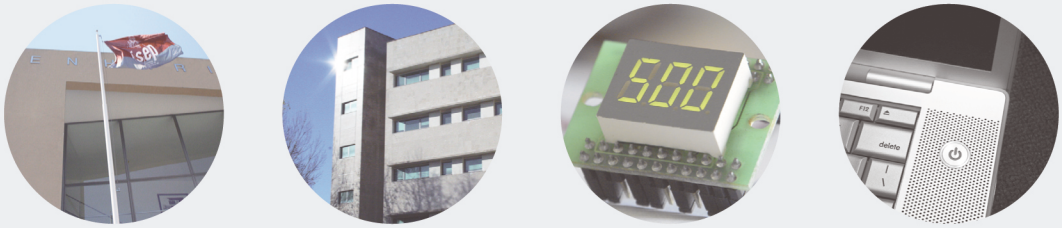
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES



Integrating RFID POS in Hardware-in-the-Loop Simulations

NUNO ALVES MARTINS PENA DA SILVA
novembro de 2024

POLITÉCNICO
DO PORTO

Politécnico do Porto
Instituto Superior de Engenharia do Porto

Integrating RFID POS in Hardware-in-the-Loop Simulations

Nuno Silva

Master in Electrical and Computer Engineering
Specialization Area of Automation and Systems

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto

November, 2024

This dissertation partially satisfies the requirements of the Thesis/Dissertation course of the program Master in Electrical and Computer Engineering, Specialization Area of Automation and Systems.

Candidate: Nuno Silva, No. 1151316, 1151316@isep.ipp.pt

Scientific Guidance: Paula Viana, pmv@isep.ipp.pt

Scientific Co-Guidance: Vítor Cunha, vrc@isep.ipp.pt

Company: Sensormatic

Advisor: Gustavo Oliveira, gustavo.oliveira@jci.com

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Departamento de Engenharia Eletrotécnica

Instituto Superior de Engenharia do Porto

Rua Dr. António Bernardino de Almeida, 431, 4200–072 Porto

November, 2024

Acknowledgements

Gostaria de expressar a minha sincera gratidão à equipa de Firmware de Portugal da Sensormatic. Um agradecimento especial ao Miguel Vieira, que me acompanhou e apoiou ao longo de todo o meu projecto de estágio, e ao Gustavo Oliveira, cuja contribuição no desenho e na conceptualização do projecto foi fundamental.

Queria também agradecer aos meus orientadores do ISEP, Paula Viana e Vítor Cunha, cujo apoio foi essencial na correta estruturação e revisão deste documento, bem como pela forte recomendação do uso da ferramenta LaTeX para a elaboração do projeto, sendo o coorientador Vítor Cunha o criador do template utilizado. Ao longo do desenvolvimento da tese, deu para perceber a ajuda enorme que foi.

Por fim, gostaria de agradecer à minha família, por serem pessoas especiais, que sempre estiveram ao meu lado, apoiando-me em tudo o que precisei ao longo da minha vida; aos amigos, pelos momentos bem passados; e à minha namorada, por todo o apoio que tenho tido ao longo deste desafiante percurso.

Obrigado a todos.

Abstract

In the field of retail stores technology, Point of Sale systems equipped with Radio Frequency Identification readers, are often used to process transactions. Typically, the firmware development for these systems involves multiple testing phases in order to approve a new firmware release. Moreover, testing can be challenging for the development and quality assurance teams if there are third-party software dependencies. In this case the closed nature of the third-party dependencies restricts and/or delays the new firmware release cycles.

To address these challenges the development of a new tool named TestFramework is proposed, which adopts the Hardware-in-the-Loop technique, to simulate the third-party software used in the point of sales firmware. The new tool incorporates a custom scripting language built for this purpose that enables black-box testing, thus, allowing the teams to design and run test scenarios without having access to the source code. This approach reduces the dependency on external systems and creates a controlled environment for both manual and automated software testing. Additionally, the use of Continuous Integration/Continuous Deployment tools, automates the testing and deployment processes by allowing constant validation of each source code change. This automation streamlines the development cycle, leading to shorter release cycles and overall enhanced software quality by enabling early problems detection.

The TestFramework tool achieved good results by improving the testing capabilities of the development and testing teams. However some limitations were identified, such as the learning curve needed to the efficient use of the custom scripting language and the possible lack of edge cases testing. Future work aims to make the tool more user friendly by developing a graphical interface that will simplify the scripts editing and allow a easier configuration of the automated testing pipeline.

Keywords: Hardware-in-the-Loop, Black-box testing, Point of Sale, Radio Frequency Identification, Continuous Integration/Continuous Deployment, Domain-specific language

Resumo

No campo da tecnologia para lojas de retalho, os sistemas de Ponto de Venda equipados com leitores de Identificação por Radiofrequência são frequentemente usados para processar transações. Tipicamente, o desenvolvimento de *firmware* para estes sistemas envolve múltiplas fases de teste para aprovar um novo lançamento de *firmware*. Além disso, os testes podem ser desafiantes para as equipas de desenvolvimento e garantia de qualidade se houver dependências de *software* de terceiros. Neste caso, a natureza fechada destas dependências restringe e/ou atrasa os ciclos de lançamento do novo *firmware*.

Para abordar estes desafios, é proposto o desenvolvimento de uma nova ferramenta denominada TestFramework, que adota a técnica de *Hardware-in-the-Loop* para simular o *software* de terceiros usado no *firmware* dos pontos de venda. A nova ferramenta incorpora uma linguagem de *script* personalizada criada para este efeito, que permite testes de caixa preta, permitindo assim que as equipas concebam e executem cenários de teste sem terem acesso ao código fonte. Esta abordagem reduz a dependência de sistemas externos e cria um ambiente controlado para testes de *software* manuais e automatizados. Adicionalmente, o uso de ferramentas de Integração Contínua/Implementação Contínua automatiza os processos de teste e implementação, permitindo a validação constante de cada alteração no código fonte. Esta automatização agiliza o ciclo de desenvolvimento, conduzindo a ciclos de lançamento mais curtos e a uma melhor qualidade global do *software*, ao permitir a deteção precoce de problemas.

A ferramenta TestFramework alcançou bons resultados ao melhorar as capacidades de teste das equipas de desenvolvimento e teste. No entanto foram identificadas algumas limitações, como a curva de aprendizagem necessária para o uso eficiente da linguagem de *script* personalizada e a possível falta de testes de casos limite. O trabalho futuro visa tornar a ferramenta mais amigável, desenvolvendo uma interface gráfica que simplificará a edição de *scripts* e permitirá uma configuração mais fácil da *pipeline* de automação de testes.

Palavras-Chave: *Hardware-in-the-Loop*, Testes de Caixa-Preta, Ponto de Venda, Identificação por Radiofrequência, Integração Contínua/Implementação Contínua, Linguagem Específica de Domínio

Contents

List of Figures

List of Tables

Listings

List of Acronyms

API	Application Programming Interface
ARM	Advanced RISC Machines
AST	Abstract Syntax Tree
CI/CD	Continuous Integration/Continuous Deployment
CPU	Central Processing Unit
DEE	*Departamento de Engenharia Electrotécnica*
DOME	Distributed Object Model Environment
EAN	European Article Number
EAS	Electronic Article Surveillance
EPC	Electronic Product Code
FOV	Field of View
GTIN	Global Trade Item Number
GUI	Graphical User Interface
H-HIL	Hybrid Hardware-in-the-Loop
HF	High Frequency
HID	Human Interface Device
HIL	Hardware-in-the-Loop
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IoT	Internet of Things
IP	Internet Protocol
ISEP	*Instituto Superior de Engenharia do Porto*

JSON	JavaScript Object Notation
LED	Light-Emitting Diode
LF	Low Frequency
LLRP	Low Level Reader Protocol
LTL	Linear Temporal Logic
MEEC-AS	*Mestrado em Engenharia Electrotécnica e Computadores - Automação e Sistemas*
MQTT	Message Queuing Telemetry Transport
POE	Power Over Internet
POS	Point of Sale
QA	Quality Assurance
RCS	Radar Cross Section
REST	Representational State Transfer
RFID	Radio Frequency Identification
ROS	Robot Operating System
SGTIN	Serialised Global Trade Item Number
SIL	Software-in-the-Loop
SOAP	Simple Object Access Protocol
SSH	Secure Shell
TCP/IP	Transmission Control Protocol/Internet Protocol
TEDI	*Tese/Dissertação*
UDP	User Datagram Protocol
UHF	Ultra High Frequency
UPC	Universal Product Code
URI	Uniform Resource Identifier
USB	Universal Serial Bus
XML	eXtensible Markup Language

List of Symbols

Symbol	Description	Units
c	Speed of light	$\mathrm{m\,s^{-1}}$
f	Frequency of the waves	Hz
G_r	Gain of the receiving antenna	-
G_t	Gain of the transmitting antenna	-
λ	Wavelength of the transmitted waves	m
P_r	Power received by the tag from the reader's transmission	W
P_t	Power transmitted by the reader	W
R	Distance between the reader and the tag	m
σ	Radar cross-section of the tag	$\mathrm{m^2}$

Chapter 1

Introduction

This document describes the work carried out within the scope of the course unit of *Tese/Dissertação* (TEDI), in the 2nd year of *Mestrado em Engenharia Electrotécnica e Computadores - Automação e Sistemas* (MEEC-AS), from the *Departamento de Engenharia Electrotécnica* (DEE), at the *Instituto Superior de Engenharia do Porto* (ISEP).

The main focus of this thesis is to achieve the development and integration of a Hardware-in-the-Loop (HIL) system for a Point of Sale (POS) with a Radio Frequency Identification (RFID) reader. The proposed system is intended to simulate a POS in a physical retail store. The HIL concept makes it possible to test in a simulation environment all the components used in an actual store, including the simulation of proprietary firmware from a third-party supplier related to the RFID tag reading hardware. Therefore, the proposed system will avoid the need to use the third-party company data to test the whole system. With this method, it is expected that the POS system can be tested and validated in a controlled environment.

In today's typical software development cycles, continuous testing plays a very important role in ensuring software quality and performance. This project aims to integrate both Continuous Integration/Continuous Deployment (CI/CD) and manual testing tools to streamline the testing and deployment processes of new POS firmware releases.

The development of this project will allow the Quality Assurance (QA) teams to easily simulate POS transactions and run black-box tests to identify errors, without requiring knowledge of the underlying source code. Moreover, it is anticipated that

this approach will enable the simulation of real-world scenarios (simulation testing) and also the validation of the system behavior and features (functional testing). This methodology offers the opportunity to improve the creation of new tests, potentially allowing for automation of repetitive testing tasks, making the entire testing process more efficient for both the development and QA teams.

1.1 Context

This project was proposed by the Portuguese firmware department of the Sensormatic company, that specializes in Electronic Article Surveillance (EAS) and other retail security solutions. Founded in 1966, Sensormatic became a major player in loss prevention technologies for retailers, offering systems that help businesses reduce theft, improve inventory management, and enhance store operations. The focus of the Portugal firmware team, is to maintain and develop new firmware for EAS related devices [1].

The systems that are used for sales in retail stores must be reliable. Issues in the POS device that directly influence sales or the correct operational workflow in a store have big impact for the Sensormatic clients. Therefore, they must be avoided in order to protect Sensormatic's reputation and not directly affect sales.

Having good developers is not enough to prevent these issues from happening, since even the best developers make mistakes. To prevent bugs from reaching production, there's a QA team deployed to try to minimize the chances of issues happening. This team will be in charge of ensuring that the code produced by the development team meets the necessary standards for functionality, performance, and stability before it is released.

Even if currently the majority of the issues are found by the QA team, some are still missed by the QA testing pipeline and manual tests. This project aims to improve this scenario in order to decrease the number of issues that still escapes their scrutiny.

1.2 Problem Definition

The primary challenge addressed in this thesis is the inability to directly modify the proprietary firmware of the POS system which controls the RFID chip and therefore the configuration of the system. This firmware code is maintained by the outside company Keonn, which owns the hardware and the firmware [2]. As a result, Sensormatic developers and testers are constrained to work with their own software component of the system. To overcome this, the HIL system will simulate the behavior of the proprietary firmware, allowing Sensormatic teams to perform

comprehensive testing without having to rely on the third-party code, while still running the onsite developed code on the hardware.

This method can also be helpful to reduce the time needed for a new firmware release. Every time the development team makes a release it needs to wait for Keonn to also update their own code, when needed. Due to the team's need to wait for Keonn's new code before it can begin testing a release candidate, or even developing a future release, this significantly delays the software development cycles.

1.2.1 Objectives

Given the stated problem definition, the key objectives of this thesis are defined as follows:

- Develop a Hardware-in-the-Loop (HIL) system for POS testing that simulates RFID tag reading and transaction processing.
- Develop a custom scripting language that will enable black-box testing.
- Integrate CI/CD tools to automate testing and deployment processes using the custom scripting language.
- Ensure that the simulation is as close as possible to the actual proprietary system, providing reliable and accurate test results.
- Provide a flexible testing framework that can be easily adapted to future POS updates requirements.

1.2.2 Expected Results

By the end of this project, the expected outcomes include:

- A fully functional HIL system capable of simulating the POS.
- A custom scripting language that will enable black-box testing, which can be manually executed or integrated into a testing pipeline.
- An automated testing pipeline using CI/CD tools, which enables continuous testing and rapid feedback during the development process.
- Improved software quality due to thorough testing under various scenarios, leading to a more robust and reliable POS system.

1.3 Work Plan

The project work plan is divided into several phases, namely:

1. **Research and Literature Review:** Survey the state of the art methods in RFID technology, HIL systems, black-box testing, CI/CD tools, Application Programming Interface (API) Go and Message Queuing Telemetry Transport (MQTT).

2. **Research the Keonn API:** Before starting development for the Keonn API simulator, a research phase is needed.

3. **Development of HIL System:** Designing and implementing the simulator for the POS system.

4. **Development of the custom scripting language** This will be the base for all the tests, where each test is validated as black-box testing.

5. **Integration of CI/CD Tools:** Configuring the testing framework to work with CI/CD platforms such as Jenkins or GitLab CI.

6. **Testing and Validation:** Testing of the HIL system to ensure it accurately simulates the data and provides reliable results.

7. **Documentation and Results Analysis:** Documenting the entire process, analyzing test results, and identifying areas for improvement.

1.4 Dissertation Organization

This document is organized in six chapters. The first one, Chapter 1, provides the context, problem definition, objectives, and work plan of the whole project. It follows the chapter entitled "State of the Art" that discusses RFID technology, the principles of Hardware-in-the-Loop testing, black-box simulation, the role of CI/CD in modern development and the use for APIs. Additionally, its also used for an introduction to the programming language Go and an overview to the MQTT protocol with a Mosquitto broker. The third chapter details the hardware and software components needed to achieve the proposed goals. Chapter 4, is used to explain the architecture of the TestFramework tool, the development of the HIL system, the integration with CI/CD tools and the black-box testing. The following chapter, number five, presents the results of the testing, including both manual and automated tests. The key findings of this project and the suggested future development directions are presented in the 6^{th} and final chapter.

Chapter 2

State of the Art

This chapter will focus on the literature survey required for evaluating and testing the features and behaviour of an RFID based POS reader. Firstly, a research about the fundamentals and principles of RFID technology is outlined. Next, the HIL testing methodology is explored, since it represents a state of the art approach to validate the performance of the POS system in a simulated environment. A survey about black-box simulation, which enables to test the system behaviors without requiring internal knowledge of the system structure, is then presented. Next, it is addressed the importance of adopting CI/CD pipelines in order to facilitate automated testing and deployment, with a focus on tools such as Jenkins and GitLab CI. The role of a API is also reviewed to enable smooth communication between software modules, highlighting JavaScript Object Notation (JSON) and eXtensible Markup Language (XML) formats and effective API design standards. The use of Go in API development and testing is discussed, including its strengths in cross platform development and black-box testing frameworks. And finally, an introduction to the MQTT protocol, with the Mosquito broker.

The knowledge gained from this different topics will provide the necessary foundations, required for the development of the project.

2.1 RFID Technology: Principles and Applications

The Radio Frequency Identification (RFID) technology offers a smooth, efficient, and accurate method for data transfer and object tracking, without the need for

direct line of sight. It uses specific frequencies within the electromagnetic spectrum to enable wireless data transfer between tags and readers, enabling instant item identification and tracking [3]. This section aims to detail the physics, operational principles, types, and main applications of the RFID technology, providing a comprehensive understanding of its role and capabilities within the retail and logistics sectors.

2.1.1 Fundamentals of RFID Technology

Electromagnetic waves are waves that consist of oscillating electric and magnetic fields. In contrast to mechanical waves that need a medium to travel, electromagnetic waves can propagate through vacuum at the speed of light. One of the main characteristics of these waves is that they are transverse (Figure 2.1). This property means that the direction in which the wave travels is perpendicular to the direction of the oscillation. For electromagnetic waves, this means that the electric and magnetic fields are perpendicular to each other and to the direction of the wave's propagation. This orientation allows electromagnetic waves to carry energy across vast distances [4].

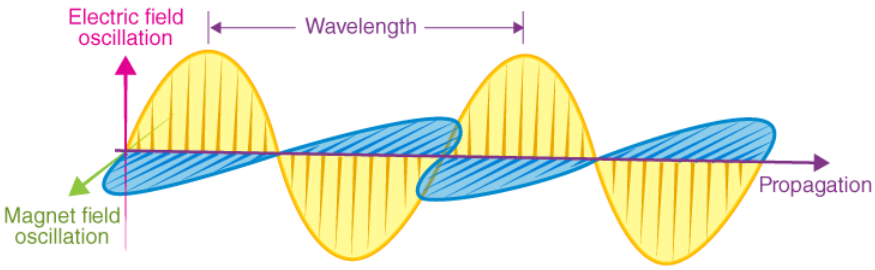


Figure 2.1: Electromagnetic wave propagation [5].

These waves are characterized by their wavelength (λ), frequency (f) and the speed of light (c):

$$c = \lambda \cdot f \tag{2.1}$$

The frequency of the electromagnetic wave will change its properties and its suitability for different RFID applications. For example, RFID systems typically operate in the Low Frequency (LF), High Frequency (HF), and Ultra High Frequency (UHF) bands, each offering different advantages in terms of range, data transmission rates, and interference. For the context of RFID readers used in retail, UHF is the frequency commonly used. It must follow the EPCglobal Gen2 ISO/IEC 18000-6:2013 standard, which enables global interoperability between all the readers [6].

As electromagnetic waves propagate, they go against different materials that can absorb, reflect, or transmit these waves. These interactions (Figure 2.2) can significantly affect the efficiency and reliability of the communication between tags and readers. Materials such as metals will reflect more while liquids will absorb more energy, resulting in challenges for RFID system design. Specially in more complex environments with lot of noise like industrial settings or fluid filled spaces [7].

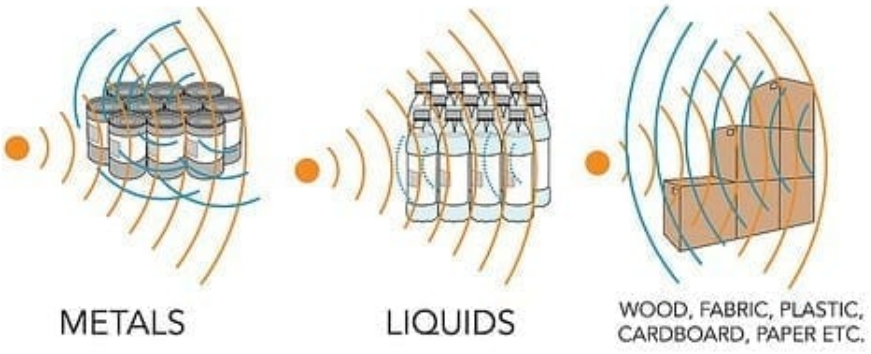


Figure 2.2: Electromagnetic wave propagation against different materials [8].

The fundamental operation of an RFID system involves the transfer of energy from the reader to the tag, enabling data exchange without direct contact. This process is illustrated in Figure 2.3 and it begins when an RFID reader emits electromagnetic waves towards a tag. The tag's antenna captures this energy, activating the tag. The activated tag then modulates the backscattered waves according to the stored information and sends it back to the reader [9]. This exchange of energy is dictated by the tag's Radar Cross Section (RCS), which quantifies how effectively the tag can reradiate the energy it receives [10].

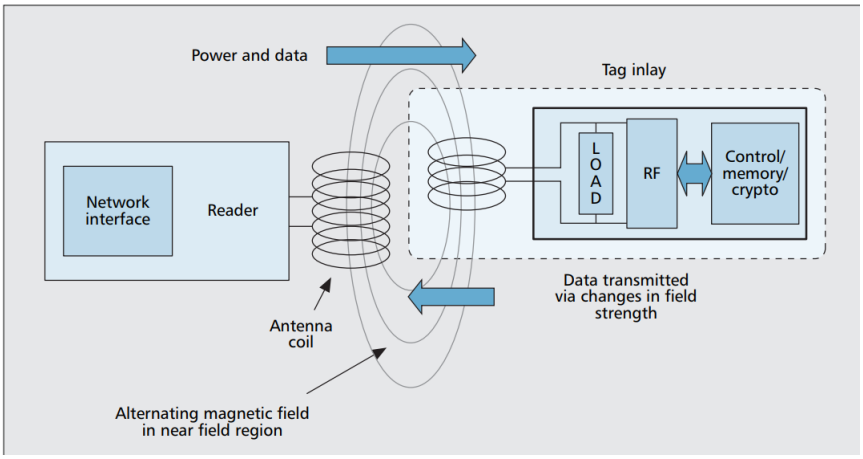


Figure 2.3: Data transmission between reader and tag [9].

The power received by the tag from the reader's transmission can be quantitatively described by the equation:

$$P_{\mathrm{r}} = \frac{P_{\mathrm{t}} G_{\mathrm{t}} G_{\mathrm{r}} \lambda^2 \sigma}{(4\pi)^3 R^4}, \tag{2.2}$$

where P_{r} represents the power received by the reader, G_{t} and G_{r} are the gains of the transmitting and receiving antennas, respectively, λ denotes the wavelength of the waves used, σ stands for the radar cross-section of the tag, and R is the distance between the reader and the tag. This equation highlights how the effectiveness of the RFID system is influenced by both the physical properties of the system components and their operational environment.

2.1.2 Principles of Operation

The RFID systems have three main components for their usual purpose: a tag, a reader, and an antenna [11].

- **Tag**: Also known as a transponder, it contains an integrated circuit and an antenna. The tag stores data and communicates with the reader. Tags can either be passive, active, or semi-passive. Passive tags have no internal power source and rely on the reader to induce a current on the tag's antenna to power it up, receive and transmit data. Active tags have their own power source, which allows them to broadcast a signal to a reader over a greater distance. Semi-passive tags also have a power source but its only used to power the internal circuit while relying on the reader to start communication [11]. Figure 2.4 shows an RFID tag used within the scope of the described project.

Figure 2.4: Sensormatic RFID tag.

- **Reader**: Or interrogator, sends and receives radio waves through its own antenna to communicate with RFID tags. The reader emits a signal that should activate the tag, which then modulates the signal with its own information and sends it back to the reader [11]. An example device of this type is presented in Figure 2.5.

Figure 2.5: IDX-4000, Sensormatic RFID reader.

- **Antenna**: This component (Figure 2.6) broadcasts radio frequency signals between the tag and the reader. The design and size of the antenna determine the system operational range and efficiency. Both readers and tags have antennas that are specifically designed based on the field requirements [11].

Figure 2.6: IDA-1000 , Sensormatic RFID antenna for IDX-4000.

2.1.3 Integration of Barcodes and RFID Electronic Product Code

There are multiple ways to encode a barcode (Figure 2.7) of an item. Within this context the most important are the specifications for EAN-13 and Universal Product Code (UPC)-A [12][13]. The EAN-13 and UPC-A barcodes encode the Global Trade Item Number (GTIN), which uniquely identifies a product type. The Serialised Global Trade Item Number (SGTIN) extends the GTIN by adding a unique serial number to each specific item [14].

Figure 2.7: EAN-13 encoded barcode.

An Electronic Product Code (EPC) is a unique identifier used in RFID systems to differentiate individual items. In the retail industry an EPC is embedded in an

RFID tag attached to a product. When the RFID reader scans the tag, it returns the EPC. This EPC can then be used to access information about the item. This allows retailers to track products from the point of manufacture through distribution centers and into retail stores, creating visibility of inventory levels through the whole supply chain [15].

The EPC is the format used to encode the SGTIN into RFID tags. It standardizes the way the SGTIN is represented in RFID systems, ensuring interoperability across different technologies and platforms. By scanning an EPC it's possible to retrieve the SGTIN of an item [16].

It's also the EPC on RFID tags that contains the information if a tag is sold or not, which will control the exit system on a store. After a tag is sold, there is a specific bit depending on the vendor, which will be flipped. The pedestals of an exit system, with the assistance of RFID readers will alarm using this bit to detect theft.

2.2 Hardware-in-the-Loop

The HIL approach is an advanced simulation technique used to develop and validate complex real-time systems. By integrating physical hardware components in a dynamic, real-world simulation, HIL testing offers a powerful tool for system verification and validation across multiple industries including automotive, aerospace, and electronics [17].

This technique distinguishes itself by incorporating actual hardware components into a simulation loop that executes system responses in real-time to simulated input signals, like illustrated in the example of Figure 2.8. As a result, developers can observe how hardware responds to theoretical scenarios and conditions in a controlled, repeatable environment. Some strong points of this technique can be pointed out:

- **Real-Time Feedback**, allows an instant overview into how a system reacts to inputs, enabling rapid iterations and development.
- **Safety and Risk Management**, as it can simulate conditions that may be hazardous in the real world, HIL testing reduces the risks compared to a live test.
- **Cost Efficiency**, since it reduces the need for multiple physical prototypes and shows defects early in the development cycle, reducing overall project costs.

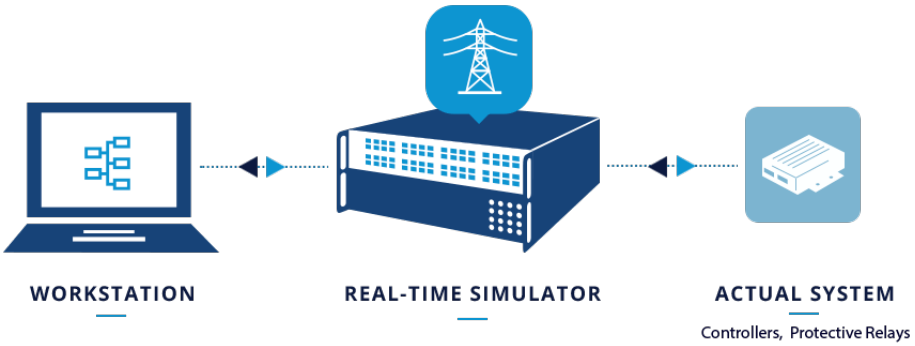


Figure 2.8: HIL generic example [18].

The main goal of including real hardware is to ensure that the system behaves as intended in a theoretical model, but also responds accurately to the physical signals and interactions it would have in actual use [19].

2.2.1 Alternative Approaches

There exist other alternatives to HIL, which have been considered for system simulation and testing:

- **Software-in-the-Loop (SIL)**: This approach involves integrating the software code into a simulation environment without the physical hardware. This is useful for early stages of testing, but lacks the real hardware to make the tests close to reality [20].
- **Full Hardware Emulation**: Emulating hardware entirely through software can be useful for testing. But an emulation will fail to perfectly replicate the real hardware, so the drawback is that the results won't be as faithful [21].
- **Hardware/Software Co-Simulation**: Using only virtual models, to simulate both the hardware and software, will provide the biggest flexibility from all methods. This method has the disadvantage of being the one most far way from reality [22].

Given the limitations of the alternative approaches, HIL was selected as the most suitable approach. It provides the highest level of accuracy among the methods discussed which leads to a more reliable and robust system, as most issues will be able to be detected before deployment.

2.2.2 Hybrid Hardware-in-the-Loop

The Hybrid Hardware-in-the-Loop (H-HIL) approach uses a setup that bridges the gap between purely simulated environments and full-scale physical testing by integrating both real and virtual components. This hybrid setup (Figure 2.9) is particularly advantageous for evaluating complex systems where certain components are either too costly or impractical to engage in repeated tests [23].

In an H-HIL configuration, the physical elements, typically sensors, actuators, or control hardware, interact with a simulated environment that models the dynamics of the system under test. This interaction occurs in real-time, reflecting a more accurate depiction of how the system would perform under operational conditions. The key benefit of this setup is its ability to provide realistic feedback on the system's behavioral response without the risk or expense of a complete physical prototype [24].

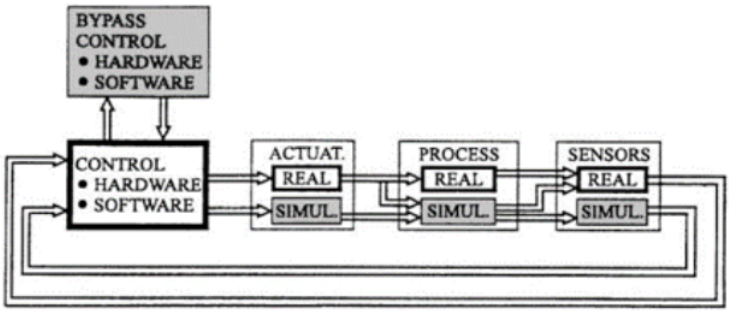


Figure 2.9: Hybrid structures for HIL simulations [24].

2.2.3 HIL Simulation for RFID-Enabled Robot Navigation

This subsection will dissect a paper that presents an innovative application of HIL simulation, integrated with RFID technology, to enhance the navigation capabilities of mobile robots in industrial settings [25]. The primary focus was to develop a software architecture that incorporates RFID sensors into the motion strategy of a simulated mobile robot, increasing real time interaction and decision-making based on environmental data captured via RFID tags, with the general overview being on Figure 2.10.

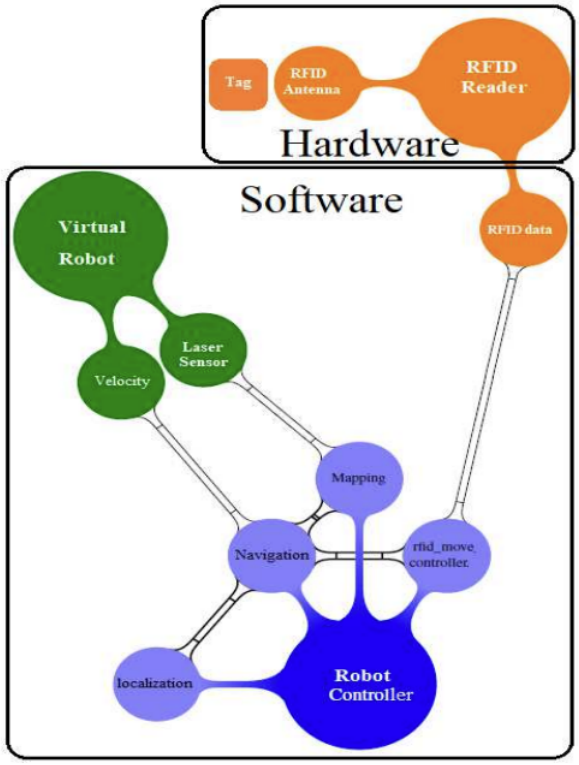


Figure 2.10: General overview of the proposed solution [25].

The RFID reader fetches the data from tags through antennas (Figure 2.11) and uses it to dynamically control the robot's speed, direction chance, or setting new checkpoints based on the conditions and objectives.

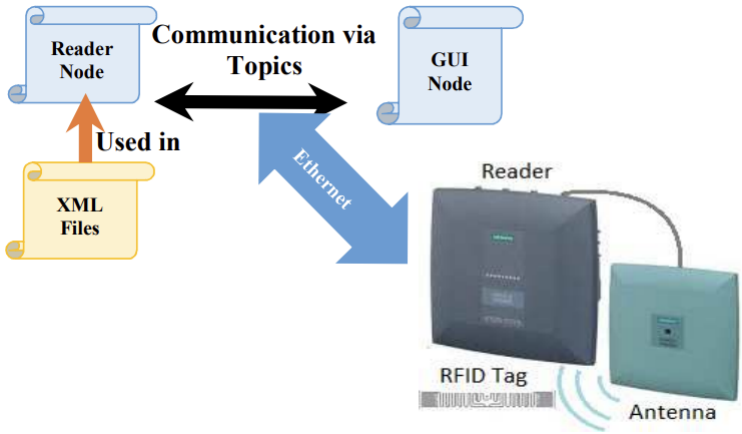


Figure 2.11: Communication of the RFID system [25].

The proposed HIL simulation framework involves a dual setup where the RFID hardware is deployed in a real environment, while the mobile robot is simulated through a Gazebo 3D simulation environment running under Robot Operating System (ROS). This setup allows for the testing of RFID-driven navigation algorithms

in a controlled yet realistic virtual environment, reducing the risks and costs associated with physical tests.

2.3 Black-box Simulation for System Testing

Black-box simulation is a testing methodology where the internal workings of a system are not considered or known by the tester. The system is treated as a "black-box", meaning that only the inputs and outputs of the system are observed, and its internal structure, design, or implementation remains unknown as exemplified by Figure 2.12. This contrasts with white-box testing, where the internal logic is visible and test cases are designed based on the system's internal workings [26] [27].

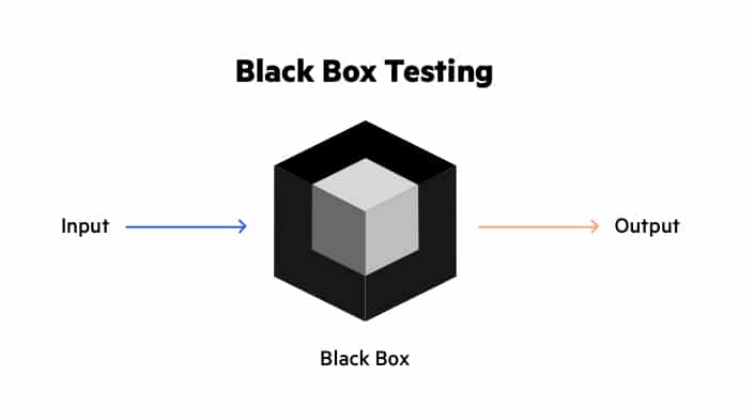


Figure 2.12: Black-box simulation [28].

A black-box simulation is widely used to validate system behavior, verify compliance with specifications, and test system interactions. It focuses on ensuring that the software meets the defined requirements without needing detailed knowledge of how the system's functionality is achieved.

2.3.1 Principles of Black-box Simulation

The core principle behind black-box simulation is that it should be able to abstract from the system internal operations. This allows testers to focus only on the system functionality and performance based on its inputs and expected outputs. Generally speaking, black-box testing can be described by [29]:

- **Input-Output Analysis**: The system is fed a set of inputs, and the outputs are analyzed without understanding or relying on the internal logic.
- **Specification-Based Testing**: Test cases are generated based on functional specifications, ensuring that the system behaves as intended according to the documented requirements.

- **No Access to Internal Code**: The test environment is designed without access to the source code or internal components, allowing the test to simulate real world conditions where users interact with the system at a higher level.
- **Test Coverage**: The goal is to cover as many functional requirements and input combinations as possible to ensure robust validation.

2.3.2 Advantages and Limitations

Like with every other testing model there are advantages and limitations, namely:

- Advantages
  - **Ease of Use**: Testers do not need specialized knowledge of the internal system structure, making black-box simulation accessible to a broader range of testers.
  - **Realistic Testing Environment**: Since the internal logic is hidden, black-box testing can simulate end-user interactions more accurately, providing a more realistic view of the system's behavior.
  - **Modular**: It is effective in modular systems, where individual components can be tested independently of the overall system architecture.
- Limitations
  - **Limited Test Coverage**: Since black-box simulation focuses only on the system's inputs and outputs, it may not cover all possible scenarios that could be caused from internal faults.
  - **Not possible to Debug Internal Logic**: Black-box simulation does not provide insight into internal system failures, making it harder to diagnose and fix underlying issues related to the source code.

2.3.3 Different Types of Simulation in Modern Systems Testing

In modern system design, various testing methodologies such as black-box, white-box, and grey-box testing are developed to achieve comprehensive coverage [30]. For instance, in a CI/CD pipeline, black-box tests are used to ensure that new features or updates do not introduce regressions or break existing functionality, while white-box tests can be used to ensure internal code still functions correctly. Grey-box testing combines both internal and external perspectives while having partial knowledge of the system's internals.

For the testing tool being developed, black-box testing was chosen because it is suitable for both QA testers and developers. Since QA does not have access to the

source code, black-box testing allows to validate system functionality based solely on inputs and expected outputs.

As the complexity of software and hardware systems continues to grow, black-box simulation will remain an important tool in ensuring system stability and reliability. The ability to abstract from the internal complexity while focusing on external outputs, makes it highly adaptable to a wide range of testing scenarios, from Internet of Things (IoT) systems to large scale distributed applications [31].

2.4 Continuous Integration and Continuous Deployment

The CI/CD pipelines are tools that can automate the processes of integration, testing, and deployment, making it possible for the developers to deliver code changes more frequently, reliably, and efficiently [32]. In the context of CI/CD, a *pipeline* refers to an automated sequence of stages (Figure 2.13), that source code changes undergo, from initial integration to deployment in production environments [33]. A CI/CD pipeline allows:

- **Source Control**: Managing and versioning code changes in a shared repository.
- **Build**: Compiling the source code.
- **Automated Testing**: Running a group of tests to verify the functionality, performance, and security of the code.
- **Deployment**: Automatically deploying the tested code to staging or production environments.
- **Monitoring**: Monitor the deployed applications to ensure no issues are found.

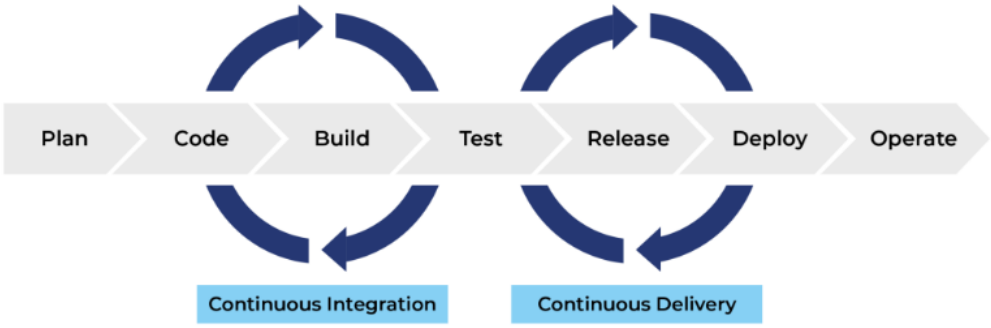


Figure 2.13: CI/CD Pipeline [34].

Currently, there are several CI/CD tools available to use with advantages and disadvantages between each of them [35]. For the context of the project two different

tools were researched more in depth, since they align with this project on specific criteria:

- **Jenkins**: On Sensormatic, some projects were already using jenkins.
- **Gitlab CI**: The source code of this specific project, is hosted on gitlab.

2.4.1 Jenkins

Jenkins is one of the most popular open-source automation servers that helps to create continuous integration and continuous deployment in software projects. Known for its helpful community support and extensive plugin ecosystem, Jenkins allows for the customization and expansion of its capabilities to fit a wide range of development needs [36].

Jenkins works by creating jobs, a Jenkins job is a specific task or a set of steps configured within Jenkins to perform actions such as building, testing, and deploying software. Jenkins functions by pulling updates from a source code repository as they are made and then initiating automated builds and tests. This process, which is defined within a Jenkinsfile using a domain-specific language, can be triggered by multiple different types of events, commits, scheduled via cron, defined manually and more. Once Jenkins completes a build, it can deploy the build to a server and provide feedback on the outcome through its user interface, email, or other notification methods [37].

Some of the advantages of Jenkins can be summed up as follows:

- **Flexibility**: Jenkins pipeline setup through code approach provides developers with great control over build and deployment processes.
- **Extensive Plugin System**: With over 1,000 plugins available in its marketplace, Jenkins can be integrated with a big number of types of development, testing, and deployment tools.
- **Community and Support**: Jenkins has a vast and active community. The accessibility of numerous resources help in resolving issues quickly and efficiently.
- **Familiarity and Adoption**: Jenkins is already utilized in different projects within Sensormatic. This can reduce the learning curve and integration effort for new projects.

On the other hand, some disadvantages can also be pointed out:

- **Complex Setup**: Setting up Jenkins, especially in a complex environment, can be challenging. The initial configuration and management of Jenkins require significant sysadmin knowledge.

- **Resource Intensive**: Jenkins can be quite demanding in terms of system resources, especially when managing multiple projects and handling large builds.
- **User Interface and Usability**: While functional, Jenkins' user interface is often considered less modern and intuitive compared to newer CI/CD tools like GitLab CI.

2.4.2 GitLab CI

GitLab CI is an integral part of the GitLab platform, which is a web based DevOps tool that provides a Git repository manager providing wiki, tracking issues, and CI/CD pipeline features. The ability to have an integration of these services within a single platform makes GitLab CI a good choice for projects that aim to streamline their development and deployment processes [38].

GitLab CI/CD pipelines are configured using a YAML file called `gitlab-ci.yml` placed at the root of the repository. This file defines scripts that are executed by runners (agents that run builds), what to execute using Docker containers, and more. The process automates the build, test, and deployment stages for every commit or push to the repository, creating continuous integration and delivery with minimal effort on the part of developers [39].

The advantages of GitLab CI can be summarized as follows:

- **Single Application for the entire DevOps Cycle**: GitLab offers a single application for the entire software development process. This approach reduces the complexity of managing multiple tools and integrates CI/CD closely with source code management.
- **Powerful Integration Capabilities**: Since GitLab CI is tightly integrated with GitLab itself, issues, merge requests, and repositories are all linked directly to the CI/CD pipelines. This integration increases visibility and traceability across the project.
- **Auto DevOps**: GitLab can automatically detect, build, test, deploy, and monitor applications based on a predefined CI/CD configuration, which significantly simplifies the setup for new projects.
- **Project already hosted on GitLab**: A significant advantage for this specific project is that the source code is already hosted on GitLab. This existing setup simplifies the integration of CI/CD pipelines, as there is no need to set up additional integration's or migrate the code base, reducing setup time and potential integration errors.

However, there are a few disadvantages to consider:

- **Resource Intensive**: Similar to other CI/CD tools, GitLab CI can be resource-intensive, specially when running multiple pipelines that include complex builds and tests.
- **Learning Curve**: While GitLab CI is powerful, new users may face a learning curve understanding all of its features and best practices for setup and configuration.

2.5 Application Programming Interface

An API is a set of protocols and tools that allow software components to communicate with one another exemplified by Figure 2.14. APIs are essential in modern software development as they enable different applications, systems, and services to interact. By documenting clear interfaces, APIs abstract the internal workings of a system, allowing external systems to utilize its functionality without needing to understand the underlying architecture [40].

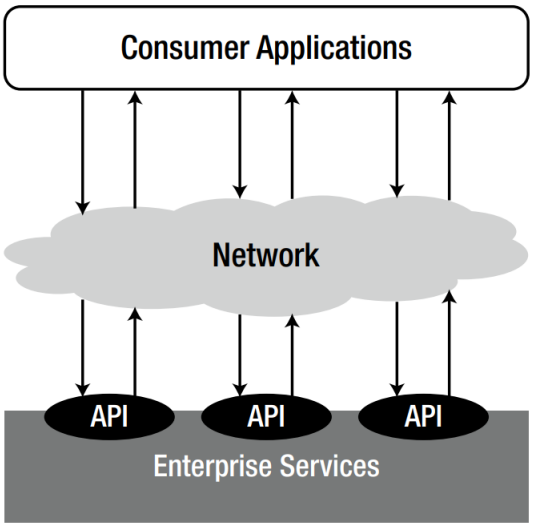


Figure 2.14: An API provides an interface for an user to interact with over the network [40].

APIs can be implemented in various protocols such as Representational State Transfer (REST) , Simple Object Access Protocol (SOAP), and GraphQL. In the context of software development, REST APIs are popular due to their simplicity, statelessness, and the ability to easily integrate.

2.5.1 API Data Interchange Formats

In API development, the format in which data is transferred plays an important role in ensuring compatibility and ease of use. Two of the most widely used formats are JSON and XML.

The JSON format is lightweight and its easy for both humans and machines to read and write. It is widely used in web APIs due to its simplicity and easy integration with most programming languages. Furthermore, JSON objects are structured in a key to value format, and their small size makes it a good choice for high performance applications where bandwidth efficiency is important [41]. Listing 2.1 shows an example of a typical JSON response.

```
{
  "id": 1,
  "name": "Example",
  "status": "active"
}
```

Listing 2.1: Example of a JSON data format response.

The XML is another data format that is widely used for API data interchange, being more common in SOAP based APIs. It is a more verbose format (Listing 2.2) compared to JSON but offers a higher level of flexibility due to its strict hierarchical structure. It can also be used to describe complex data structures and includes support for attributes, namespaces, and schema validation [42].

Even though JSON has become a more popular choice, XML is still used in industries that require structured data with detailed metadata and more rigid validation requirements, such as in enterprise applications or systems where backward compatibility with legacy systems is necessary.

```
<response>
  <id>1</id>
  <name>Example</name>
  <status>active</status>
</response>
```

Listing 2.2: An example of an XML data response.

2.5.2 API Design Principles

When developing an API, there are several important principles to ensure it is efficient, scalable, and easy to use [43]:

- **Statelessness**: The API should be stateless, meaning each request from the client should contain all necessary information to process the request. This allows for better scalability and simpler design.

- **Consistency**: The API should maintain consistent formatting, error handling, and responses on all endpoints to minimize confusion.
- **Versioning**: Over time, APIs go over changes, and backward compatibility may be needed. Correct versioning helps manage changes without affecting existing clients.
- **Security**: API security is a priority, specially when it's handling sensitive data. Techniques like token based authentication and ensuring the use of Hypertext Transfer Protocol Secure (HTTPS) connections should be the standard.

2.6 Introduction to Go

Go or Golang, is a statically typed, compiled programming language designed by Google. Released in 2009, it was built with the goal of simplifying software development, enhancing performance, and addressing the limitations of other programming languages such as C and Java. Go's simplicity, combined with its powerful concurrency model, has made it a popular choice for modern backend development, cloud services, and systems programming [44].

One of the standout features of Go is its inherent ability to support cross operating system and cross architecture development *natively*. Go applications can be compiled to run on multiple platforms with minimal or no changes to the source code. This is achieved due to Go's built in toolchain, which simplifies the process of cross compilation [45]. Go can easily compile code to run on a wide range of operating systems [46], including Linux, Windows, macOS, FreeBSD, among others.

This versatility makes Go a good option for developing software that needs to run in multiple environments. By setting environment variables during the build process (`GOOS` for the operating system and `GOARCH` for the architecture), developers can compile their Go applications for different target platforms very easily.

In addition to supporting multiple operating systems, Go can also target different Central Processing Unit (CPU) architectures natively [46], including x86 (32-bit and 64-bit) and ARM (32-bit and 64-bit) to just name the most common.

This cross architecture flexibility makes Go a great choice for developing applications that need to run on a big range of devices, from servers in the cloud to embedded systems. By targeting multiple architectures with a single codebase, Go reduces the complexity of maintaining separate code versions for different platforms, which will improve development efficiency and scalability.

Important to note that support for cross operating system and cross architecture development, was always supported on Go, but was a much bigger challenge prior to Go 1.5 version. Requiring separate Go compilers for each target platform. On Go 1.5

version which was released in 2015, all supported architectures and operating systems were included in the standard distribution. Developers could cross-compile by simply setting the `GOOS` and `GOARCH` environment variables before building, without any additional setup [47].

2.6.1 API Development with Go

The Go standard library includes support for Hypertext Transfer Protocol (HTTP), making it easy to build an API without relying on external frameworks. The `net/http` package is at the center of Go's HTTP handling capabilities, allowing developers to quickly create a RESTful API with a minimal setup [48]. Listing 2.3 presents a basic API in Go that illustrates the simplicity of the setup process. In this example a basic web server is created with minimal code and with no external libraries. By using Go's builtin concurrency (e.g., goroutines), it is easy to scale this API to handle high traffic without heavily reducing performance.

```
1  package main
2
3  import (
4      "net/http"
5      "fmt"
6  )
7
8  func helloHandler(w http.ResponseWriter, r *http.Request) {
9      fmt.Fprintf(w, "Hello, World!")
10 }
11
12 func main() {
13     http.HandleFunc("/hello", helloHandler)
14     http.ListenAndServe(":8080", nil)
15 }
```

Listing 2.3: Example of a Go API endpoint.

2.6.2 Black-box testing in Go

In the context of the Go programming language, black-box testing can be used to verify if Go programs are performing as expected. Especially in concurrent environments, where goroutines and channels make it much harder to do proper tests. One framework designed for black-box testing in Go that is most common is Go2Pins, with an overview of the framework on Figure 2.15 [49].

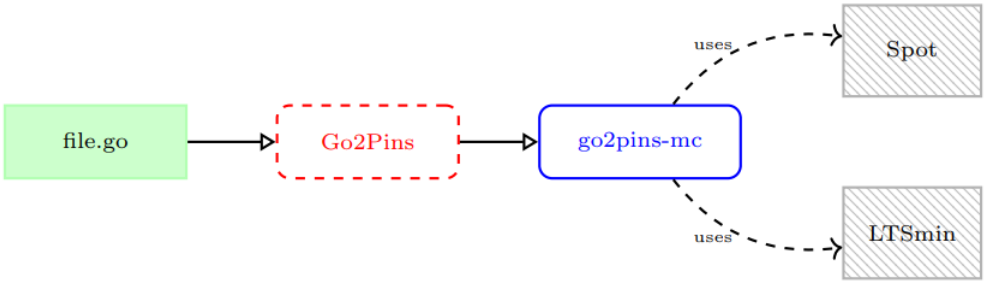


Figure 2.15: Overview of Go2Pins [49].

This tool links Go programs with model checkers such as LTSMin [50] and Spot [51] to perform Linear Temporal Logic (LTL) verification, as illustrated in Figure 2.16. This LTL technique allows the specification of temporal properties, that need to be checked across multiple program states. It operates by abstracting the internal transitions of a Go program into a form compatible with these model checkers. This abstraction is accomplished through a testing method named *black-box transitions*.

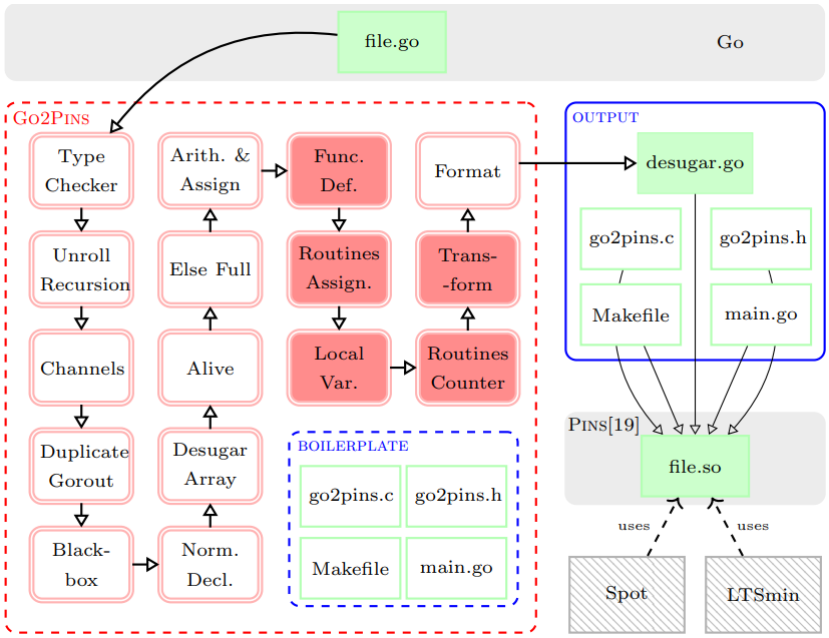


Figure 2.16: The dashed boxes represent the Go2Pins tool while the blue plain box represents the output directory produced by Go2Pins [49].

Black-box transitions testing, reduces the complexity of Go programs during testing. By abstracting away non important internal transitions, Go2Pins allows the model checker to focus only on the necessary state changes, significantly reducing the state space explosion problem [52].

This technique is particularly useful when testing systems that incorporate external libraries or standard library functions. For example, if a Go program uses a function from the Go standard library, such as `math.Sqrt`, Go2Pins abstracts the internal workings of that function. This prevents unnecessary complexity from affecting the model-checking process, as the tool treats external function calls as atomic operations and focuses only on the relevant program behavior.

It's important to note that black-box testing and black-box transition testing, while similar, are not the same. Black-box testing focuses on evaluating the system functionality without any knowledge of its internal structure or code, treating the software as a "black box". On the other hand, black-box transition testing specifically deals with verifying the correct transitions between different states of the software system, ensuring that it behaves as expected during state changes. Black-box transition testing has a more specialized focus on the system state behavior, while black-box testing uses broader functional testing.

2.6.3 Custom Scripting Language in Go

A custom scripting language is a domain-specific language designed to automate tasks, increase functionalities, or simplify complex operations on a specific application or system [53]. These languages allow users to write scripts that can interact with the application internal API, providing flexibility and customization without having to change the core codebase.

To interpret a custom scripting language using Go, the following steps are typically involved [54]:

1. **Tokenization**: The script's source code is broken down into tokens using a lexer. Go's `text/scanner` package can be used to read Unicode characters and identify tokens.

2. **Parsing**: The sequence of tokens is parsed to build an Abstract Syntax Tree (AST), representing the grammatical structure of the scripting language. This can be achieved using recursive descent parsers or parser generator tools compatible with Go.

3. **Execution**: The AST is used to execute the script according to the defined semantics. This involves implementing the logic for each language construct, such as variables, control flow statements, and function calls.

To simplify, it can also be used a Line-by-Line approach where each line of the script gets parsed and executed independently from the other lines.

A custom scripting language called DOME-X based on Distributed Object Model Environment (DOME) (Figure 2.17), is introduced to facilitate the deployment and

testing of distributed automation systems [55]. Like other domain-specific scripting languages, DOME-X enables developers to automate repetitive tasks and manage distributed processes across various nodes.

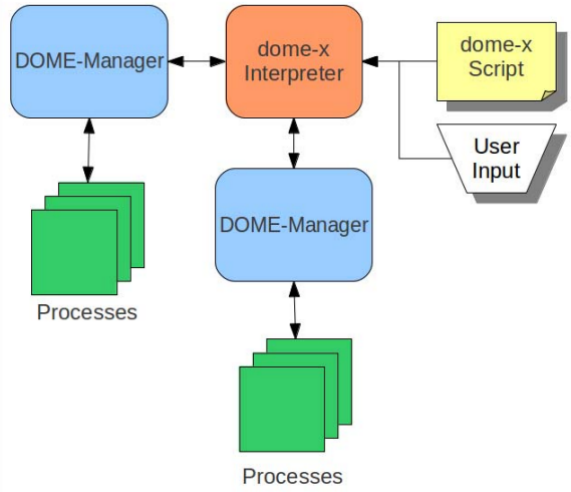


Figure 2.17: Overview of the script processing with DOME-X [55].

The DOME-X language provides a simplified scripting interface that interacts with the DOME framework. It allows users to deploy applications, start and stop processes, modify settings, and observe the behavior of the system in real time. However, DOME-X emphasizes on the ease of use for users with limited automation experience by maintaining a minimal set of commands and focusing on runtime flexibility.

2.7 MQTT Protocol

The MQTT is a lightweight, publish-subscribe protocol that operates over Transmission Control Protocol/Internet Protocol (TCP/IP). It is designed to function even with low-bandwidth and high latency networks. This protocol follows a client-server architecture (Figure 2.18) where a central broker acts as a middleman between the communication between clients. Moreover, clients can act as publishers, subscribers, or both. Publishers send messages to a topic, and subscribers receive messages from topics they are subscribed to. This design enhances scalability and message routing [56].

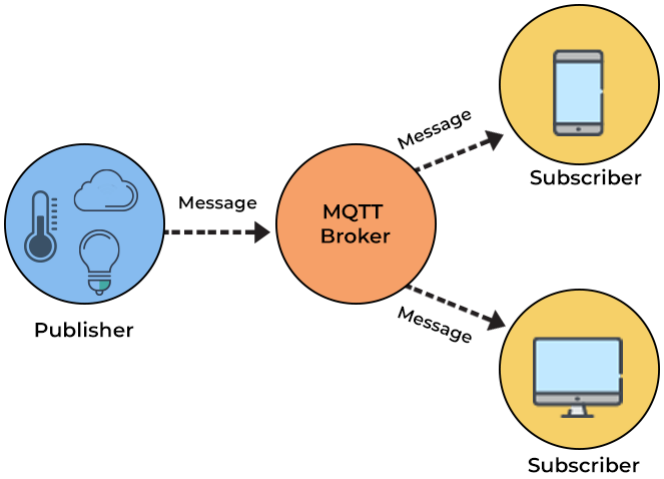


Figure 2.18: MQTT Process [57].

When selecting which broker to use, it's essential to consider the unique features (Table 2.1) and performance characteristics of each option. The choice should be guided by the specific requirements of the use case, as different brokers are good in different scenarios [58].

Table 2.1: Features of the most popular MQTT brokers [58].

Broker	**Cluster**	**Bridge**	**Purpose**	**Language**
ActiveMQ	✓	✓	Product	Java
D-MQTT		✓	Research	C
DM-MQTT		Rendezvous system	Research	-
Emitter	✓		Product	Go
Emma		Dynamic	Research	Java
EMQX	✓		Product	Erlang
FogMQ		Bridge-based	Research	-
HiveMQ	✓	✓	Product	Java
HbMQTT			Project	Python
ILDM		External Tool	Research	-
Aedes	✓	✓	Project	Javascript
Mosquitto		✓	Product	C
Moquette-io			Project	Java
MQTT-NEG		External Tool	Research	Python
MQTT-ST		✓	Research	C
RabbitMQ	✓	✓	Product	Erlang
Schmit et Al.		Dynamic	Research	C
VerneMQ	✓	✓	Product	Erlang

2.7.1 Mosquitto MQTT Broker

The Eclipse Mosquitto is an open source MQTT broker that currently supports the protocol versions 3.1, 3.1.1, and 5.0. Mosquitto is suitable for both small devices and large scale environments. By following the MQTT standards, it ensures compatibility with multiple MQTT clients [59].

Using the Mosquitto application, MQTT clients are usable through the command line `mosquitto_pub` and `mosquitto_sub` are used for publishing and subscribing to messages, respectively [60][61]. These tools are useful for interacting with MQTT brokers without developing custom client software.

The choice of using Mosquitto as the MQTT broker for this project, was influenced by it being the standard broker used by the firmware team while meeting the necessary requirements.

2.8 Discussion

This chapter explored technologies and methodologies pertinent to the project. By understanding these technologies, a solid foundation was built for designing and developing a testing framework in the context of an RFID POS in retail.

The choice of technologies, including the MQTT protocol and API development using Go, created a foundational understanding of how the components can be brought together effectively. This knowledge establishes a clear groundwork for developing the TestFramework tool, ensuring the RFID system can be tested in a simulated yet accurate environment without direct dependence on third-party systems like the Keonn API. Moreover, this information serves as a foundation for the upcoming chapter, which examines particular in-store hardware and software that this project seeks to improve.

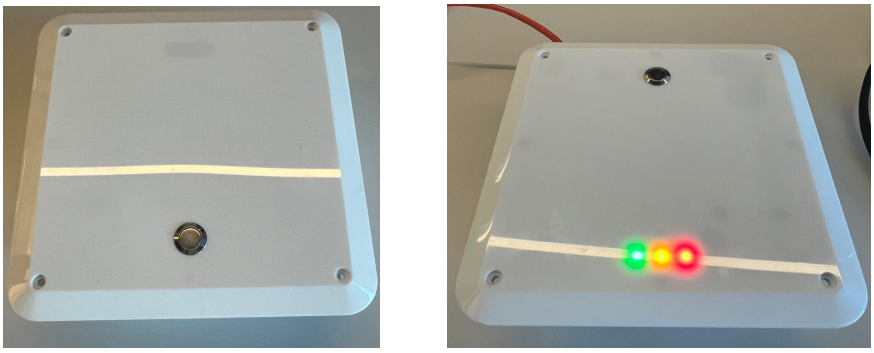
Chapter 3

In-store Hardware and Operations

This chapter presents the information about the POS that will be used, the Advanpay 120. This device is a specific model from the Advanpay series from Keonn [62]. For simplicity, throughout the rest of the document, the Advanpay 120 will be referred as only Advanpay. The architecture and the process of an in-store transaction will be detailed, followed by the approach to be used to integrate the Advanpay with HIL and black-box testing. The purpose being the later integration of these tests with a CI/CD tool.

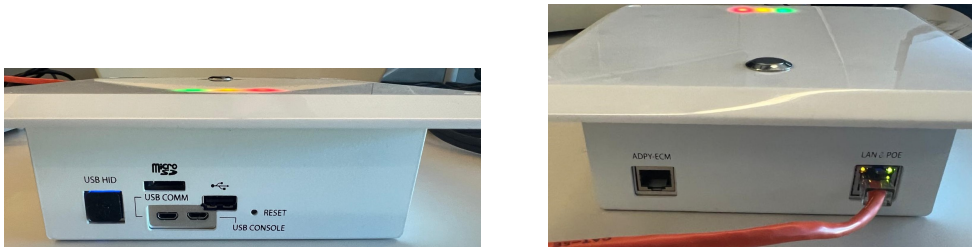
3.1 Advanpay Hardware

The Advanpay hardware POS device (Figure 3.1a) and its firmware are built and developed by Keonn, a Sensormatic partner. The Advanpay is powered by Power Over Internet (POE) and can be optionally connected to an ethernet network switch. It also has three Light-Emitting Diode (LED) (Figure 3.1b) (red, yellow and green) and a buzzer to reflect its current status to the user.



(a) (b)

Figure 3.1: The Advanpay device: (a) overview and (b) status LED.



(a) (b)

Figure 3.2: The Advanpay device: (a) connectivity and (b) connectivity and power.

For connectivity, the device supports Universal Serial Bus (USB) Human Interface Device (HID) functionalities across its USB ports (Figure 3.2a). It features two Micro USB ports: one is dedicated to console access, allowing the device management and troubleshooting, while the other works as an interface that can be accessed through a specific Internet Protocol (IP) address, for access through Secure Shell (SSH). Additionally, it includes one type A USB port (Figure 3.2b), a Micro SD card slot for formatting the device, and a reset button.

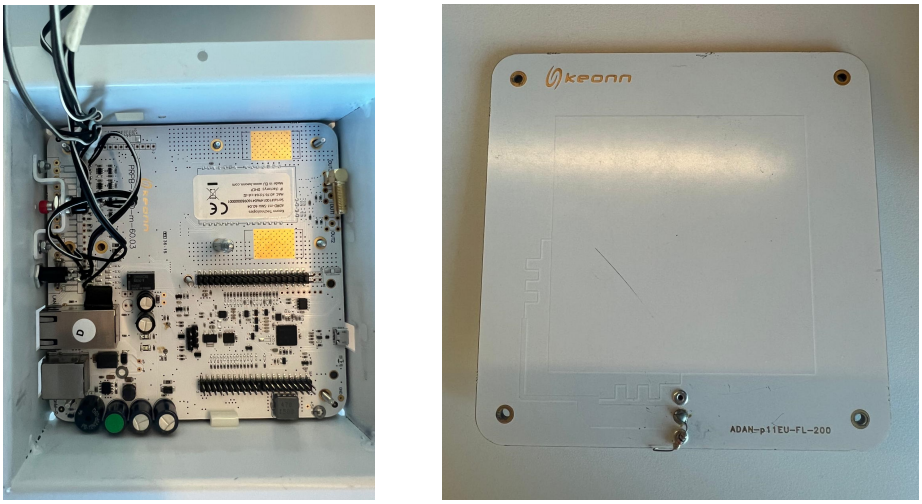
The Advanpay system uses a BeagleBone board (Figure 3.3a) that has the Texas Instruments AM3505 CPU [63]. This processor uses the ARM Cortex-A8 microarchitecture, which is based on the ARMv7-A architecture [64]. To enable RFID capabilities, it is equipped with an RFID module from Thingmagic, enabling it to perform various RFID related tasks efficiently [65].

Since the CPU is based on Advanced RISC Machines (ARM) architecture, cross compiling will be needed to compile applications to run on the Advanpay, since the machines being used for development are based on the x86-64 architecture.

The device has integrated RFID antennas developed by Keonn (Figure 3.3b), with circular polarization. This type of antenna allows the reception of signals regardless of the tag's orientation, enhancing readability and operational flexibility. Additionally, the design confines the radio frequency field to the immediate area

of the antenna, reducing the likelihood of unintentionally reading nearby tags not aligned with the system's intended target.

Refer to Table A.1, in Appendix A, for the full specifications of the Advanpay hardware system.



(a) (b)

Figure 3.3: The Advanpay device: (a) internal board and (b) RFID antenna.

3.2 Software - POS1.0

The primary function of a POS system is to process transactions. With RFID integration, AdvanPay enables contactless transactions by reading RFID tags attached to the products. The Advanpay is used as a POS to enable sale and returns of items in retail stores.

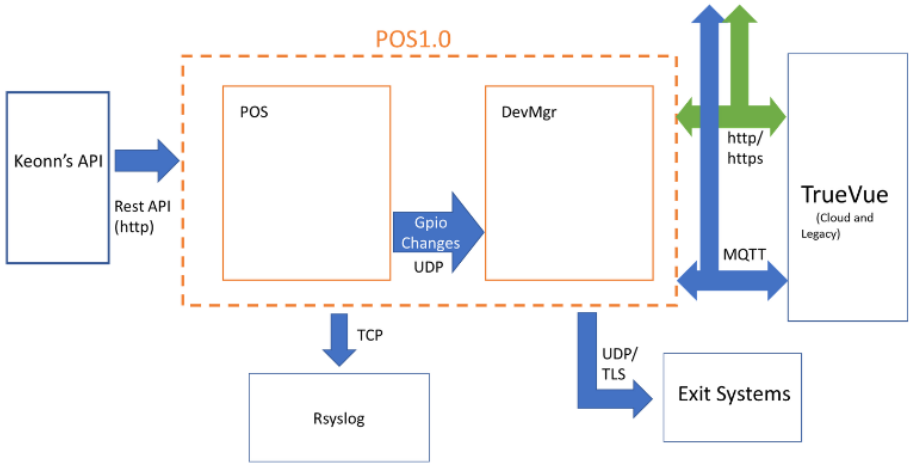


Figure 3.4: Advanpay workflow.

The Figure 3.4 illustrates the top level workflow operating in the Advanpay device, where POS1.0 refers to two Sensormatic software applications, named POS and DevMgr. These applications are fully developed by Sensormatic teams. The most important blocks to consider for this project are the Keonn's API and the POS application. Its important to distinguish the general term POS that designates the Advanpay device, and the POS application, which is a software application named POS runing in the Advanpay device.

The Low Level Reader Protocol (LLRP) is a standardized protocol used for communication between RFID readers and software applications. Developed by the EPCglobal network, LLRP is designed to support a wide range of RFID reader functionalities across different manufacturers, providing a common interface to work on [66]. Keonn's module is a HTTP REST API, it enables the user to communicate with the Advanpay hardware and make changes to the configurations without having to use the LLRP protocol directly, to communicate with the RFID reader. This method simplifies the process to integrate with Keonn's hardware. For instance, a single API call is needed to perform operations like checking if there's any RFID tags being read, or change the power of the antenna of the Advanpay device.

3.3 Software - Workflow of a Sale

With the POS1.0 there are two different modes that can be selected, Autonomous and Transaction. For this project only the Transaction mode will be considered. This mode comprises the several stages of an in-store sale, as depicted in the Figure 3.5. The sale workflow consists of:

1. **Initiating the Sale:**
    - The store operator (client) starts the sale process by interacting with the POS system.
2. **Starting the Transaction:**
    - The POS system communicates with the POS application (POS1.0) to start a new transaction.
    - The POS application begins the process of looking for tags associated with the transaction.
    - The POS application acknowledges the start of the transaction and confirms it back to the POS system.
3. **Scanning Items:**
    - The POS system prompts the store operator to begin scanning items.

- The store operator scans the item, and the POS system captures this input.
- The POS system sends the scanned item information to the POS application to add the products to the current transaction.
- The POS application acknowledges the addition of the products to the transaction.

4. **Waiting for Tag Processing:**
   - The POS system waits for 500 milliseconds to allow the POS application to process the tags.
   - After the wait, the POS system checks the status of the transaction with the POS application.
   - The POS application responds with the status and provides a list of the tags found during the scanning process.
5. **Confirming Item Detection:**
   - The POS system shows the store operator that the item was successfully found and added to the transaction.
6. **Finishing the Sale:**
   - The store operator completes the sale process by interacting with the POS system.
   - The POS system sends a command to the POS application to execute and finalize the transaction.
   - The POS application sends the tags to the exit system (to manage inventory or theft prevention).
7. **Ending the Transaction:**
   - The POS application acknowledges the successful execution and end of the transaction.
   - The POS system then sends a request to the POS application to delete the transaction from its records.
   - The POS application confirms that the transaction has been deleted.

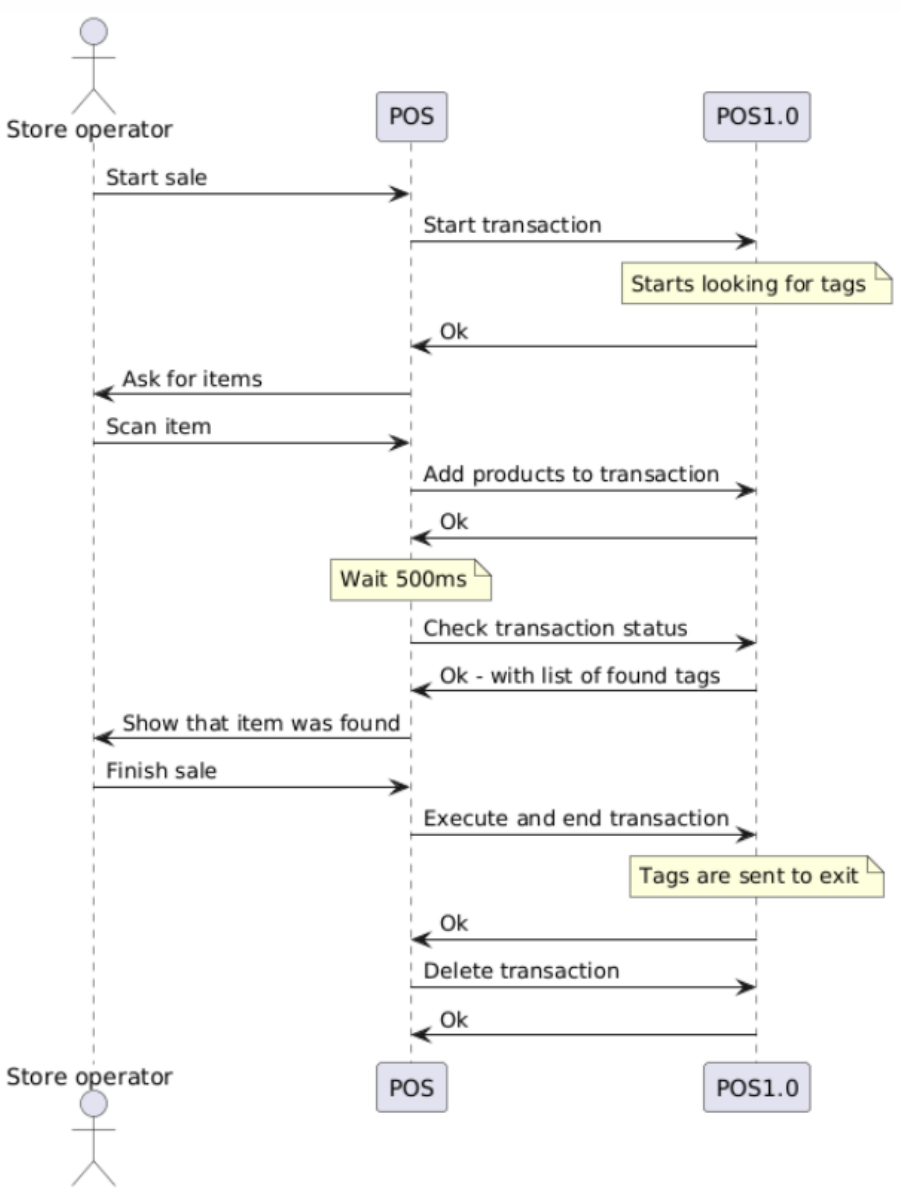


Figure 3.5: Transaction mode workflow.

3.4 Software - Transaction Mode API of the POS Application

The POS1.0 API, provides the tools to operate throughout the workflow of the transaction. In greater detail:

- **API Rate Limiting:** The API includes a limiter that controls both the number of connections to the API and the number of requests allowed per second. Each connection to the API is saved in a list of visitors. These visitors are checked and removed after 5 minutes of inactivity or if the request rate exceeds the expected limits (Average: 3 requests/second, Burst: 10 requests/second).

- **Schema:** The API supports HTTP 1.1, and all exchanged data must be in JSON format. The following HTTP headers should be set in all requests:

  - `Accept: application/json`
  - `Content-Type: application/json`

  Moreover, all requests to the API must use the following standard HTTP verbs/methods [67]:

  - `GET`, to retrieve resources.
  - `POST`, to create resources or perform custom actions.
  - `DELETE`, to delete resources.

  All `DateTime` type fields should be represented in the ISO 8601 format [68].

- **Errors:** Errors are reported with HTTP error responses, most of which adhere to the HTTP specifications. The Table-3.1 gives a brief description of the most common HTTP response codes [69]:

Table 3.1: POS application API error codes.

Response Code	Description
200 OK	The request has been successfully processed.
204 No Content	The request has been successfully processed, blank response
403 Forbidden	The server is refusing action; not having necessary permissions
401 Unauthorized	The requested resource requires authentication.
400 Bad Request	The request could not be understood by the server.
500 Internal Server Error	A generic error has occurred on the server.
415 Unsupported Media Type	The request is of an unsupported type.
404 Not Found	The requested resource does not exist on the server.

- **HTTP Authentication:** All communications between the client and the server must be authenticated using HTTP Digest Authentication. The required username and password must be configured, with credentials stored in a configuration file on the reader.

- **Transaction Management:**

1. ***Start Transaction***: The transaction begins when the RFID reader starts performing repeated inventories on all tags within the Field of View (FOV). The RFID reader will automatically perform multiple inventories with different settings to optimize matching products from the POS software with RFID tags seen.
   - **Request URL:** `POST /transaction`
   - **Request Body:** see Listing 3.1.

```
{
  "Id": "123456789",
  "EmpId": "987654321",
  "PosId": "123459876",
  "Mode": "Sale",
  "Antenna": 1
}
```

Listing 3.1: Transaction request body structure.

   - **Normal Response:** An empty 204 response is expected.
   - **Error Responses:**
     * Accept `application/json` header required.
     * Content-Type `application/json` header required.
     * Error: only one transaction supported at a time.
     * Please specify a valid transaction id.
     * Reader in Autonomous Mode, no transactions allowed.
     * Self-test is running, transaction cannot be created.
2. ***Add Products to the Transaction***: Adds a new item or group of products to the transaction. The reader will attempt to match each EAN13/UPC-A added with the SGTIN-encoded tags in the FOV.
   - **Request URL:** `POST /transaction/product`
   - **Request Body:** see Listing 3.2.

```
{
  "Products": [
    "4003994155487",
    "4003994155488",
    ...
  ]
}
```

Listing 3.2: Add sale item request body structure.

- **Normal Response:** An empty 204 response is expected.
- **Error Responses:**
  - Accept `application/json` header required.
  - Content-Type `application/json` header required.
  - Error: only one transaction supported at a time.
  - Please specify a valid transaction id.

3. ***Remove Products from the Transaction***: Removes one or more products from the transaction. Each repeated EAN13/UPC-A removes only one product; to remove two units of the same product, either provide the same EAN13/UPC-A twice or call this method twice.
   - **Request URL:** `DELETE /transaction/product`
   - **Request Body:** see Listing 3.3.

```
{
  "Products": [
    "4003994155487",
    "4003994155488",
    ...
  ]
}
```

Listing 3.3: Remove sale item request body structure.

   - **Normal Response:** An empty 204 response is expected.
   - **Error Responses:**
     - Accept `application/json` header required.
     - Content-Type `application/json` header required.
     - Please create a transaction first.
     - Please specify a valid transaction id.
     - Client has to provide at least one EAN13/UPC-A.

4. ***Check Transaction Status***: Retrieves the current transaction status. Note that the status of a product might change over time, so client POS software may check the status periodically.
   - **Request URL:**
     `GET /transaction?unmatched={Unmatched}&transaction={Id}`
   - **Request Parameters:**
     - **Unmatched (Boolean, Optional, Default: False):** If enabled, includes the unmatched field in the response.

    - **Id (String, Optional for AdvanPay):** Specifies the transaction to retrieve in a multi-transaction topology.
- **Normal Response:** see Listing 3.4.
- **Error Responses:**
    - Accept `application/json` header required.
    - Content-Type `application/json` header required.
    - Please specify a valid transaction id.
    - Please create a transaction first.

```
{
  "Id": "123456789",
  "EmpId": "987654321",
  "PosId": "123459876",
  "Mode": "Sale",
  "Start": "DateTime",
  "Products": [
    {
      "Id": "4003994155486",
      "Status": "Missing"
    },
    {
      "Id": "4003994155487",
      "Status": "Singulated"
    },
    ...
  ],
  "Unmatched": [
    "123456789ABCDEF123456789",
    "987654321FEDCBA987654321",
    ...
  ]
}
```

Listing 3.4: Check transaction request body structure.

For each EAN13/UPC-A, there are five possible statuses:

- Missing: No tag was found for the given EAN13/UPC-A.
- Singulated: Only one tag was found, or there is a clear best match among multiple tags seen.
- Multiple: Multiple tags were found, but the reader cannot yet decide which should be singulated.
- Detachable: The reader is waiting for a tag to be detached.
- Detached: The reader detached a tag for the given EAN13/UPC-A.

5. ***Execute and End Transaction***: Executes the transaction. The reader will perform the appropriate action based on the transaction mode (e.g., Sale, Return, Transfer) for products that have been matched or detached.
   - **Request URL:** `POST /transaction/execute`
   - **Normal Response:** An empty 204 response is expected.
   - **Error Responses:**
     * Accept `application/json` header required.
     * Content-Type `application/json` header required.
     * Please specify a valid transaction id.
     * Please create a transaction first.
6. ***Delete Transaction***: Deletes the current transaction and readies the reader for the next one. The response provides a description of the transaction's end state.
   - **Request URL:** `DELETE /transaction`
   - **Request Body:** see Listing 3.5.

```
{
  "Id": "123456789",
  "EmpId": "987654321",
  "PosId": "123459876",
  "Mode": "Sale",
  "Start": "DateTime",
  "Products": [
    {
      "Id": "4003994155486",
      "Status": "Processed"
    },
    {
      "Id": "4003994155487",
      "Status": "Processed"
    },
    {
      "Id": "4003994155488",
      "Status": "Failed",
      "Reason": 1
    }
    ...
  ]
}
```

Listing 3.5: Delete transaction request body structure.

   - **Error Responses:**

    - Accept `application/json` header required.
    - Content-Type `application/json` header required.
    - Please specify a valid transaction id.
    - Please create a transaction first.
    - Could not read request body.

  For each EAN13/UPC-A, there are two possible statuses:

  - Processed: The operation completed successfully.
  - Failed: The operation could not be completed. A bitmask value in the `Reason` field details the cause of the failure:
    - Bit 1: The 'sold bit' could not be written.
    - Bit 2: The sale could not be reported to TrueVUE.
    - Bit 3: The EPC could not be multicast.

3.5 Discussion

This chapter examined the in-store hardware and software, focusing on the AdvanPay device and the POS1.0 applications. By detailing the components and processes, including the workflow of a sale in Transaction mode, the chapter demonstrated how the system integrates with in-store operations. Understanding the physical and software components in detail is essential to reproduce them accurately in a Hardware-in-the-Loop setting, allowing developers to emulate real-world scenarios effectively.

The next chapter will discuss the methodology and implementation of a test framework that replicates the Keonn API and a custom scripting language, enabling black-box testing and integrating it with a CI/CD pipeline.

Chapter 4

Methodology and Implementation

To simulate and validate the interactions between the hardware components and the software systems, within the context of the project, a testing environment was implemented named TestFramework. This environment serves as a test framework for ensuring the system behaves as expected under various scenarios on the Transaction mode of the POS application. This chapter will go over the implementation of the proposed solution.

4.1 Test Framework Architecture

The architecture of the TestFramework is depicted in the Figure 4.1, where the actores, components and actions can be stated as follows:

- **GitLab**: The CI/CD pipeline starts the process by triggering the TestFramework (the repository will be hosted in Gitlab).
- **User**: A user can also initiate the process manually.
- **TestFramework** :
  - **Script Runner**: Executes the testing scripts, manages communication between components, and collects results.

- **AdvanPay Simulator**: Simulates the Keonn AdvanNet API.
- **POS1.0**: Represents the POS1.0 applications (POS and DevMgr).
- **MQTT Broker**: Manages MQTT communication between the components.

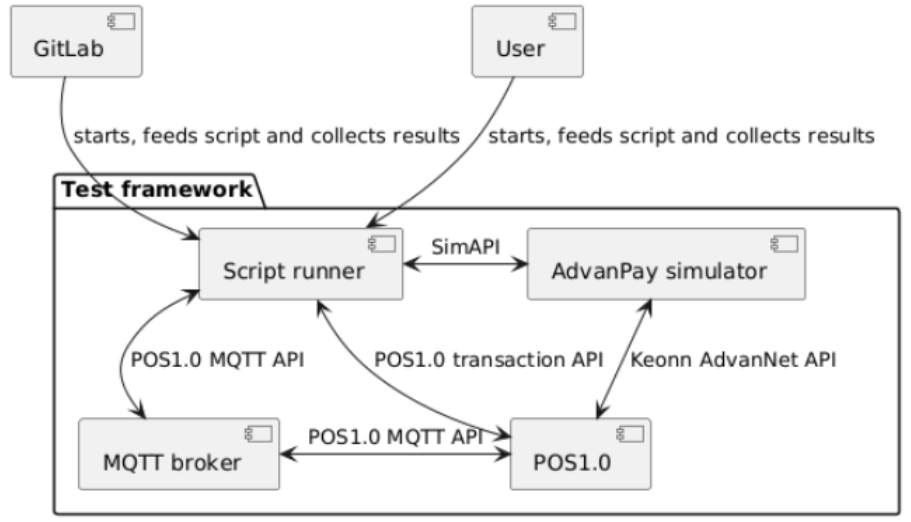


Figure 4.1: TestFramework workflow.

The core of the TestFramework is a Go application that acts as a script reader. This application interprets and executes test scripts written in a custom scripting language conceptualized specifically for this purpouse. This scripting language must meet the specifications of a sale in Transaction mode.

Furthermore, the HIL testing environment includes a REST API developed in Go, designed to mimic the features of the AdvanNet API (Keonns API), used for the Advanpay.

4.2 AdvanPay simulator API

One of the main goals of Sensormatic is to abstract from Keonns API and, to that end, a simulated API was created in Go named AdvanPay simulator. This application will handle all the requests that were needed from Keonn's module during a transaction and also for the configuration requests. It will also allow to include new endpoints that weren't available with the Keonn's API which can prove to be useful for testing different scenarios and future development directions.

4.2.1 Keonn AdvanNet Request Handlers

There are five different endpoints that are supported. All of them are used to retrieve information from the system, not the transaction. The user has two options

for modifying any of the responses that come from these endpoints. Changing the response file itself, which was used for testing in the preliminary stages of the development. Or using a custom script command, `SET_ADVANNET_RESPONSE`, which will be introduced in the following Section 4.3.1. With this command the user will be able to dynamically set the response it wants for each of these endpoints. Therefore, the user will be able to test a myriad of scenarios of a response, in an automated way.

The five legacy endpoints are:

- `GET /status`, configurations of the system.
- `GET /devices/AdvanPay-cf-us-120/reader/parameter/RF_READ_POWER`, power of the antenna.
- `GET /devices/AdvanPay-cf-us-120/inventory`, tags being read.
- `GET /devices/AdvanPay-cf-us-120/start`, enables RFID scanning.
- `GET /devices/AdvanPay-cf-us-120/stop`, disables RFID scanning.

4.2.2 New Request Handlers

Additionally to the five legacy Keonn's API endpoints, two new handlers were created:

- `POST /startpos`
- `POST /response`

The `startpos` endpoint is used as a timer for when a new transaction starts. At this moment this endpoint is used, which allows to track the duration of every operation. On the other hand, the `response` endpoint is related to the custom script command mentioned before, `SET_ADVANNET_RESPONSE`. This is the endpoint that will receive the data send by the Script runner application and change the responses of the AdvanNet endpoints.

4.3 Script runner Commands

Before starting the development for the Script runner application itself, the commands that would be needed for this custom scripting language were first designed. The overall command syntax was defined as:

- **Command Name:** Each command line begins with the command name, which is highlighted in bold. This directly indicates the action that will be executed, such as **START_TRANSACTION**, **ADD_PRODUCT**, or **DELETE_TRANSACTION**.

- **Arguments:** Following the command name, the code line can include necessary arguments. These arguments can be specific files, response parameters, or other types of required data.

4.3.1 Testframework Control Commands

To achieve the desired features the following commands were defined:

- **START_POS**: Initiates the POS1.0 application systems. This command is used to start all necessary services and processes required by POS1.0.
- **STOP_POS**: Terminates the POS1.0 application systems. This command stops all services and processes associated with POS1.0, effectively shutting down the system.
- **SLEEP <duration>**: Pauses the script for a specified **<duration>** in seconds. This command is useful for creating delays within the script, allowing time for processes to complete without affecting the overall operations of POS1.0.
- **SET_ADVANNET_RESPONSE <URI> <code> <bodyFile> <duration> <durationUnit> <delay>**: Configures responses for simulated requests, detailing how the system should react to interactions with POS1.0 based on the specified parameters. This command is essential for testing and verifying how POS1.0 handles different network conditions and request responses.

Regarding the **SET_ADVANNET_RESPONSE** command it can be added that it allows the customization of the response received from a specific request and its duration. The **<URI>** (Uniform Resource Identifier) parameter defines in which endpoint the response should be changed. Additionally, the **<code>** parameter defines the HTTP response that it should return, while the **<bodyFile>** is the body of the response which should point to a file. To define how long this response is intended to be active the parameter **<duration>** is used. Furthermore, the parameter **<durationUnit>** is used to define if the counting units will be done with `Time` or number of iterations (`Count`). The last argument, **<delay>**, sets the duration of how long this response will be active.

This set of commands will enable the user of the Testframework to define any test to their needs without editing any of the source code. Everything can be done through the custom scripting language developed and presented in this section. It can be stated that since no programming knowledge is required to create any of the tests, this tool has a low skill barrier.

4.3.2 POS1.0 Transaction Mode Commands

The commands that are used to replicate a sale in Transaction mode are detailed in this section. These commands will directly interact with the API of the Transaction mode making them the core commands for the black-box testing. A command will be the input, while the user provides the expected output. After that, this output will be compared to the transaction API response for validation. Consider the commands:

- **START_TRANSACTION <transactionFile> <response>**: Initializes a transaction. If **<response>** is specified, the system's response is compared to the file's contents.
- **ADD_PRODUCT <productFile> <response>**: Sends a request to add a product. If **<response>** is provided, it compares the outcome with the expected POS1.0 response.
- **REMOVE_PRODUCT <productFile> <response>**: Issues a request to remove a product. The optional **<response>** parameter allows comparison with the actual POS1.0 response.
- **GET_TRANSACTION <response>**: Retrieves the details of a transaction, comparing it with the specified **<response>** file if provided.
- **EXECUTE_TRANSACTION <response>**: Executes a transaction, verifying the process with provided file.
- **DELETE_TRANSACTION <response>**: Ends a request to delete a transaction, checking the system's response against the expected result.

4.3.3 Example Script

To demonstrate the practical use of the defined scripting commands, Listing 4.1 presents an example of a test script for a standard transaction. Additionally, listings B.1, B.2, B.3 and B.4 (Appendix B) are examples of possible JSON objects and XML elements used for the transactions. Consider the explanation of the example script based on Listing 4.1 line numbers:

1. (`START_POS`) Initiates all necessary services for Advanpay and starts the timer to track the duration of the test. This step ensures that all system components are operational before proceeding with the transaction workflow.
2. (`SLEEP`) Introduces a 10-second pause to allow all services to become fully operational.

```
START_POS
SLEEP 10
START_TRANSACTION transaction.json response.json
SET_ADVANNET_RESPONSE "/inventory" 200 tag.xml 200 Time 0
SLEEP 5
ADD_PRODUCT products.json response.json
GET_TRANSACTION getresponse.json
EXECUTE_TRANSACTION executeresponse.json
DELETE_TRANSACTION deleteTransactionResponse.json
STOP_POS
```

Listing 4.1: Example of custom script for scriptRunner.

3. (`START_TRANSACTION`) Sends an HTTP POST request to the transaction API to initiate a new transaction. The input data is specified in ‘transaction.json‘ with an example in Listing B.1, and the expected output is validated against ‘response.json‘.

4. (`SET_ADVANNET_RESPONSE`) Configures the ‘/inventory‘ endpoint to return a predefined response (‘tag.xml‘ Listing B.2) with an HTTP status code of 200. This response is valid for 200 seconds and is set to simulate the presence of tags after the transaction starts.

5. (`SLEEP`) Introduces a pause to ensure that the response from the ‘/inventory‘ endpoint is updated before the Advanpay system continues querying for tags.

6. (`ADD_PRODUCT`) Sends an HTTP POST request to the transaction API to add a new product (‘product.json‘ Listing B.3) to the active transaction. The system checks whether the product is detected in the ‘/inventory‘ response and compares the result with the expected output in ‘response.json‘.

7. (`GET_TRANSACTION`) Retrieves the current status of the transaction by sending a GET request to the transaction API. The response is compared with ‘getresponse.json‘ (Listing B.4) to ensure the transaction is proceeding as expected.

8. (`EXECUTE_TRANSACTION`) Finalizes the transaction by sending a request to the transaction API. This step triggers an MQTT message to the cloud broker and sends a multicast User Datagram Protocol (UDP) message to exit systems to prevent alarms when the item exits the store. The response is validated against ‘executeresponse.json‘.

9. (`DELETE_TRANSACTION`) Cleans up by deleting the transaction via the transaction API and stops any ongoing checks to the ‘/inventory‘ endpoint. The system compares the received response with ‘deleteTransactionResponse.json‘ to confirm the successful deletion.

10. (`STOP_POS`) Stops all services initiated for the test, marking the end of the test procedure and gathering all results for further analysis.

4.4 Executing the Script runner Application

The required steps for users to clone, compile, and run the Script runner application are presented in Listing 4.2.

```
# Clone the repository
git clone http://sna1-hw-repo.cfsad.com/goliveira/POS1.0.git

#Change to the TestFramework branch
git checkout TestFramework

# Navigate to the application directory
cd TestFramework/webserver/

# Download dependencies (Go modules)
go mod tidy

# Start the web server
go build -o WebServer
./WebServer &

# Download dependencies (Go modules)
cd ..
go mod tidy

#Build scriptRunner
go build -o scriptRunner

# Run the compiled application
./scriptRunner config.cfg SampleScript
```

Listing 4.2: Steps to Clone Compile and Run the Application.

Before running the main application, the web server that replicates the Keonn API, the Advanpay simulator, must be started. In fact, it only needs to be running in the background on the server that will host the TestFramework. With the web server running, the Script Runner needs to be compiled so it can then be executed by entering the following command in the terminal:

```
./scriptRunner config.cfg SampleScript
```

In this command the first argument, `config.cfg`, represents the configuration file that contains all the necessary initial settings, which can be modified as

needed.The second argument, `SampleScript`, contains the custom script that will be executed.

The Listing 4.3 is an example of a configuration file, which will be parsed into the `scriptRunner` as global variables. This will allow to easily change settings and test scripts, without the need to recompile the code.

```
[paths]
pathPos = POS1.0/POS1.0
pathDevMgr = POS1.0/DevMgr/DevMgr
pathEventlogd = POS1.0/Eventlogd/Eventlogd

[communication]

ReaderHost = 192.168.0.247
APIport = 8888
AdvanNetInnerIP = 127.0.0.1
AdvanNetInnerPort = 10000
User = user
Password = password
```

Listing 4.3: Example of config.cfg file for scriptRunner.

4.5 Continuous Integration and Continuous Deployment

The TestFramework is designed to work in two distinct scenarios. Initially, it allows users to manually build and execute tests via the Script runner application. Additionally, it supports automated testing, activated each time a commit is made to the main development branch on GitLab. This automation ensures that system tests are performed with each new commit, verifying that recent changes have not disrupted existing functionalities. The framework is highly customizable, enabling a straightforward integration of new tests regardless of their programming knowledge. The designed CI/CD flowchart for the pipeline is depicted in Figure 4.2.

The tool chosen for CI/CD was Jenkins. The reason for this was that the main disadvantages that can be pointed out to this tool don't apply to Sensormatic's development context. At the time of this project Jenkins was already in use by the development team so the potentially troublesome initial setup was not needed.

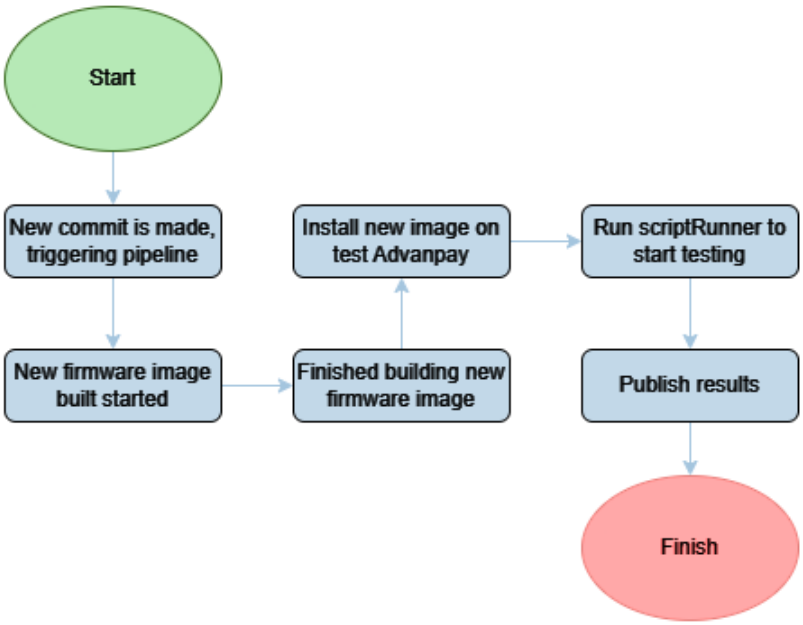


Figure 4.2: Pipeline flowchart.

4.5.1 Firmware Build

There was already a Jenkins job developed by a team member of the firmware team, which builds the firmware of the Advanpay with the applications for POS1.0 and the original Keonn firmware. This will be integrated into this project's pipeline so there isn't replicated work. The new Jenkins job which will handle the pipeline was named `POS1.0_TestFramework` (Figure 4.3).

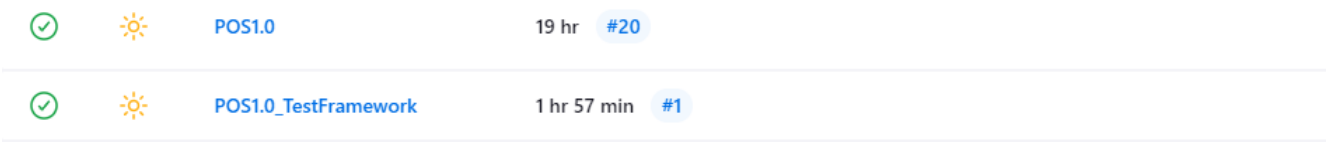


Figure 4.3: Jenkins jobs.

4.5.2 Jenkins Job Trigger

To handle how the pipeline gets triggered, a Jenkins feature (Figure 4.4) was used that relies on a webhook. This mechanism is configured on Gitlab, which will then be able to detect any push events and merge requests that are made.

For a first development stage, only one specific branch will support this method, identified in Figure 4.5. Once it's fully stable and operational, it will be rolled out to all remaining branches.

Figure 4.4: Jenkins push triggers.

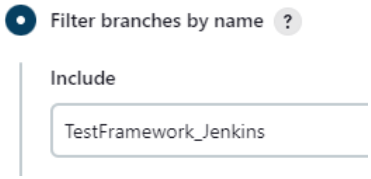


Figure 4.5: Jenkins branch filter.

4.5.3 Installing the New Firmware

The POS1.0 supports firmware upgrades that get triggered though MQTT. This feature proved to be useful for this part of the pipeline. Every Advanpay has a unique identifier known by `Base topic`. This is customizable on the Advanpay webpage as shown in Figure 4.6.

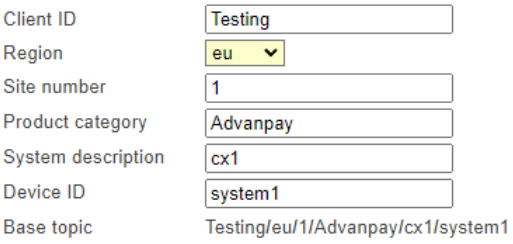


Figure 4.6: Advanpay base topic.

In order to initiate the firmware upgrading process, users must publish to the following MQTT broker topic, where `basetopic` is the unique identifier for the Advanpay device:

`"'basetopic'/DevMgr/manage/oobFirmwareUpdateReq".`

Moreover, the base topic for the Advanpay test, which will conduct all of the tests, was defined on Figure: 4.6 as:

`Testing/eu/1/Advanpay/cx1/system1`

Thus, the full topic for the firmware upgrade becomes:

`Testing/eu/1/Advanpay/cx1/system1/DevMgr/manage/oobFirmwareUpdateReq`

To proceed, the payload of the message needs to be formatted in JSON with a single object named Uniform Resource Identifier (URI). This allows to point to a specific source of the firmware image. The Jenkins jobs that built the firmware image hosts this firmware at the URI:

`http://10.162.192.95:8080/userContent/Advanpay_120_image_TestFramework/`

Finally, all it remains to be done is the Jenkins job to publish a MQTT message to the mentioned topic with the URI payload using the `mosquitto\_pub` application installed on the machine. The firmware upgrade process usually takes around 2 minutes.

4.5.4 Running scriptRunner

Once the firmware upgrade is complete, the Advanpay will send a MQTT message to the topic `"basetopic/DevMgr/oobFirmwareUpdateStatus"`. The Jenkins job will wait for this message with a timeout of 10 minutes, if it doesn't get it, the pipeline will fail. This will be used as a signal for the `scriptRunner` to start.

The `scriptRunner` application will run using a `sampleScript` that is loaded on the directory by default. This script file can be edited anytime by the user to adapt the type of tests being performed.

4.5.5 Publishing the Results

Once the script finishes it will publish the results in two different ways. Firstly there is a log folder where all tests results gets stored until 100 entries. Secondly, it will automatically email the results to the Jenkins user configured as the owner/maintainer of the project.

4.6 Discussion

This chapter presented the methodology and implementation of a testing framework designed to simulate the POS1.0 application for black-box testing without requiring access to the Keonn API. The whole testing approach was meant to be integrated

into a pipeline so that the testing process could be automated. The dissertation highlighted the architecture of the testing environment, which includes elements like the AdvanPay simulator, the Script runner, and CI/CD integration via Jenkins. By implementing this framework dependencies on external systems were reduced, allowing for easier and more effective testing. The next chapter will validate the effectiveness of the TestFramework tool through practical test scenarios.

Chapter 5

Results Validation

This chapter explores two distinct types of testing methodologies that were employed to validate the proposed system. The first is user testing, in which users create custom scripts to verify specific features or scenarios of interest. The second one is the automation of tests, where the procedure is automatically triggered by commits to a GitLab branch, leading to the corresponding pipeline to run a predefined set of tests.

Both user and automated testing provide valuable insights, each suited to different scenarios. User testing allows for more specific, customized tests, which are useful for handling unique or complex situations. In contrast, automated testing is ideal for covering common testing scenarios, ensuring that no functionality is broken with each new commit to a specific branch.

5.1 User Testing

To set up the testing environment, all the necessary configuration files must be defined, starting with the POS1.0 configuration file, which is accessible through the URI `http://READER_IP:3161/pay.html`. This address gives access to the web page shown in Figure 5.1

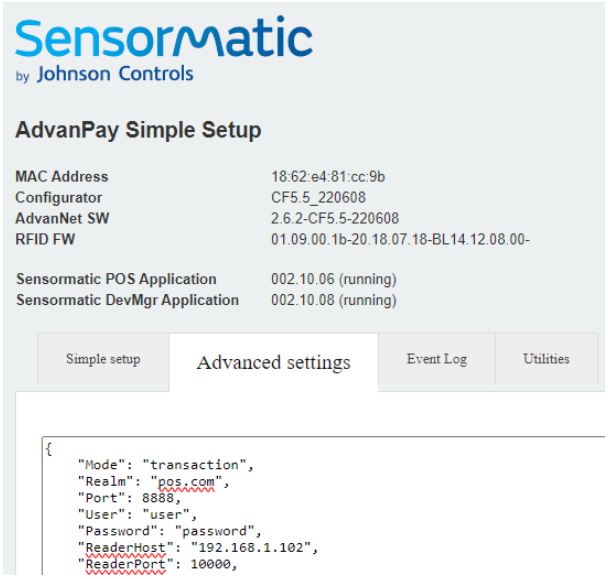


Figure 5.1: Configuration page for Advanpay (cropped settings).

There are three parameters to consider in order to adapt the default configurations to the current testing environment:

- **Mode**: The selected mode needs to be Transaction.
- **ReaderHost**: This is usually 127.0.0.1 since the AdvanNet API is hosted on the device. It will have to point to the machine that is running the AdvanNet simulator, in this case 192.168.1.102.
- **ReaderPort**: This is usually 3161 which is the port that Keonn defined for the AdvanNet API. In this case, the AdvanNet simulator application uses the port 10000.

After applying the settings, the POS1.0 applications will reboot, loading the new configurations. The POS will then wait for the AdvanNet API to respond. In the next step, the AdvanNet simulator is started using the steps provided in Section 4.4. The last step is to edit the `scriptRunner` configuration file, shown in Listing 5.1. The content of this file is used for:

- **ReaderHost**: In the `scriptRunner`, the `ReaderHost` references where the POS1.0 is running, pointing to the Advanpay IP.
- **ReaderPort**: This is the port where the transaction API is being hosted, which is on the POS1.0 at the port 3161.
- **AdvanNetInnerIP**: The IP of the AdvanNet simulator. Due to running on the same machine the `scriptRunner` and the simulator, the localhost IP is used.

```
[communication]

ReaderHost = 192.168.1.100
APIport = 8888
AdvanNetInnerIP = 127.0.0.1
AdvanNetInnerPort = 10000
User = user
Password = password
```

Listing 5.1: Configuration file for scriptRunner.

- **AdvanNetInnerPort**: The port of the AdvanNet simulator, i.e., the simulator will listen the port 10000.
- **User**: The user for the transaction API. The `user` is a placeholder for the actual user.
- **Password**: The password for the transaction API. The `password` is a placeholder for the actual password.

After this procedure, the environment is considered to be ready to start the tests. The user performing the tests must now provide the scripts with the tests themselves, and then run the `sampleScript` application.

Since the system is a black-box tester, the user only has to give inputs, and then wait for specific outputs. If these outputs are not as expected, then the test will be considered to have failed. This tests are completed within seconds using the TestFramework. To replicate these tests without the TestFramework, the tester would need physical access to the Advanpay device to scan the tags and be constrained to the Keonn API reducing the scope of the tests. Additionally, analyzing all the responses would be time consuming and subject to human error.

5.1.1 First Test - Regular Transaction

For the first test, a regular transaction with the POS1.0 was simulated, where only one product was sold, and no error was planned to happen during the simulation.

The Listing 5.2 provides the sample script used for the tests. As a result, running the script `./scriptRunner config.cfg SampleScript`, outputs a log file. In Appendix C, the Listing C.1 provides a cropped version (due to too verbose) of the log file that results from this test. It can be verified that every expected output matched with the inputs provided on the HIL simulation, therefore, the black-box tests passed.

```
START_POS
SLEEP 10
START_TRANSACTION transaction.json
GET_TRANSACTION getTransactionResponse_empty.json
SET_ADVANNET_RESPONSE /inventory 200 tag.xml 200 Time 0
SLEEP 1
GET_TRANSACTION getTransactionResponse_unmatched.json
ADD_PRODUCT products.json
GET_TRANSACTION getTransactionResponse_singulated.json
EXECUTE_TRANSACTION
DELETE_TRANSACTION deleteTransactionResponse.json
STOP_POS
```

Listing 5.2: Test of a regular transaction.

5.1.2 Second Test - Failed Transaction

For this test a faulty build of the POS application was compiled and installed on the Advanpay device. In this case the introduced fault was that when a product is requested to be added to an ongoing transaction, the Transaction API responds indicating that the product was added, but internally, it fails to do so.

In this case, running the same test as before (Listing 5.2) using the script `./scriptRunner config.cfg SampleScript`, produces the log files shown in Listing C.2, in Appendix C.

The test of a regular transaction failed, and the issue was quickly identified by the tester, including where it occurred. Upon analyzing the logs, it can be concluded that the products failed to be added to the transaction. Listing 5.3 highlights the logs ending message delivered to the user.

```

...

Response doesn't match expected response
exit status 3

Test FAILED:  Response of "GET_TRANSACTION" command
doesn't match expected response
```

Listing 5.3: Outcome of a failed test.

5.1.3 Third Test - Simulating Changes to the Keonn API

One of the advantages of how the `TestFramework` was implemented is that the user can modify the responses from the simulated Keonn API using the command `SET_ADVANNET_RESPONSE`.

For this next test, let's use an example where the project owner requests a change to the Keonn API, so that the `/inventory` responses are provided in JSON format instead of XML.

This enables that before Keonn makes the necessary changes to their firmware and provide the new version, the Sensormatic teams can start to develop and test a new system specification. This feature addresses one of the main challenges posed to the development team until now. As a result, the POS application can also be early tested regarding the new specification.

To conduct this test it will be necessary to run the default script, but with a change in the arguments of the `SET_ADVANNET_RESPONSE` command. In this case, the `tag.xml` file will be changed to a `tag.json` file (line 5 of Listing 5.4) formatted using the JSON schema provided by the project owner.

```
1  START_POS
2  SLEEP 10
3  START_TRANSACTION transaction.json
4  GET_TRANSACTION getTransactionResponse_empty.json
5  SET_ADVANNET_RESPONSE /inventory 200 tag.json 200 Time 0
6  SLEEP 1
7  GET_TRANSACTION getTransactionResponse_unmatched.json
8  ADD_PRODUCT products.json
9  GET_TRANSACTION getTransactionResponse_singulated.json
10 EXECUTE_TRANSACTION
11 DELETE_TRANSACTION deleteTransactionResponse.json
12 STOP_POS
```

Listing 5.4: Testing changes to Keonn API.

Once again, running the script `./scriptRunner config.cfg SampleScript` the new firmware version will be simulated, testing what Keonn would provide in the future. The output file yield by this procedure is presented in Listing 5.5. Note that this log output was heavily cropped due to the verbose nature of the logging routines.

This test failed since there was no change from the POS application side to support the format change. With the POS application version under test, the tags can only be "read" if they are in XML format. Therefore, the expected response will not as expected.

```
Execute Get Transaction
URL:> https://192.168.1.100:8888/transaction?unmatched=true

Body: {
  "Antenna": -1,
  "EmpId": "987654321",
  "Id": "123456789",
  "Mode": "Sale",
  "PosId": "12344478",
  "Products": null,
}
Expected Response: {
  "Antenna": -1,
  "EmpId": "987654321",
  "Id": "123456789",
  "Mode": "Sale",
  "PosId": "12344478",
  "Products": null,
  "Unmatched": [
    "30801191a50200602a24c27d"
  ]
}

Response doesn't match expected response
exit status 3

Test FAILED:  Response of "GET_TRANSACTION" command
doesn't match expected response
```

Listing 5.5: Test output of Keonn's API changes.

Now the developer is ready to start implementing the new JSON format and test it, as if Keonn had already provided this new firmware. This type of scenarios is one of the main reasons the TestFramework was developed, to reduce dependency on third parties.

5.2 Automatic Testing

To initiate automatic testing an Advanpay device needs to be setup on the same network as the machine running Jenkins. This Advanpay will be used for all the automatic tests. It is recommended to set the IP to a static one (Figure 5.2), so the IP address does not need to be updated in the Jenkins job in case it changes.

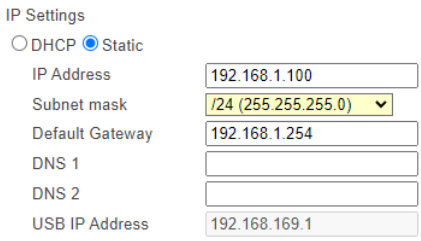


Figure 5.2: Changing IP to static.

The `testFramework` tool was cloned into the machine running Jenkins and the usual setup will be followed as described in Section 4.4.

For this test, the second script from Subsection 5.1.2 will be reused, and the same faulty firmware will be built. The only requirement to trigger the test will be a commit to the branch `TestFramework_Jenkins`, with no needed input from the user, using the webhooks set up to listen for specific events. See Section 4.5.2 for more details.

The testing pipeline was triggered by the commit that was done, as depicted in Figure 5.3, and it began building the Advanpay firmware image with the faulty code.

Figure 5.3: Commit that triggered the pipeline.

Once the build was finished, the firmware image (Figure 5.4) was hosted at the following URL:

`http://192.168.222.160:8080/userContent/`
`Advanpay_120_image_TestFramework/12_09_2024_CF5.5_signed.zip`

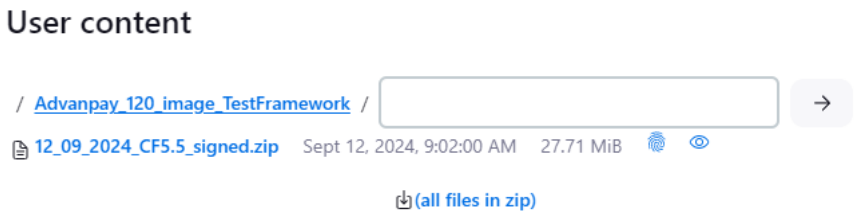


Figure 5.4: Firmware image.

The Jenkins job sent an MQTT message to the broker of the Advanpay device with the following parameters:

```
Topic: 0001/eu/1/1200/advanpay/1/DevMgr/manage/oobFirmwareUpdateReq
Message: {"uri": "http://192.168.222.160/AdvanPay/12_09_2024_CF5.5
_signed.zip"}
```

This initiated the firmware update of the newly compiled firmware image. The Jenkins job, as configured, waited for 10 minutes to receive confirmation. The firmware update was completed upon receiving the following MQTT message:

```
Topic: 0001/eu/1/1200/advanpay/1/DevMgr/oobFirmwareUpdateStatus
Message: {"status":"Update Completed Successfully","fileSize":
29053915,"error_code":0,"error_msg":"","timestampMsec":1726128526518}
```

The Jenkins job, after getting this message on the topic, executed the command to start `scriptRunner`, which performed the black-box tests with the simulation of a regular transaction. A cropped version of the obtained results are presented in Listing 5.6.

```
1 ...
2 ----------------------------------------------------------------------

3 Response doesn't match expected response
4 exit status 3
5
6 Test FAILED:  Response of "GET_TRANSACTION" command
7 doesn't match expected response
8 ----------------------------------------------------------------------
```

Listing 5.6: Automatic testing outputs.

The results are identical to those of the second user test, with the advantage that these tests will automatically happen every time a new commit is made. With this

automatic tests implementation, every commit will trigger a black-box test of the full steps of a transaction. Several different tests can be configured for the automatic pipeline, to test as much scenarios as possible.

5.3 Discussion

The results of both user and automated testing validated the effectiveness of the TestFramework. User tests demonstrated how the framework could detect issues during a transaction by analyzing the logs produced by the `scriptRunner` application. The ability to simulate changes made to the Keonn API was also done, allowing developers to adjust to any future changes of the API. Automatic testing was implemented using Jenkins via a CI/CD pipeline. These automatic tests replicated user tests without any intervention and identified the same failure as the user tests.

Overall, this chapter highlighted the significant improvements in testing capabilities due to the combination of manual flexibility and automated reliability, ultimately reducing dependencies on third-party systems and enhancing the robustness of the Advanpay testing environment.

Chapter 6

Conclusion

This thesis has explored the development and integration of a HIL system for a POS RFID reader with black-box tests, named TestFramework. The creation of a custom scripting language was made, which enables users to design their own test scenarios. This custom language enables the simulation of a transaction while at the same time, allowing flexibility to perform a wide range of operations. This adaptability will vastly help to increase the system's overall robustness and quality.

To complete the TestFramework, an AdvanNet simulator named Advanpay simulator, was also implemented. The simulator serves as an API for the transactions in the POS. This simulator replicates the API developed by a third-party, Keonn, allowing access to RFID tags being read, configuration settings, reader power adjustments and more. By providing this level of simulation, the system reduces dependency on third parties during testing, creating a controlled environment for both manual and automated testing. The automated testing uses pre-defined scripts which can be changed, with each code change running the pre-defined scripts ensuring constant validation. The manual testing benefits from the custom scripting capabilities to simulate unique scenarios while automated testing provides constant validation.

In conclusion, the development of the TestFramework tool has shown very promising results as demonstrated in Chapter 5. This approach proved to have the capability to detect issues early in the development cycle and the ability to heavily customize different type of tests. While the project has achieved significant milestones, there are still limitations. Manual testing, even though very customizable,

is time-intensive to start with, due to the fluency period needed for a manual tester to work efficiently with the TestFramework tool. The automated testing has the advantage that it only needs to be setup once and it will keep performing tests, but the configured tests may not account for every edge case. For that purpose, it relies on users to maintain and update the tests. The next short term step for this project, is to be introduced to the QA team, so it can start being used in future firmware releases.

6.1 Future Work

The future development for a next version for this project, is to make it as accessible and easy to create a test as possible. Currently there is still a learning curve where the user needs to properly learn how the tool works to create the testing scripts. To do this, three different improvements were designed but not implemented yet.

The first one consists of integrating a new feature on the scriptRunner, where the user has a Graphical User Interface (GUI) that will help the user, by providing all the info needed for each command and also create the testing scripts on the GUI itself. The second improvement would be focused on automatic testing. Currently, to implement new automatic tests the user has to access the machine that has the Jenkins job using SSH to add the script to the machine. The goal is to implement a web page, where the user can add the scripts it wants to run on the automatic tests, to the host machine. Heavily simplifying the process of adding new tests to the automatic tests. The third one would be to join the two features together. Once a script was made using the GUI, the user would have the option to directly import the test to the automatic tests library using the same web page.

With the planned improvements completed, it would lead to a likely reduced learning curve for new users and a higher efficiency for experienced ones. These developments are seen as the next natural step since user experience was the main drawback identified with the current implementation. Furthermore, its also envisioned that in the mid term the system should integrate white-box testing in order to ensure the best possible system for Sensormatic's clients.

References

[1] Sensormatic by Johnson Controls, "About us - sensormatic." Available at https://www.sensormatic.com/about-us. [Cited on page 2]

[2] Keonn Technologies, S.L., "Keonn - seamless rfid solutions." Available at https://keonn.com. [Cited on page 2]

[3] M. Liukkonen, "Rfid technology in manufacturing and supply chain," *International Journal of Computer Integrated Manufacturing*, vol. 28, no. 8, pp. 861–880, 2015. [Cited on page 6]

[4] NASA Official for Science, "Anatomy of an electromagnetic wave." Available at https://science.nasa.gov/ems/02_anatomy/, 2023. [Cited on page 6]

[5] BYJU'S, "Electromagnetic radiation." Available at https://byjus.com/physics/electromagnetic-radiation/, 2024. [Cited on pages ix and 6]

[6] International Organization for Standardization (ISO), "Iso/iec 18000-6:2013 - information technology — radio frequency identification for item management — part 6: Parameters for air interface communications at 860 mhz to 960 mhz general." Available at https://www.iso.org/standard/59644.html, 2013. [Cited on page 6]

[7] Y. Zhang, K. Yemelyanov, X. Li, and M. G. Amin, "Effect of metallic objects and liquid supplies on rfid links," in *2009 IEEE Antennas and Propagation Society International Symposium*, pp. 1–4, 2009. [Cited on page 7]

[8] Nordic ID, "Smart rfid tags enables reading next to metals and liquids." Available at https://www.nordicid.com/resources/blog/smart-rfid-tags-enable-reading-next-to-metals-and-liquids/, 2020. [Cited on pages ix and 7]

[9] V. Chawla and D. S. Ha, "An overview of passive rfid," *IEEE Communications Magazine*, vol. 45, no. 9, pp. 11–17, 2007. [Cited on pages ix and 7]

[10] P. Nikitin and K. Rao, "Theory and measurement of backscattering from rfid tags," *IEEE Antennas and Propagation Magazine*, vol. 48, no. 6, pp. 212–218, 2006. [Cited on page 7]

[11] S. Amsler, "What is rfid and how does it work?." Available at https://www.techtarget.com/iotagenda/definition/RFID-radio-frequency-identification, 2020. [Cited on pages 8 and 9]

[12] GS1 Sweden AB, "Ean-13 and ean-8 gs1." Available at https://gs1.se/en/standards-and-services/ean-13-and-ean-8/. [Cited on page 9]

[13] GS1 Sweden AB, "Upc-a and upc-e gs1." Available at https://gs1.se/en/standards-and-services/upc-a-and-upc-e/. [Cited on page 9]

[14] GS1 Norway, "Guideline for unique identification of products with sgtin (serialized gtin). labelling with gs1 datamatrix barcode and tagging with epc / rfid gen 2 uhf rfid tags." Available at https://www.gs1.org/docs/technical_industries/, 2018. [Cited on page 9]

[15] X. Gong and W. Yi, "Epc's competitive advantages in enhancing the impact of commercial enterprises and china's strategy to response," *2010 International Conference on Logistics Systems and Intelligent Management (ICLSIM)*, vol. 2, pp. 1076–1080, 2010. [Cited on page 10]

[16] H. Zhao, J. Sun, and K. Wang, "Research of extensibility on barcode in epc network," *Transaction on Control and Mechanical Systems*, vol. 1, 2012. [Cited on page 10]

[17] The MathWorks, Inc, "Basics of hardware-in-the-loop simulation - matlab and simulink." Available at https://www.mathworks.com/help/simscape/ug/what-is-hardware-in-the-loop-simulation.html. [Cited on page 10]

[18] OPAL-RT Technologies, "Hardware-in-the-loop hil simulation opal-rt." Available at https://www.opal-rt.com/hardware-in-the-loop/. [Cited on pages ix and 11]

[19] J. Bélanger, P. Venne, and J.-N. Paquin, "The what, where and why of real-time simulation," *Journal of Simulation and Modeling*, vol. 37, pp. 1–48, 2022. [Cited on page 11]

[20] M. B. Ayed, L. Zouari, and M. Abid, "Software in the loop simulation for robot manipulators," *Engineering, Technology & Applied Science Research*, vol. 7, pp. 2017–2021, 2017. [Cited on page 11]

[21] M. Ade, P. Wauters, R. Lauwereins, M. Engels, and J. Peperstraete, "Hardware-software trade-offs for emulation," in *Proceedings of IEEE Workshop on VLSI Signal Processing*, pp. 224–232, 1993. [Cited on page 11]

[22] P. H. Cheung, K. Hao, and F. Xie, "Component-based hardware/software co-simulation," in *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pp. 265–270, 2007. [Cited on page 11]

[23] J. Reitz, A. Gugenheimer, and J. Roßmann, "Virtual hardware in the loop: Hybrid simulation of dynamic systems with a virtualization platform," in *2020 Winter Simulation Conference (WSC)*, pp. 1027–1038, 2020. [Cited on page 12]

[24] A. Bernardes, C. de Farias, R. S. Rodrigues, A. Murilo, R. V. Lopes, and S. Avila, "Low-cost hardware-in-the-loop platform for embedded control strategies simulation," *IEEE Access*, vol. 7, pp. 111499–111512, 2019. [Cited on pages ix and 12]

[25] A. Isma, A. Kamel, and S. Abderrahmane, "Hardware in the loop simulation for robot navigation with rfid," in *2022 7th International Conference on Image and Signal Processing and their Applications (ISPA)*, pp. 1–6, 2022. [Cited on pages ix, 12, and 13]

[26] M. E. Khan, "Different approaches to white box testing technique for finding errors," *International Journal of Software Engineering and its Applications*, vol. 5, pp. 1–14, 2011. [Cited on page 14]

[27] T. Ostrand, *White-Box Testing*. John Wiley & Sons, 2002. [Cited on page 14]

[28] Imperva, "What is black box testing | techniques & examples | imperva." Available at `https://www.imperva.com/learn/application-security/black-box-testing/`, 2024. [Cited on pages ix and 14]

[29] A. Roman, *Black-Box Testing Techniques*, pp. 25–60. Cham: Springer International Publishing, 2018. [Cited on page 14]

[30] M. E. Khan and F. Khan, "A comparative study of white box, black box and grey box testing techniques," *International Journal of Advanced Computer Science and Applications*, vol. 3, pp. 12–15, 2012. [Cited on page 15]

[31] B. Qu, C. Nie, B. Xu, and X. Zhang, "Test case prioritization for black box testing," in *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, vol. 1, pp. 465–474, 2007. [Cited on page 16]

[32] Red Hat, Inc., "What is ci/cd?." Available at `https://www.redhat.com/en/topics/devops/what-is-ci-cd`, 2023. [Cited on page 16]

[33] N. Singh, "Ci/cd pipeline for web applications," *International Journal for Research in Applied Science and Engineering Technology*, vol. 11, pp. 5218–5226, 5 2023. [Cited on page 16]

[34] John | Cloud4Y, "What is ci/cd? | cloud4y." Available at `https://www.cloud4y.ru/en/blog/what-is-ci-cd/`, 2023. [Cited on pages ix and 16]

[35] L. E. Gryzun, Y. Skorin, and I. Detochenko, "Choosing a continuous integration tool for software automated testing," *Bulletin of Kharkov National Automobile and Highway University*, 2022. [Cited on page 16]

[36] Jenkins, "Jenkins handbook." Available at `https://www.jenkins.io/doc/book/`. [Cited on page 17]

[37] R. Leszko, *Continuous Delivery with Docker and Jenkins*. Packt Publishing, 2017. [Cited on page 17]

[38] M. S. Arefeen and M. Schiller, "Continuous integration using gitlab," *Undergraduate Research in Natural and Clinical Sciences and Technology Journal*, vol. 3, 1 2019. [Cited on page 18]

[39] GitLab B.V., "Get started with gitlab ci/cd | gitlab." Available at `https://docs.gitlab.com/ee/ci/`. [Cited on page 18]

[40] B. De, *Introduction to APIs*, pp. 1–14. Berkeley, CA: Apress, 2017. [Cited on pages ix and 19]

[41] C. Sun, X. Zeng, C. Sun, H. Si, and Y. Li, "Research and application of data exchange based on json," in *2020 Asia-Pacific Conference on Image Processing, Electronics and Computers (IPEC)*, pp. 349–355, 2020. [Cited on page 20]

[42] World Wide Web Consortium (W3C), "Extensible markup language (xml) 1.0 (fifth edition)." Available at `https://www.w3.org/TR/REC-xml/`, 2008. [Cited on page 20]

[43] J. Stylos and B. Myers, "Mapping the space of api design decisions," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*, pp. 50–60, 2007. [Cited on page 20]

[44] R. Pike, "Go at google: Language design in the service of software engineering." Available at `https://go.dev/talks/2012/splash.article`, 2012. [Cited on page 21]

[45] S. S. Brimzhanova, S. K. Atanov, M. Khuralay, K. S. Kobelekov, and L. G. Gagarina, "Cross-platform compilation of programming language golang for raspberry pi," in *Proceedings of the 5th International Conference on Engineering and MIS*, ICEMIS '19, (New York, NY, USA), Association for Computing Machinery, 2019. [Cited on page 21]

[46] Golang Docs, "Building applications in golang." Available at `https://golangdocs.com/building-applications-in-golang`, 2024. [Cited on page 21]

[47] Golang Cookbook, "Cross compiling in go." Available at `https://golangcookbook.com/chapters/running/cross-compiling/`, 2024. [Cited on page 22]

[48] The Go Authors, "Package net/http." Available at `https://pkg.go.dev/net/http`, 2024. [Cited on page 22]

[49] A. Kirszenberg, A. Martin, H. Moreau, and E. Renault, "Go2pins: a framework for the ltl verification of go programs," in *Proceedings of the 27th International SPIN Symposium on Model Checking of Software*, pp. 1–11, ACM, 2021. [Cited on pages ix, 22, and 23]

[50] A. L. Gijs Kant, J. v. d. P. Jeroen Meijer, and T. v. D. Stefan Blom, "Ltsmin: High-performance language-independent model checking," in *Proceedings of the Formal Methods and Tools Conference*, University of Twente, 2015. [Cited on page 23]

[51] A. Duret-Lutz, "Manipulating ltl formulas using spot 1.0," in *Proceedings of the ATVA*, p. 15, Springer, 2013. [Cited on page 23]

[52] I. Kang, I. Lee, and Y.-S. Kim, "An efficient state space generation for the analysis of real-time systems," *IEEE Transactions on Software Engineering*, vol. 26, no. 5, pp. 453–477, 2000. [Cited on page 23]

[53] M. Fowler and R. Parsons, *Domain-Specific Languages*. Addison-Wesley Professional, 2010. [Cited on page 24]

[54] V. Brun, "How to write syntax tree-based domain-specific languages in go." Available at `https://betterprogramming.pub/how-to-write-syntax-tree-based-domain-specific-languages-in-go-\b15537f0d2f3`, 2022. [Cited on page 24]

[55] M. Riedl, R. Schneckenhaus, and M. Meier, "Scripting language for distributed application design," in *2012 9th IEEE International Workshop on Factory Communication Systems*, pp. 95–98, 2012. [Cited on pages ix and 25]

[56] HiveMQ, "Mqtt essentials - all core concepts explained." Available at `https://www.hivemq.com/mqtt/`. [Cited on page 25]

[57] C. BasuMallick, "Mqtt working, types, applications." Available at `https://www.spiceworks.com/tech/iot/articles/what-is-mqtt/`, 2022. [Cited on pages ix and 26]

[58] E. Longo, A. E. C. Redondi, M. Cesana, and P. Manzoni, "Border: A benchmarking framework for distributed mqtt brokers," *IEEE Internet of Things Journal*, vol. 9, pp. 17728–17740, 9 2022. [Cited on pages xi and 26]

[59] R. Light, "Mosquitto man page | eclipse mosquitto." Available at `https://mosquitto.org/man/mosquitto-8.html`. [Cited on page 27]

[60] R. Light, "mosquitto_pub man page | eclipse mosquitto." Available at `https://mosquitto.org/man/mosquitto_pub-1.html`. [Cited on page 27]

[61] R. Light, "mosquitto_sub man page | eclipse mosquitto." Available at `https://mosquitto.org/man/mosquitto_sub-1.html`. [Cited on page 27]

[62] Keonn Technologies, S.L., "Advanpay-120 : Confined reading area rfid reader for points of sale." Available at `https://keonn.com/systems-product/advanpay-120/`. [Cited on page 29]

[63] BeagleBoard, "Beaglebone black." Available at `https://www.beagleboard.org/boards/beaglebone-black`, 2024. [Cited on page 30]

[64] Texas Instruments Incorporated, "Am3517, am3505 sitara processors 1 device summary - datasheet." Available at `https://www.ti.com/lit/ds/symlink/am3505.pdf`, 2009. [Cited on page 30]

[65] JADAK, "Thingmagic rfid products." Available at `https://www.jadaktech.com/products/thingmagic-rfid/`. [Cited on page 30]

[66] GS1 EPCglobal, "Low level reader protocol (llrp) standard version 1.1." Available at `https://www.gs1.org/standards/epc-rfid/llrp/1-1-0`, 2010. [Cited on page 32]

[67] Mozilla Foundation, "Http request methods - http | mdn." Available at `https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods`. [Cited on page 35]

[68] International Organization for Standardization (ISO), "Iso - iso 8601 — date and time format." Available at `https://www.iso.org/iso-8601-date-and-time-format.html`. [Cited on page 35]

[69] Mozilla Foundation, "Http response status codes - http | mdn." Available at `https://developer.mozilla.org/en-US/docs/Web/HTTP/Status`. [Cited on page 35]

Appendix A

Full Advanpay Specifications

Air Protocol Interface	EPCglobal UHF Class 1 Gen 2 / ISO 18000-6C
Frequency	FCC (NA, SA) (917.4 – 927.2) MHz ETSI (EU) (865.6 - 867.6) MHz TRAI(India) (865 - 867) MHz KCC (Korea) (917 – 923.5) MHz MIC (Japan) (916.9 – 923.4) MHz ACMA (AU) (920 – 926) MHz NZ (New Zealand) (922 - 927) MHz SRRC-MII (P.R.China) (920.125 – 924.875) MHz MY (Malaysia) (919.0 – 923.0) MHz ID (Indonesia) (923.0 – 925.0) MHz PH (Philippines) (918.0 – 920.0) MHz TW (Taiwan) (922.0 – 928.0) MHz MO (Macao) (920.0 – 925.0) MHz RU (Russia) (866.0 – 868.0) MHz SG (Singapore) (920.0 – 925.0) MHz VN (Vietnam) (866.0 – 869.0) MHz TH (Thailand) (920.0 – 925.0) MHz AR (Argentina) (915.0 – 928.0) MHz HK (Hong Kong) (865.0 – 868.0) MHz BD (Bangladesh) (925.0 – 927.0) MHz Brazil (917.4 – 927.2) MHz by using channel selection Chile(916 – 928) MHz by using channel selection Peru (917.4 – 927.2) MHz by using channel selection Taiwan (922.600 – 927.2) MHz by using channel selection Open Region (859 – 873) MHz and (915 – 930) MHz (by using channel selection)
RF Power	Programmable from 0 dBm to +27 dBm in 0.5 dBm steps
RF Antenna	Integrated circular polarized antenna. RF field is confined to avoid reading unwanted tags.
Data communications	Ethernet: IEEE 802.3 up to 100 Mbps Type B USB HID to emulate barcode reader Console USB (USB micro Type-B connector) Maintenance port Ethernet point-to-point over USB (USB micro Type-B connector)
Power supply	Power Over Ethernet (PoE) • Supports IEEE 802.3af (Type I) and IEEE 802.3at (Type II) • Power consumption: Class 34 • Isolated from Ethernet cable Ratings & Tolerances PSE Type 1: 48 V (- 4 V / +9 V) Maximum power: 15.4 W PSE Type 2: 56 V (- 6 V / +1 V) Maximum power: 30 W On-board battery for RTC chip (CR2032)
On-board actuators	Buzzer 2 Watt loudspeaker (only available in CF versions)
LED indicators	A three-color LED for indicating the active operation mode: • Payment mode (green) • Return mode (red) • Read-only mode (blue)
Compatibility with software applications	Can be easily integrated with any application software, through keyboard wedge
Power consumption	Idle consumption < 2.5 W Default consumption (@10 dBm) < 7 W Max consumption (@27 dBm) < 9 W
Temperature range	Operating temperature -20 °C to +50 °C5 Storage temperature -30 °C to +60 °C
Dimensions (table flush mount)	200 x 200 x 62 mm (7.87 x 7.87 x 2.44 inches)
Dimensions (under table mount)	180 x 180 x 56 mm (7.09 x 7.09 x 2.20 inches)
Dimensions (desktop mount)	200 x 200 x 78 mm (7.87 x 7.87 x 3.07 inches)
Weight (table flush mount)	935 g (0.021 lb)
Weight (under table mount)	820 g (0.018 lb)
Weight (desktop mount)	1330 g (2.932 lb)

Table A.1: Full Advanpay device specifications

Appendix B

Example Files in a Transaction

```
{
    "Id": "123456789",
    "EmpId": "987654321",
    "PosId": "12344478",
    "Mode": "Sale"
}
```

Listing B.1: transaction.json example used on custom script.

```
<inventory>
  <script/>
  <type>inventory</type>
  <ts>1478205424479</ts>
  <status>OK</status>
  <msg-version>2.3.0</msg-version>
  <op>inventory</op>
  <data>
  <advanNetId>AdvanNet-instance-18:62:e4:81:cc:9b--1</advanNetId>
  <deviceId>AdvanPay-cf-us-120</deviceId>
  <inventory>
    <class>INVENTORY</class>
    <deviceId>AdvanPay-cf-us-120</deviceId>
    <size>1</size>
```

```
15      <items>
16      <item>
17          <class>READ_EVENT</class>
18          <epc>080000000000000057fdebd50e230403</epc>
19          <ts>1478205424475</ts>
20          <deviceId>AdvanPay-cf-us-120</deviceId>
21          <data>
22            <class>TAG_DATA</class>
23            <hexepc>080000000000000057fdebd50e230403</hexepc>
24            <props>
25              <prop>TIME_STAMP:1478205424475</prop>
26              <prop>RSSI:-50</prop>
27              <prop>RF_PHASE:174</prop>
28              <prop>ANTENNA_PORT:1</prop>
29              <prop>MUX1:0</prop>
30              <prop>MUX2:0</prop>
31              <prop>FREQ:919566</prop>
32              <prop>GPI:[0,0,0,0]</prop>
33            </props>
34          </data>
35      </item>
36      </items>
37    </inventory>
38  </data>
39  </inventory>
```

Listing B.2: tag.xml example used on custom script.

```
1  {
2      "Products": [
3          "492140707763",
4          "492140707764"
5      ]
6  }
```

Listing B.3: products.json example used on custom script.

```
{
  "Id": "123456789",
  "EmpId": "987654321",
  "PosId": "12344478",
  "Mode": "Sale",
  "Antenna": -1,
  "Start": "2021-05-28T18
      :04:13.971686258+01:00",
  "Products": [
    {
      "Id": "492140707764",
      "Status": "Missing"
    }
  ]
}
```

Listing B.4: getResponse.json example used on custom script.

Appendix C

Testing Logs

```
1  ---------------------------------------------------------------------

2  Execute START_POS args:  []
3  Starting executables
4  2024/09/02 18:17:48 Running command and waiting for it to finish
       ...
5  2024/09/02 18:17:48 Command finished with error: exit status 6
6  Response: {}
7
8  ---------------------------------------------------------------------

9
10 Execute SLEEP  args:  [10]
11
12 ---------------------------------------------------------------------

13
14 Execute StartTransaction args:  [transaction.json]
15 Contents of file: {
16     "Id": "123456789",
17     "EmpId": "987654321",
18     "PosId": "12344478",
19     "Mode": "Sale"
20 }
21
```

```
22 ------------------------------------------------------------------

23
24 Execute Get Transaction
25 URL:> https://192.168.1.100:8888/transaction?unmatched=true
26 Body: {
27   "Antenna": -1,
28   "EmpId": "987654321",
29   "Id": "123456789",
30   "Mode": "Sale",
31   "PosId": "12344478",
32   "Products": null
33 }
34 Expected Response: {
35   "Antenna": -1,
36   "EmpId": "987654321",
37   "Id": "123456789",
38   "Mode": "Sale",
39   "PosId": "12344478",
40   "Products": null
41 }
42 Response matches the expected response
43
44 ------------------------------------------------------------------

45
46 Execute SET_ADVANNET_RESPONSE args:  [/inventory 200 tag.xml 200
       Time 0]
47 URL:> http://127.0.0.1:10000/response
48
49 ------------------------------------------------------------------

50
51 Execute SLEEP  args:  [1]
52
53 ------------------------------------------------------------------

54
55 Execute Get Transaction
56 URL:> https://192.168.1.100:8888/transaction?unmatched=true
57 Body: {
58   "Antenna": -1,
59   "EmpId": "987654321",
60   "Id": "123456789",
61   "Mode": "Sale",
62   "PosId": "12344478",
63   "Products": null,
64   "Unmatched": [
65     "30801191a50200602a24c27d"
```

```
  ]
}
Expected Response: {
  "Antenna": -1,
  "EmpId": "987654321",
  "Id": "123456789",
  "Mode": "Sale",
  "PosId": "12344478",
  "Products": null,
  "Unmatched": [
    "30801191a50200602a24c27d"
  ]
}
Response matches the expected response

------------------------------------------------------------------------


Execute AddProduct args:  [products.json]
Contents of file: {
    "Products": [
        "0188644885761"
    ]
}
URL:> https://192.168.1.100:8888/transaction/product

------------------------------------------------------------------------


Execute Get Transaction
URL:> https://192.168.1.100:8888/transaction?unmatched=true
Body: {
  "Antenna": -1,
  "EmpId": "987654321",
  "Id": "123456789",
  "Mode": "Sale",
  "PosId": "12344478",
  "Products": [
    {
      "Id": "0188644885761",
      "Status": "Singulated"
    }
  ]
}
Expected Response: {
  "Antenna": -1,
  "EmpId": "987654321",
  "Id": "123456789",
  "Mode": "Sale",
```

```
  "PosId": "12344478",
  "Products": [
    {
      "Id": "0188644885761",
      "Status": "Singulated"
    }
  ]
}
Response matches the expected response

-------------------------------------------------------------------


Execute SLEEP  args:  [1]

-------------------------------------------------------------------


Execute ExecuteTransaction
URL:> https://192.168.1.100:8888/transaction/execute

-------------------------------------------------------------------


Execute SLEEP  args:  [1]

-------------------------------------------------------------------


Execute DeleteTransaction
URL:> https://192.168.1.100:8888/transaction
Body: {
  "Antenna": -1,
  "EmpId": "987654321",
  "Id": "123456789",
  "Mode": "Sale",
  "PosId": "12344478",
  "Products": [
    {
      "Id": "0188644885761",
      "Status": "Processed"
    }
  ]
}
Expected Response: {
  "Antenna": -1,
  "EmpId": "987654321",
  "Id": "123456789",
  "Mode": "Sale",
```

```
  "PosId": "12344478",
  "Products": [
    {
      "Id": "0188644885761",
      "Status": "Processed"
    }
  ]
}
Response matches the expected response

------------------------------------------------------------------------


Execute STOP_POS args:  []
2024/09/02 18:17:56 Running command and waiting for it to finish
    ...
2024/09/02 18:17:56 Command finished with error: exit status 6

------------------------------------------------------------------------

2024/09/02 18:17:56 Test PASSED
```

Listing C.1: Logs for a Passed test using scriptRunner.

```
------------------------------------------------------------------------

Execute START_POS args:  []
Starting executables
2024/09/11 10:54:06 Running command and waiting for it to finish
    ...
2024/09/11 10:54:06 Command finished with error: exit status 6

------------------------------------------------------------------------


Execute SLEEP  args:  [10]

------------------------------------------------------------------------

Execute StartTransaction args:  [transaction.json]
Contents of file: {
    "Id": "123456789",
    "EmpId": "987654321",
    "PosId": "12344478",
    "Mode": "Sale"
}

```

```
20 ------------------------------------------------------------------

21
22 Execute Get Transaction
23 Body: {
24   "Antenna": -1,
25   "EmpId": "987654321",
26   "Id": "123456789",
27   "Mode": "Sale",
28   "PosId": "12344478",
29   "Products": null
30 }
31 Expected Response: {
32   "Antenna": -1,
33   "EmpId": "987654321",
34   "Id": "123456789",
35   "Mode": "Sale",
36   "PosId": "12344478",
37   "Products": null
38 }
39 Response matches the expected response
40
41 ------------------------------------------------------------------

42
43 Execute SET_ADVANNET_RESPONSE args:  [/inventory 200 tag.xml 200
       Time 0]
44 URL:> http://127.0.0.1:10000/response
45
46 ------------------------------------------------------------------

47
48 Execute SLEEP  args:  [1]
49
50 ------------------------------------------------------------------

51
52 Execute Get Transaction
53 URL:> https://192.168.1.100:8888/transaction?unmatched=true
54 Body: {
55   "Antenna": -1,
56   "EmpId": "987654321",
57   "Id": "123456789",
58   "Mode": "Sale",
59   "PosId": "12344478",
60   "Products": null,
61   "Unmatched": [
62     "30801191a50200602a24c27d"
63   ]
```

```
}
Expected Response: {
  "Antenna": -1,
  "EmpId": "987654321",
  "Id": "123456789",
  "Mode": "Sale",
  "PosId": "12344478",
  "Products": null,
  "Unmatched": [
    "30801191a50200602a24c27d"
  ]
}
Response matches the expected response

------------------------------------------------------------------------


Execute AddProduct args:  [products.json]
Contents of file: {
    "Products": [
        "0188644885761"
    ]
}
URL:> https://192.168.1.100:8888/transaction/product

------------------------------------------------------------------------


Execute Get Transaction
URL:> https://192.168.1.100:8888/transaction?unmatched=true

Body: {
  "Antenna": -1,
  "EmpId": "987654321",
  "Id": "123456789",
  "Mode": "Sale",
  "PosId": "12344478",
  "Products": null,
  "Unmatched": [
    "30801191a50200602a24c27d"
  ]
}
Expected Response: {
  "Antenna": -1,
  "EmpId": "987654321",
  "Id": "123456789",
  "Mode": "Sale",
  "PosId": "12344478",
  "Products": [
```

```
111     {
112       "Id": "0188644885761",
113       "Status": "Singulated"
114     }
115   ]
116 }
117
118 Response doesn't match expected response
119 exit status 3
120
121 Test FAILED:  Response of "GET_TRANSACTION" command
122 doesn't match expected response
123 ------------------------------------------------------------------------
```

Listing C.2: Logs for a Failed test using scriptRunner.