented framework.

5.1. Discussion and Conclusion

After describing some related work aspects including OSGi and IoT at the beginning of this thesis, analyzing our own framework in depth, and conducting my own test scenario, I want to briefly discuss the most important topics of this paper once more to draw appropriate conclusions. To start with, I delineate some positive as well as negative facets of our framework. The positive ones enfold the reduced complexity because of the way our modules and classes are organized. Hand in hand with this go its reusability and generic nature. All of them are especially important to make it simple to add new sensor devices to the framework.

Furthermore, they are also listed, amongst others, as main advantages of OSGi on their official website [2]. Another advantage is that only partial framework knowledge is required to add another device, as most components do not need to be touched. One potential drawback, however, is the missing foundation regarding additional major features. Small adaptions and modifications are most likely no problem and our framework is fairly flexible in this respect, as long as they do not restrict the connected sensor devices in any way which could limit their application area or functionality. Adding an additional sensor value data type, for instance, would not require many changes. On the contrary, adding IoT components such as cloud services or allowing the sensors or sensor devices to communicate with each other would necessitate significant refactoring work. Naturally, this depends on the fact whether such changes are desired in the first place. IoT in conjunction with future development of our framework is addressed in the next chapter.

Further limitations include the inherent Android focus which renders it impossible to run it anywhere else than on Android devices. Fundamental changes are necessary to change this fact. Depending on the specific Android version, the maximum number of allowed Bluetooth connections varies which consequently limits the framework’s field of application. Moreover, data transmission problems, lags, or interferences may occur and there is hardly any way to obviate them for good. For recording, it is advisable to choose an environment with as little signal interference as possible. To support more Bluetooth devices, external Bluetooth adapters can be used. The more devices, the faster the battery of the recording device gets drained which is why supplementary portable battery packs might be necessary.

Unfortunately, all our sensor devices use proprietary data protocols which is why each of them needs its own library unless it is of the exact same type. Our framework does not support any official protocol to which standardized sensor devices could connect which means that a respective external library is mandatory in any case. Although the calibration aspect has been neglected in my thesis, it is an important aspect to gain reference values and know the exact initial sensor position. Some devices may need special attention at this point which is also why all device classes must implement this method on their own.

Not all sensor devices support time stamps together with their event callbacks. In such cases, only the time stamps added by the recording device are available. In general, this is not a major
issue because the data packages are very small and, thus, the transfer rate via Bluetooth very fast. However, a transmission delay may result in distorted data logs. Such a problem including its implications is close to impossible to identify or rectify in retrospect.

Reasons why we did not use or need to follow the IoT paradigm have already been stated. Reasons for not using the OSGi framework have been given in the second chapter as well. Since we worked with proprietary protocols, it would not have saved us much time or effort to use OSGi. Although OSGi can be used in connection with Android, for future developments, it might be required to keep all the code private instead of using open-source software or frameworks. Additionally, we have very high performance requirements and, therefore, we need to have as much control over the application and its code as possible.

We could have chosen a Windows device so the AsTeRICS suite would run on it but for some of our sensor devices, only Android libraries were supported. iOS was not relevant either way due to said library reason and because AsTeRICS would also not work with iOS. And again, code privacy played an important part in our decisions. What is more, our individual solution offers flexible and hypothetically unlimited customization.

Naturally, our framework is not perfect and offers room for improvements, e.g., some driver components which are required by the SENEX recorder could be moved to the core module instead. We also experienced several challenging situations, such as connecting that many Bluetooth devices all at once, saving battery life as well as possible, and implementing all device classes properly, as the manufacturers’ documentations were not always thorough enough. Another major problem we had to deal with were connection issues and unexpected disconnects while recording which were mostly solved through automated reconnect attempts and manual reconnect options. However, we achieved our goal in creating a generic sensor framework for parallel and continuous data processing. Together with adequate analysis tools, all desired information can be extracted from the recorder’s data logs as exemplified in my test chapter.

At the end of this chapter I would like to add some personal thoughts regarding the entire SENEX project. As the project’s purpose is very innovative, it was intriguing to be part of it. Although my part in this project is over, it has not been completed at this point, as not enough patients could be found so far and because it is a rare occasion that a patient would experience a moment of confusion while recording. Nevertheless, the project is going well.

Some human aspects which are worth mentioning include that the physical sensors attached to the patient must be as unobtrusive as possible, so the patient does not feel foolish and behaves naturally; also because the entire sensor construct may look a bit strange for outsiders. Judging from my personal experience during my experiment, I can fully relate why a patient may feel that way. Especially the vest including the external battery pack could be improved to lower its impact on the user’s movements and general feeling. All in all, the SENEX project was a valuable experience for a good cause.

5.2. Future Work

This very last sub-section should provide a brief glimpse into the future regarding our developed framework. It is not about planned steps which are going to be taken but about personal suggestions and ideas, so more people could benefit from our solution.

What can be improved implementation-wise is the stability of the recording device running the
main application because due to the heavy load of many Bluetooth devices, connection failures can randomly occur. Whether this problem can be conquered by future Android versions or else is up to the prospective developers. Maybe it would also be smart to try OSGi for Android and find out whether the framework could benefit from it. Another technical feature could support standard protocols for data transfer between devices which support them. Speaking of devices, it would greatly improve the framework’s potential for various fields of application if the focus was shifted from sensor devices in the SENEX context to all kinds of different devices. This would result in an even more generic application plus make it more interesting for everyday life situations. In this regard, I am certain that the IoT approach would prove itself useful as well.

Data collected by various (sensor) devices could be collected, sent to, and processed in the cloud, and trigger smart follow-up decisions and actions which could significantly affect the user’s life or specific situation in a positive way. For the purpose of SENEX’s cause, the information in the cloud could notify external services, call family members, or report back to the user offering guidance. I already talked about this idea above, so, to add another facet, it would be interesting to connect the sensor network with an external user right away who would then be able to monitor the individual in a way that machines cannot while being provided with valuable additional information from the cloud. This way, IoT and human interaction or surveillance would be combined.
### Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Abstract problem</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>OSGi layers (taken from [5])</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>OSGi layer interaction (taken from [3])</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>Abstract OSGi framework</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>OSGi comparison with our framework</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>AsTeRICS example</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>IoT concept (taken from [10])</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>IoT example</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>Abstract solution</td>
<td>18</td>
</tr>
<tr>
<td>10</td>
<td>Module dependencies</td>
<td>19</td>
</tr>
<tr>
<td>11</td>
<td>SENEX-Recorder intra-module dependencies</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>SENEX-Recorder application</td>
<td>21</td>
</tr>
<tr>
<td>13</td>
<td>SENX-Remote-Control</td>
<td>23</td>
</tr>
<tr>
<td>14</td>
<td>Core intra-module dependencies</td>
<td>23</td>
</tr>
<tr>
<td>15</td>
<td>Driver intra-module dependencies</td>
<td>25</td>
</tr>
<tr>
<td>16</td>
<td>Application start and setup process</td>
<td>33</td>
</tr>
<tr>
<td>17</td>
<td>Connection of devices</td>
<td>34</td>
</tr>
<tr>
<td>18</td>
<td>Sensor device state change</td>
<td>35</td>
</tr>
<tr>
<td>19</td>
<td>Sensor value change</td>
<td>35</td>
</tr>
<tr>
<td>20</td>
<td>Digital representation of the CortriumC3 device (taken from [11])</td>
<td>36</td>
</tr>
<tr>
<td>21</td>
<td>Digital representation of the MetaWearCPro device (taken from [14])</td>
<td>37</td>
</tr>
<tr>
<td>22</td>
<td>Config file overwriting process</td>
<td>46</td>
</tr>
<tr>
<td>23</td>
<td>Application start-up</td>
<td>48</td>
</tr>
<tr>
<td>24</td>
<td>Initialization A</td>
<td>50</td>
</tr>
<tr>
<td>25</td>
<td>Initialization B</td>
<td>51</td>
</tr>
<tr>
<td>26</td>
<td>Initialization C</td>
<td>51</td>
</tr>
<tr>
<td>27</td>
<td>Initialization D</td>
<td>52</td>
</tr>
<tr>
<td>28</td>
<td>Initialization F</td>
<td>52</td>
</tr>
<tr>
<td>29</td>
<td>Initialization E</td>
<td>53</td>
</tr>
<tr>
<td>30</td>
<td>Initialization G</td>
<td>54</td>
</tr>
<tr>
<td>31</td>
<td>Initialization H</td>
<td>54</td>
</tr>
<tr>
<td>32</td>
<td>Initialization overview</td>
<td>55</td>
</tr>
<tr>
<td>33</td>
<td>Connection process A</td>
<td>56</td>
</tr>
<tr>
<td>34</td>
<td>Connection process B</td>
<td>57</td>
</tr>
<tr>
<td>35</td>
<td>Connection process C</td>
<td>58</td>
</tr>
<tr>
<td>36</td>
<td>Connection process D</td>
<td>59</td>
</tr>
<tr>
<td>37</td>
<td>Connection process overview</td>
<td>60</td>
</tr>
<tr>
<td>38</td>
<td>Data logging A</td>
<td>62</td>
</tr>
<tr>
<td>39</td>
<td>Data logging B</td>
<td>63</td>
</tr>
<tr>
<td>40</td>
<td>Data logging overview</td>
<td>64</td>
</tr>
<tr>
<td>41</td>
<td>Test scenario staircase</td>
<td>66</td>
</tr>
<tr>
<td>42</td>
<td>Test scenario setup</td>
<td>67</td>
</tr>
<tr>
<td>43</td>
<td>CortriumC3 body temperature</td>
<td>69</td>
</tr>
<tr>
<td>44</td>
<td>CortriumC3 device temperature</td>
<td>70</td>
</tr>
<tr>
<td>45</td>
<td>CortriumC3 respiratory rate</td>
<td>71</td>
</tr>
<tr>
<td>46</td>
<td>CortriumC3 accelerometer data</td>
<td>72</td>
</tr>
</tbody>
</table>
Figure 47 - MetaWearCPro accelerometer data left hand ..................................................74
Figure 48 - MetaWearCPro accelerometer data right hand ..............................................75
Figure 49 - MetaWearCPro accelerometer data left ankle ..............................................76
Figure 50 - MetaWearCPro accelerometer data right ankle ...........................................77
Figure 51 - MetaWearCPro barometer data left hand .....................................................77
Figure 52 - MetaWearCPro gyroscope data right hand ...................................................79
Figure 53 - MetaWearCPro gyroscope data left ankle ....................................................80
Figure 54 - CortriumC3 body temperature extract ..........................................................81
Figure 55 - CortriumC3 device temperature extract .......................................................81
Figure 56 - CortriumC3 respiratory rate ........................................................................81
Figure 57 - CortriumC3 accelerometer data .................................................................81
Figure 58 - MetaWearCPro accelerometer data left hand extract ....................................82
Figure 59 - MetaWearCPro accelerometer data right hand extract ..................................82
Figure 60 - MetaWearCPro accelerometer data left ankle extract ..................................82
Figure 61 - MetaWearCPro accelerometer data right ankle extract ..................................82
Figure 62 - MetaWearCPro gyroscope data right hand extract .......................................83
Figure 63 - MetaWearCPro gyroscope data left ankle extract .........................................83
Figure 64 - MetaWearCPro barometer data left hand extract .........................................83
Table of Tables

Table 1 - Diagram elements explanation.................................................................47
References


