POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Politecnico di Torino

Master's Degree Thesis

AI FIRMWARE FOR REMOTE WIRELESS PLANT HEALTH MONITORING

Supervisors

Prof. Danilo DEMARCHI

Prof. Umberto GARLANDO

Ph.D. Federico CUM

Candidate

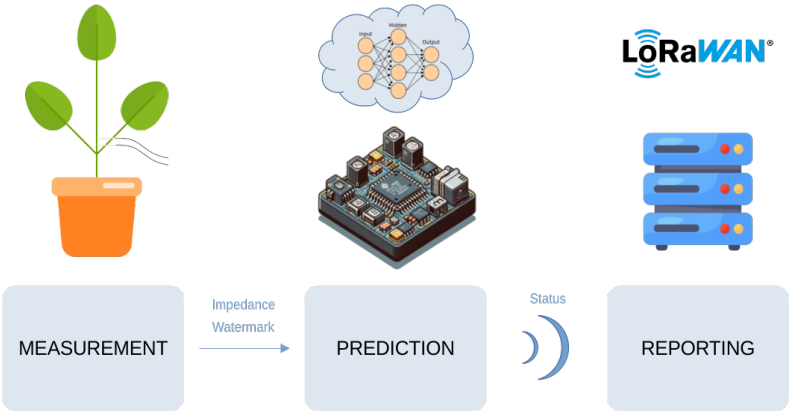
Andrea REPETTO

July 2024

Summary

Plant health monitoring is critical for ensuring agricultural productivity and sustainability. Traditional methods of monitoring plant health are often labor-intensive, time-consuming, and subject to human error. With the increasing global demand for food and the challenges posed by climate change, there is a pressing need for more efficient, accurate, and scalable solutions. Neural networks, a subset of artificial intelligence, have emerged as a powerful tool to address complex problems in various domains, including agriculture.

This master's thesis investigates the creation and deployment of firmware designed to monitor plant health via impedance analysis, utilizing neural networks. The importance of monitoring plant health in fields such as agriculture, horticulture, and environmental science highlights the need for effective and reliable methodologies. Conventional methods frequently lack accuracy and real-time monitoring capabilities, often relying solely on environmental condition predictions. This research aims to use intrinsic characteristics to develop firmware capable of accurately assessing the health status of plants.



The research starts by reviewing neural network architectures to find suitable designs. Firmware was developed using an STM32 Nucleo devkit with a sensor shield for data measurement, conducted by doctoral students from the eLiONS group. The project includes remote control features via the low-power, long-range LoRaWAN protocol. Results were evaluated using The Things Network, a common LoRaWAN stack.

In summary, the integration of neural networks into plant health monitoring systems represents a significant advance in agricultural technology. This thesis aims to contribute to the body of knowledge in this field by providing a comprehensive study of neural network firmware, its design, implementation, and practical applications in monitoring plant health.

Acknowledgements

"To Ennio, my dear friend who left us during the Covid pandemic."

I would like to express my deepest gratitude to my supervisors, Professor Garlando and Dr. Cum, for their invaluable guidance, support, and encouragement throughout the course of this research. Their expertise and insights were instrumental in shaping this thesis, and their continuous mentorship provided me with the knowledge and motivation to overcome the challenges encountered during this project.

A heartfelt thank you goes to my family for their unwavering support and love. Their belief in my abilities and their constant encouragement have been the foundation upon which I have built my academic journey. I am especially grateful to my parents for their sacrifices and for always being there to support me in every possible way.

Lastly, I would like to extend my sincerest thanks to my girlfriend, Sabina. Her patience, understanding, and unwavering support have been a constant source of strength and motivation for me. Her encouragement and love have been indispensable in helping me stay focused and determined throughout the highs and lows of this journey.

To everyone who has supported me along the way, thank you. This thesis would not have been possible without your help and encouragement.

Table of Contents

List of Tables

List of Figures

Acronyms

CLI
Command line interface

AI
Artificial Intelligence

FUOTA
Firmware update over-the-air

FW
Firmware

HW
Hardware

IoT
Internet of Things

I2C
Inter-Integrated Circuit

LoRa
Long Range protocol - physical layer

LoRaWAN
Long Range Wide Area Network - networking layers

LUT
Look-up table

MCU
Micro controller unit

ML
Machine learning

NN
Neural network

SDI-12
Serial Digital Interface at 1200 baud

SDK
Software development kit

STM
ST Microelectronics

UART
Universal Asynchronous Receiver-Transmitter

TTN
The Things Network - a LoRa cloud platform

ToA
Time over Air

WIP
Work in Progress

Chapter 1

Introduction

Plant health is critical to agricultural productivity and global food security. As the world's population continues to grow, the need for efficient and sustainable farming practices becomes more pressing. Healthy plants are essential not only for food production but also for maintaining ecological balance and biodiversity. Performing effective monitoring of plant health can help prevent crop losses, reduce the need for chemical interventions, and ensure optimal growth conditions, contributing to a sustainable future [1].

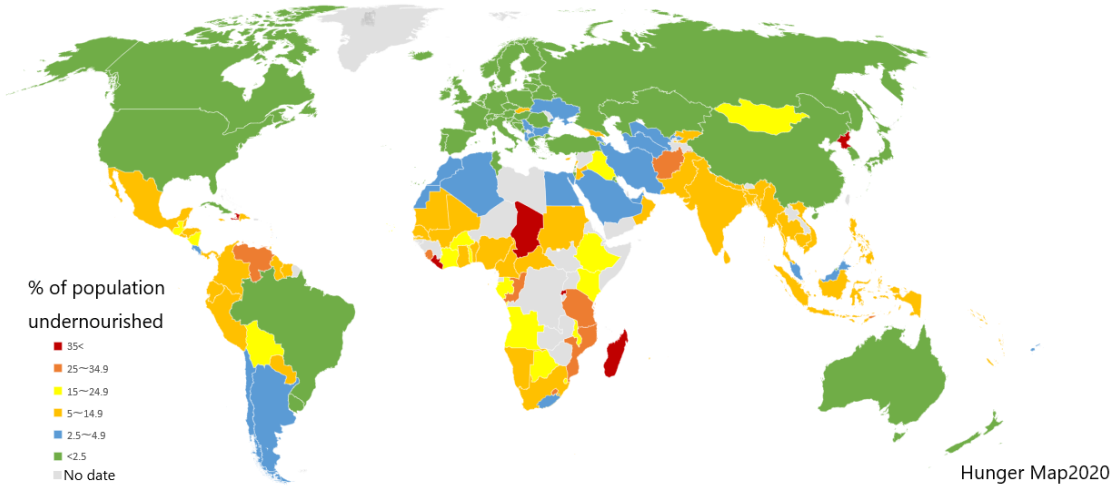


Figure 1.1: Global food security challenges highlighted by the FAO.

The Food and Agriculture Organization (FAO) of the United Nations highlights that the world is facing an unprecedented crisis in food security. Several

factors contribute to this crisis, including climate change, soil degradation, water scarcity, and the increasing prevalence of pests and diseases. Climate change, in particular, has altered weather patterns, leading to extreme events such as droughts and floods that directly impact crop yields. Soil degradation, resulting from unsustainable farming practices, reduces the land's fertility, further exacerbating food production challenges. In addition, the overuse of chemical fertilizers and pesticides has led to environmental pollution and a decrease in biodiversity [1].

Effective plant health monitoring is essential to address these challenges. Early detection of plant stress, diseases, and nutrient deficiencies can enable timely interventions, reducing the risk of significant crop loss. Advanced monitoring techniques can also provide farmers with precise information about the health of their crops, allowing for more targeted and efficient use of resources such as water and fertilizers. This not only helps in improving crop yields but also contributes to the sustainability of agricultural practices [1].

1.1 Limitations of Existing Commercial Solutions

In recent years, various commercial solutions have emerged that utilize different types of sensors and communication protocols to monitor plant health. These solutions range from simple moisture sensors to sophisticated devices that measure a variety of environmental and physiological parameters. Despite their advancements, many existing commercial nodes have limitations in terms of accuracy, range, and power consumption [2].

Figure 1.2: Example of a commercial plant health monitoring device.

For instance, some commercial devices rely heavily on environmental condition predictions and do not account for the intrinsic characteristics of plants that may indicate health issues. Others may offer high accuracy, but are limited by short-range communication protocols or high power consumption, making them less suitable for large-scale or remote agricultural operations [3]. These limitations highlight the need for a more robust and efficient approach to plant health monitoring.

1.2 Innovative Approach: Neural Network Firmware

This research aims to develop neural network firmware for monitoring plant health, addressing the shortcomings of current methods. Using impedance analysis, the firmware seeks to provide accurate assessments of plant health based on intrinsic characteristics. Furthermore, the integration of the LoRaWAN protocol (long-range wide area network) ensures low power consumption and extensive range, making it ideal for large-scale agricultural applications [4].

The development process involves a thorough examination of existing neural network architectures to identify the most suitable design for this application. The firmware is developed using a devkit from the STM32 Nucleo series, equipped with a sensor shield containing all necessary devices for data measurement. Measurement tasks are conducted by doctoral students in the eLiONS group, and the results are monitored through The Things Network, a commonly used LoRaWAN stack [5].

This thesis aims to advance plant health monitoring technologies by presenting a detailed study of neural network firmware, its design, implementation, and practical applications. The research underscores the importance of leveraging advanced technologies to ensure sustainable agricultural practices and secure the future of global food production.

Chapter 2

Neural Network Description

2.1 Machine Learning

Machine Learning (ML) is a subset of artificial intelligence that focuses on the development of algorithms and statistical models that enable computers to perform specific tasks without explicit instructions. ML algorithms build models based on sample data, known as training data, to make predictions or decisions without being programmed to perform the task [6].

The general process of machine learning involves the following steps:

- **Data Collection:** Gathering relevant data for the task.
- **Data Preprocessing:** Cleaning and organizing the data.
- **Feature Engineering:** Selecting and transforming variables (features) to improve model performance.
- **Model Training:** Using algorithms to train a model on the data.
- **Model Evaluation:** Assessing the model's performance on unseen data.
- **Model Deployment:** Implementing the model in a real-world scenario.

The goal of machine learning is to minimize a cost function, often called a loss function, which measures the error between the predicted and actual outputs. For a dataset with n samples, the Mean Squared Error (MSE) is a commonly

used loss function, defined as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where y_i is the actual value and $\hat{y}_i$ is the predicted value [7].

2.2 Neural Networks

Neural Networks are a type of machine learning model inspired by the structure and function of the human brain. They consist of layers of interconnected nodes, called neurons, that process input data to generate an output. Each connection between neurons has an associated weight, which is adjusted during training to minimize the error in predictions [8].

The output of a neuron is calculated as a weighted sum of its inputs, followed by the application of an activation function:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

$$a = \sigma(z)$$

where w_i are the weights, x_i are the inputs, b is the bias term and σ is the activation function [9].

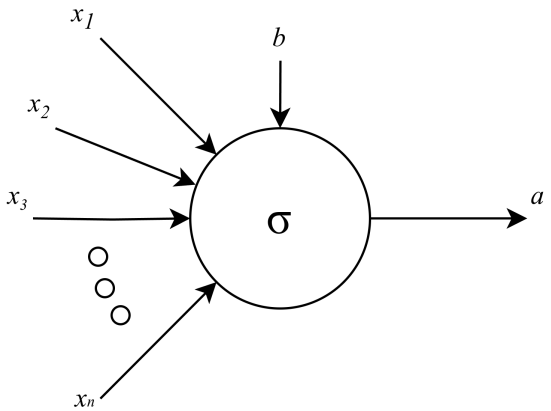


Figure 2.1: Example of a neuron

2.2.1 Activation Functions: ReLU

An activation function determines whether or not a neuron should be activated, based on the input it receives. One of the most popular activation functions is the Rectified Linear Unit (ReLU). ReLU is defined as:

$$\text{ReLU}(x) = \max(0, x)$$

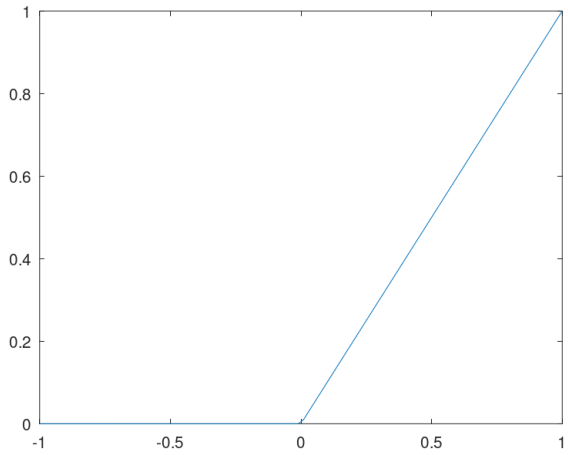


Figure 2.2: Plotting of ReLU function

ReLU introduces non-linearity into the model, enabling it to learn complex patterns. It is computationally efficient and helps mitigate the problem of vanishing gradients, making it suitable for deep neural networks [10].

2.3 Neural Network Model for Plant Health Monitoring

In this project, we develop a neural network model designed to monitor plant health by analyzing two input features: Soil Water Potential (SWP) and Impedance. The model outputs a status indicator that signifies the health condition of the plant.

An example of the NN used is provided in Figure 2.3.

2.3.1 Network Inputs

For plant health monitoring, the approach of the project tries to estimate the status from the features provided. The latter were measured with a custom

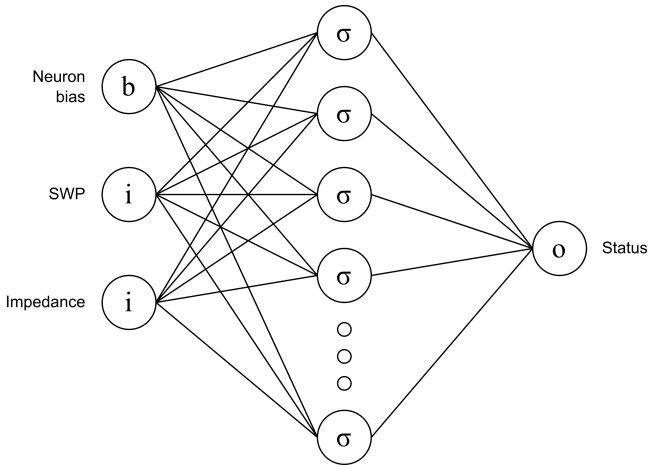


Figure 2.3: Drawing of project NN structure

shield that has been developed by doctoral Calvo [11].

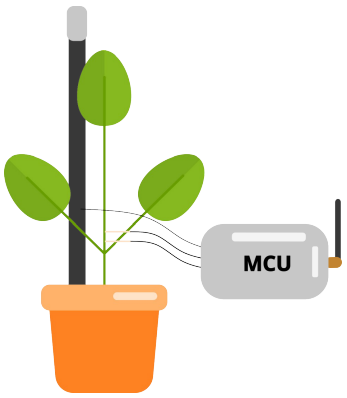


Figure 2.4: Sensors setup on plant

Soil Water Potential

The SWP is an important indicator of plant health, that measures the energy of migrating fluid, in this case water, and it is characterized by Clapeyron equations. Its main composition is the sum of osmotic and soil matric potential, but in some cases also soil gravitational and pressure matric [12]. The latter can be used for the study of flux in frozen liquid and for potential in low salinity, but for the case of study only the osmosis component is important as an indicator of the availability of energy for the plant to absorb unfrozen water from the environment. This is due to daily transpiration which make necessary to reintegrate liquids loss [13].

SWP is measured in kPa, but the sensor used in this project outputs raw data, which is called *Watermark* in this project. This implies that the value needs a conversion, which is achieved by using two LUTs: respectively converting $Watermark \rightarrow Resistance$ and $Resistance \rightarrow SWP$. These tables were provided by doctoral Calvo which used manufacturer datasheet to produce an accurate characteristic function.

Impedance Modulus

As highlighted by eLiONS team, impedance has been proved to rise under plant stem drying condition. This appears to be a direct indicator of plant health, but unfortunately measuring phase in an optical with low power consumption is difficult, especially with the tools currently available on the market [14]. The current sensor board is capable of measuring impedance using frequency variation inducted in an integrated circuit oscillator.

Chapter 3

Firmware Implementation

3.1 Introduction

In this chapter, the firmware developed by the candidate is accurately descripted for every part. The project started from the standard LoRaWAN_End_Node example provided by eLiONS doctorals, who has modified the one shipped by STM on SDK repository `STM32Cube_FW_WL_V1.3.0`. The goal of the project was to implement an application which starts with LoRa transmission ruotine and then uses the measured values to make a prediction on the health status of the plant.

3.2 Board Description

It is essential for a developer, but it is also useful for a better understanding of the thread, to know its target device. In this section, there is a detailed description of the board which has been used: STM32 Nucleo WL55JC1.

3.2.1 Block Scheme of the Architecture

The STM32 Nucleo WL55JC1 is based on the STM32WL55JC microcontroller, which integrates a dual-core architecture combining an Arm Cortex-M4 core and an Arm Cortex-M0+ core. This configuration provides flexibility and

efficiency in handling various tasks, such as communication protocols and sensor data processing, simultaneously.

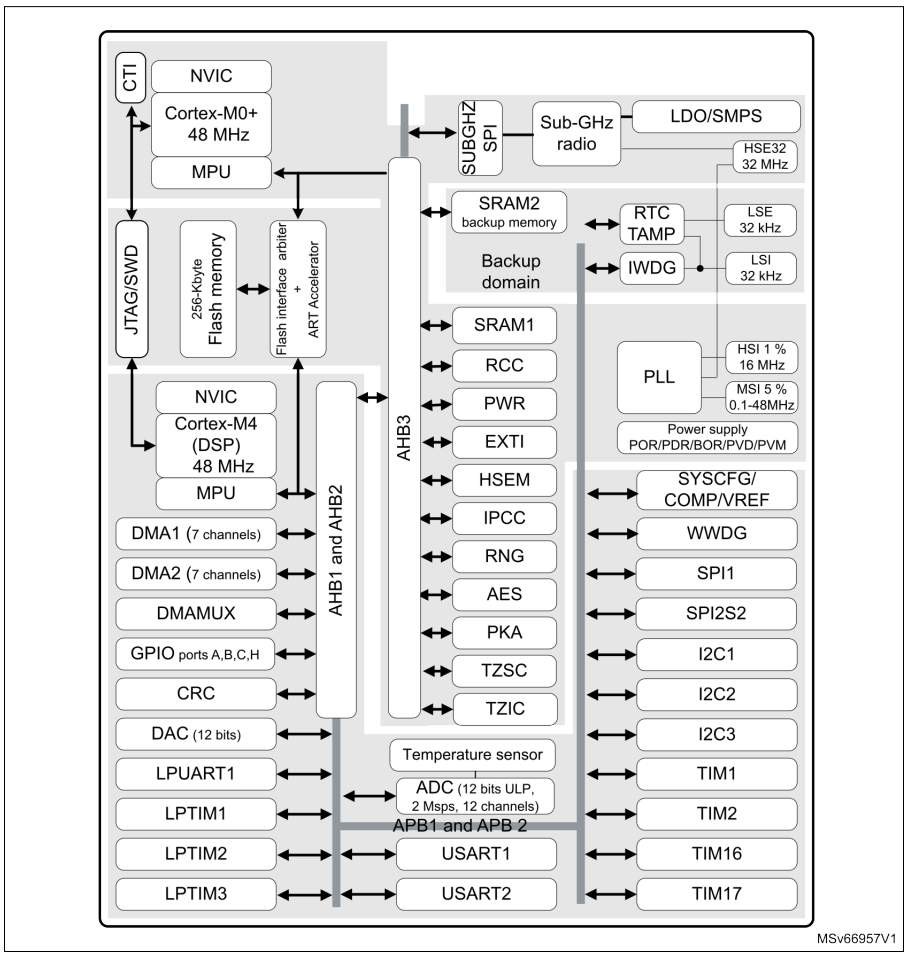


Figure 3.1: Block diagram of the STM32WL55JC architecture [15]

The block diagram (Figure 3.1) illustrates the major components and their interconnections within the STM32WL55JC microcontroller. Key components include:

- **Cortex-M4 Core:** The primary core responsible for executing the main application code, supporting DSP instructions and floating-point operations.
- **Cortex-M0+ Core:** The secondary core used for handling low-power tasks and peripheral control, optimizing energy efficiency.
- **Embedded Memories:** Including Flash memory, SRAM1, SRAM2, and system memory, providing storage for code, data, and boot operations.

- **Peripherals:** A wide range of peripherals such as GPIOs, ADCs, DACs, USARTs, I2C, SPI, and more, enabling versatile interfacing with external devices.
- **Radio Subsystem:** An integrated sub-GHz radio transceiver supporting LoRa, (G)FSK, (G)MSK, and BPSK modulations, essential for IoT applications.

3.2.2 Memory Address Designation

The STM32WL55JC microcontroller features a comprehensive memory map that designates specific address ranges for different memory types and peripherals. Understanding this memory map is crucial for efficient firmware development and debugging.

Memory Type	**Address Range**	
	Start	End
CPUx Flash	0x0000 0000	0x0003 FFFF
Flash Memory	0x0800 0000	0x0803 FFFF
SRAM2 (CPU1 only)	0x1000 0000	0x1000 7FFF
System Memory	0x1FFF 7000	0x1FFF 73FF
OTP Area	0x1FFF 7400	0x1FFF 77FF
Option Bytes	0x1FFF 7800	0x1FFF 7FFF
SRAM1	0x2000 0000	0x2000 7FFF
SRAM2	0x2000 8000	0x2000 FFFF
Peripheral Registers	0x4000 0000	0x5801 FFFF
CPUx Internal Peripherals	0xE000 0000	0xFFFF FFFF

Table 3.1: Memory address designation of STM32WL55JC [16]

The memory types and their corresponding address ranges are detailed in Table 3.1. This includes:

- **CPUx Flash:** Memory area used for executing code, mapped to the beginning of the address space.
- **Flash Memory:** Non-volatile memory used for storing the main firmware code.
- **SRAM2 (CPU1 only):** Dedicated SRAM for CPU1, providing fast access to frequently used data.
- **System Memory:** Contains the bootloader, enabling firmware updates and recovery.

- **OTP Area:** One-time programmable memory for storing critical data such as calibration values and keys.
- **Option Bytes:** Configuration memory for setting device-specific options such as read-out protection and watchdog timers.
- **SRAM1:** Primary volatile memory for data storage during operation.
- **SRAM2:** Additional volatile memory providing more space for data storage.
- **Peripheral Registers:** Address ranges dedicated to control and status registers for on-chip peripherals.
- **CPUx Internal Peripherals:** Reserved for core functionalities such as interrupt control, debug, and system configuration.

3.2.3 Peripheral Overview

The STM32 Nucleo WL55JC1 provides a rich set of peripherals that facilitate diverse applications, particularly in IoT and smart agriculture. Key peripherals include:

- **General-Purpose I/O (GPIO):** Configurable pins for digital input/output operations, essential for interfacing with sensors and actuators.
- **Analog-to-Digital Converter (ADC):** Converts analog signals to digital values, crucial for processing sensor data.
- **Digital-to-Analog Converter (DAC):** Generates analog output from digital values, useful in control applications.
- **Universal Synchronous/Asynchronous Receiver Transmitter (USART):** Facilitates serial communication, commonly used for debugging and data transmission.
- **Inter-Integrated Circuit (I2C):** Enables communication with various peripherals such as sensors and EEPROMs using a simple two-wire interface.
- **Serial Peripheral Interface (SPI):** Supports high-speed communication with external devices such as flash memory and display modules.
- **Timers and PWM:** Provides timing control and pulse-width modulation, essential for tasks like motor control and signal generation.
- **Low-Power Modes:** Multiple low-power modes to reduce energy consumption, critical for battery-operated applications.

These peripherals, combined with the dual core architecture and integrated radio subsystem, make the STM32 Nucleo WL55JC1 an ideal platform to develop sophisticated IoT applications that require reliable performance, efficient power management, and robust communication capabilities.

3.2.4 Development Environment

The development environment for STM32 Nucleo WL55JC1 involves the use of STM32CubeIDE, a comprehensive software development suite that integrates various tools for coding, debugging and programming STM32 microcontrollers.

- **STM32CubeMX Integration:** Facilitates peripheral configuration and code generation through a graphical interface.
- **Debugger:** Offers advanced debugging capabilities, including breakpoints, watchpoints, and real-time variable monitoring.
- **Code Management:** Supports version control systems such as Git, aiding in collaborative development.
- **Project Templates:** Provides ready-to-use project templates for quick development setup.
- **Makefile Generation:** Permit an automated way to generate makefiles, an easy way to quickly run a build configuration.

3.3 X-CUBE-AI

STM Nucleo boards do not support the ONNX neural network directly, as specified in Section 4.2.1, then a utility is provided by manufacturer. The following is a comprehensive description of the suite.

The X-CUBE-AI expansion package is an advanced tool and library suite developed by STMicroelectronics to facilitate the integration of artificial intelligence (AI) algorithms into STM32 microcontrollers. This comprehensive package enables developers to deploy pre-trained neural network models on STM32 devices, providing an efficient pathway from AI model development to deployment on embedded systems. The following sections provide a detailed overview of the X-CUBE-AI tool and its associated libraries [17].

3.3.1 Overview of X-CUBE-AI

X-CUBE-AI is designed to bridge the gap between AI model development environments and STM32 microcontroller platforms. It supports the conversion

of neural network models trained in popular machine learning frameworks such as TensorFlow, Keras, and ONNX, into optimized C code that can run on STM32 microcontrollers. This conversion process includes several optimization techniques to ensure that the neural network operates efficiently within the resource constraints of an embedded environment [18].

Key Features

The X-CUBE-AI package offers a range of features that facilitate the deployment of AI models on STM32 microcontrollers:

- **Model Conversion:** Converts neural network models from high-level frameworks (e.g., TensorFlow, Keras, ONNX) into optimized C code for STM32 devices [19].
- **Optimization:** Includes techniques such as quantization, pruning, and layer fusion to reduce the memory footprint and computational requirements of the neural network [20].
- **Performance Monitoring:** Provides tools for measuring the runtime performance of the neural network on the target microcontroller, helping developers identify and address performance bottlenecks [17].
- **Integration with STM32CubeMX:** Seamlessly integrates with STM32CubeMX, a graphical tool for configuring STM32 microcontrollers, simplifying the process of integrating AI models into embedded applications [21].
- **Support for Multiple Neural Network Layers:** Supports a wide range of neural network layers, including dense (fully connected), convolutional, pooling, activation, and recurrent layers [19].
- **Memory Management:** Efficiently manages memory allocation for neural network weights, biases, and intermediate activations, ensuring optimal use of the microcontroller's resources [19].

Model Conversion and Optimization

The X-CUBE-AI toolchain involves several steps to convert and optimize AI models for embedded deployment:

- **Model Import:** Neural network models are imported from supported frameworks using the X-CUBE-AI GUI or command-line interface [17].
- **Layer Mapping:** The tool maps high-level neural network layers to corresponding optimized implementations available in the X-CUBE-AI library [19].

- **Quantization:** The tool performs quantization, reducing the precision of weights and activations from floating-point to fixed-point representations, significantly decreasing the model size and computational requirements [22].
- **Pruning:** Unnecessary neurons and connections are pruned from the network, further reducing the model complexity and enhancing inference speed [23].
- **Code Generation:** The optimized model is converted into C code, which includes the necessary functions to run the neural network on the STM32 microcontroller [17].

Integration with STM32CubeMX

The integration with STM32CubeMX simplifies the configuration and deployment of neural networks on STM32 microcontrollers:

- **Graphical Configuration:** Developers can use STM32CubeMX's graphical interface to configure microcontroller peripherals and parameters required for AI applications [21].
- **Code Generation:** STM32CubeMX generates initialization code for peripherals and middleware, including the X-CUBE-AI library, ensuring seamless integration with the application code [21].
- **Project Management:** The tool provides project management capabilities, allowing developers to manage files, dependencies, and build settings within a single integrated environment [21].

Performance Monitoring and Debugging

X-CUBE-AI includes tools for monitoring the performance of neural networks on STM32 microcontrollers:

- **Runtime Analysis:** Developers can measure the execution time of each layer in the neural network, identifying performance-critical sections [19].
- **Memory Usage:** The tool provides insights into memory allocation and usage, helping developers optimize the memory footprint of their AI applications [19].
- **Debugging Support:** Integrated debugging features allow developers to step through neural network execution, inspect intermediate results, and diagnose issues [19].

3.3.2 Library Files Description

In Figure 3.2 there are some files highlighted in yellow. These are the library files which come from X-CUBE-AI, except for `app_x-cube-ai.c` that is intended to be modified by user. In Section 3.5 the user-defined methods are exhaustively detailed. There are also some files in green: they came from X-CUBE-AI C model conversion (Section 4.2.1). They need to be updated to change the neurons in the network. The folder *Middlewares* containes all headers and static libraries provided by the STM package.

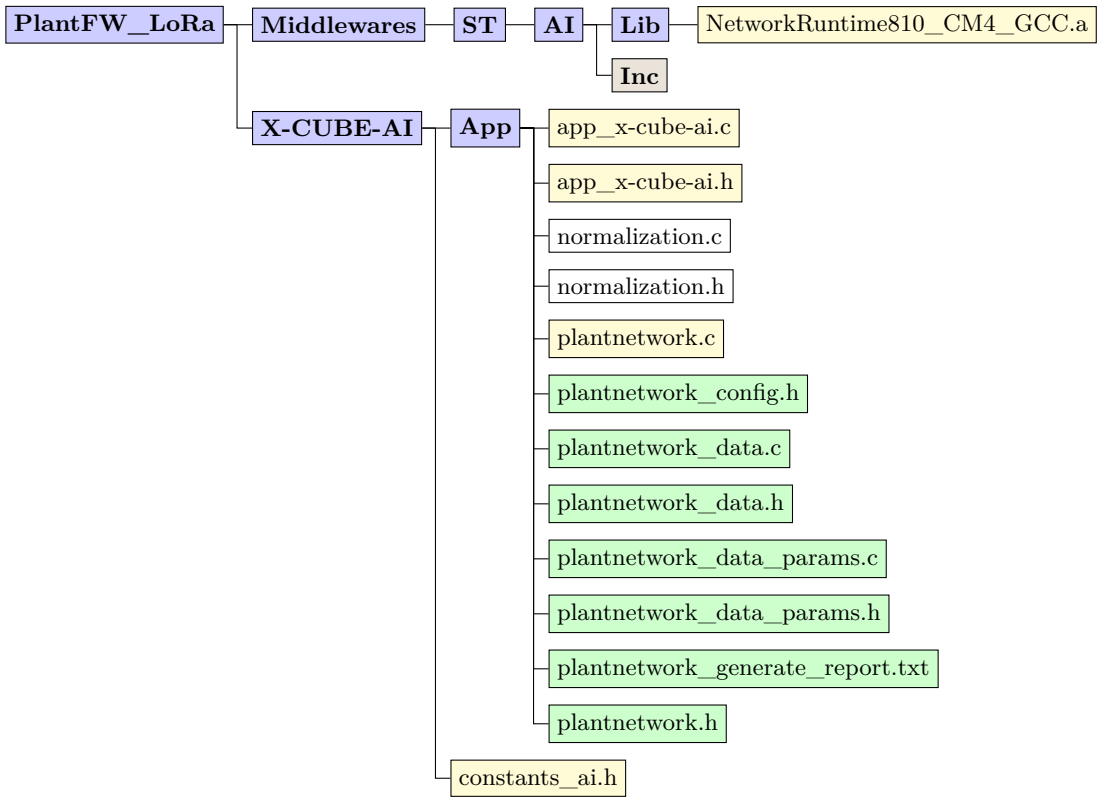


Figure 3.2: X-CUBE-AI library files

3.4 LoRaWAN: Long Range Wide Area Network

3.4.1 Introduction to LoRa

LoRa (Long Range) is a modulation technique derived from chirp spread spectrum (CSS) technology. It is designed for long-range communication with low

power consumption, making it ideal for Internet of Things (IoT) applications where devices need to communicate over long distances without consuming significant amounts of power [24]. LoRa operates in unlicensed ISM (Industrial, Scientific and Medical) radio bands, such as 868 MHz in Europe and 915 MHz in North America, providing flexibility for global deployment.

LoRa modulation spreads the signal across a wide frequency spectrum, which helps in resisting interference and allows for successful data transmission over long distances, even in challenging environments. The key advantages of LoRa include its long communication range (up to 15 km in rural areas), low power requirements, and robustness against interference, which are crucial for battery-operated devices in remote locations [4].

3.4.2 LoRaWAN Protocol

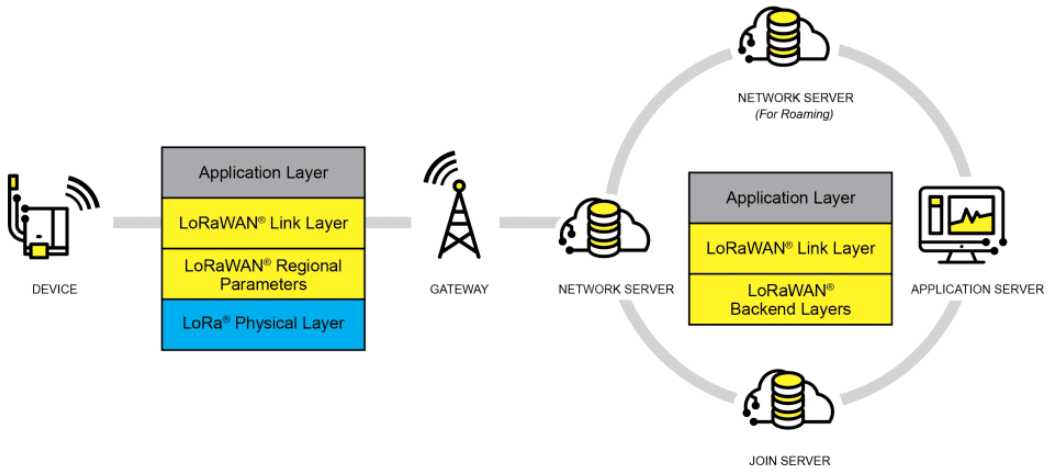


Figure 3.3: LoRaWAN Network Architecture

LoRaWAN (Long Range Wide Area Network) is a communication protocol and system architecture designed to manage LoRa wireless communication networks. It is maintained by the LoRa Alliance, a non-profit association that drives the standardization and global harmonization of the LoRaWAN protocol [24].

The LoRaWAN protocol is designed to provide low-power, long-range communication capabilities to IoT devices. It operates in a star-of-stars topology, where gateways relay messages between end devices and a central network server. Gateways are connected to the network server via standard IP connections, while end devices use single-hop wireless communication to one or multiple

gateways. The communication between the end devices and gateways is bi-directional, and the protocol supports multicast operation, which is useful for firmware updates over the air (FUOTA) [25].

3.4.3 LoRaWAN Architecture

The LoRaWAN network architecture comprises several key components:

- **End Devices:** These are the sensors or actuators that collect data or perform actions based on received commands. They are typically battery-powered and designed for low power consumption.
- **Gateways:** These devices act as bridges, receiving data from end devices and forwarding it to the network server. Gateways can cover large areas and support thousands of end devices.
- **Network Server:** The central component that manages the network. It handles data routing, network management, and security. The network server processes the data received from gateways and forwards it to the appropriate application servers.
- **Application Servers:** These servers host the end-user applications that process and use the data collected by end devices. They provide interfaces for data analysis, visualization, and control.

Figure 3.3 illustrates the LoRaWAN network architecture, highlighting the interactions between the end devices, the gateways, the network server and the application servers.

3.4.4 LoRaWAN Communication Classes

LoRaWAN defines three different device classes to address various application needs in terms of latency and energy consumption:

- **Class A (All):** This class offers the lowest power operation, suitable for battery-powered sensors. Each end device's uplink transmission is followed by two short downlink receive windows. Communication from the server to the end device can only occur immediately after the end device has sent an uplink transmission. This class is ideal for applications where downlink communication is infrequent and can wait until the next uplink.
- **Class B (Beacon):** In addition to the Class A functionalities, Class B devices open extra receive windows at scheduled times. The network server uses time-synchronized beacons to ensure that the end device receives

downlink messages at specific times. This class is suitable for applications requiring more frequent downlink communication.

- **Class C (Continuous):** This class offers the lowest latency for downlink communication, suitable for applications requiring frequent and low-latency downlink messages. Class C devices have almost continuously open receive windows, except for the short time needed to transmit uplink messages. This results in higher power consumption compared to Class A and Class B devices.

3.4.5 Security in LoRaWAN

Security is a critical aspect of LoRaWAN, ensuring data integrity, confidentiality, and authenticity. LoRaWAN employs a robust security framework that includes the following features [26]:

- **End-to-End Encryption:** Data payloads are encrypted between the end device and the application server using AES-128 encryption.
- **Network Security:** Ensures the authenticity of the device in the network using unique network session keys.
- **Application Security:** Protects application data using unique application session keys.

These security measures prevent unauthorized access and ensure that the data transmitted over the network remains secure and tamper-proof.

3.4.6 LoRaWAN Application Files

The files involved in the projects are stored in `LoRaWAN` folder. In particular, `se-identity` stores LoRaWAN credentials, `app_lorawan.c` defines the main processes (init and update) and in `lora_app` every LoRa event ruotine and application is defined. This includes also X-CUBE-AI processes because they are scheduled by LoRaWAN transmission.

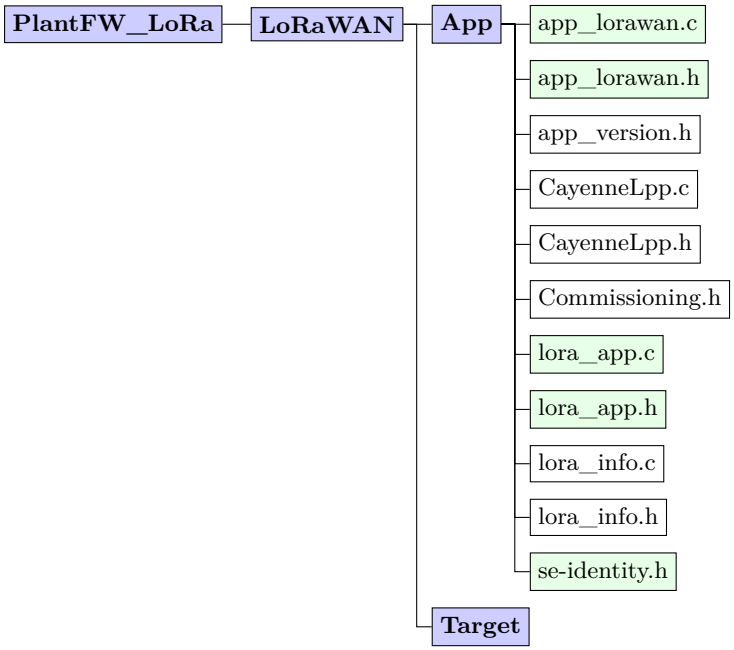


Figure 3.4: LoRaWAN application file tree

3.5 Execution Flow

In this section there is an exhaustive description of the main routine of the FW. All parts that are not described have been left unmodified by the candidate.

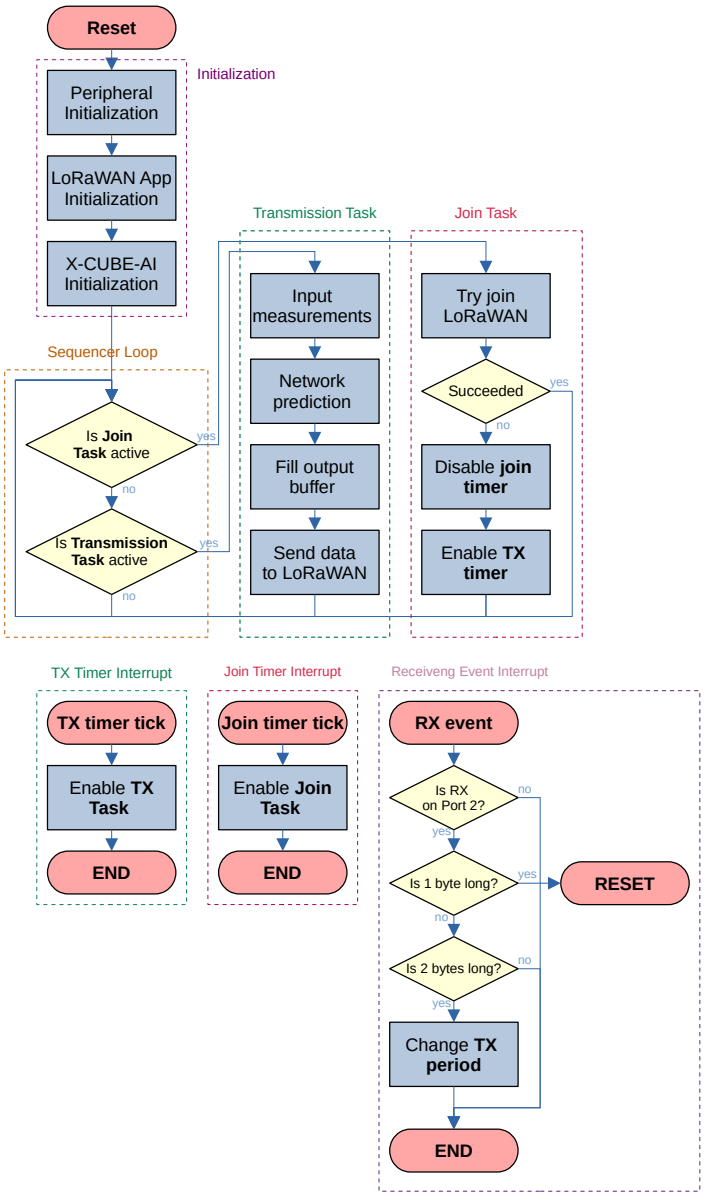


Figure 3.5: ASM chart of FW

3.5.1 Peripheral Initialization

The `main()` function in the firmware is responsible for the initial setup and configuration of the microcontroller's peripherals before entering the main program loop. This section provides a detailed explanation of the peripheral initialization process that is typically found in the `main.c` file.

System Initialization The peripheral initialization process begins with the system initialization, which includes configuring the system clock and initializing hardware abstraction layers.

```
int main(void)
{
    /* Reset of all peripherals, Initializes the Flash
   interface and the Systick. */
    HAL_Init();

    /* Init STM genuine check (CRC) */
    __HAL_RCC_CRC_CLK_ENABLE();

    /* Configure the system clock */
    SystemClock_Config();
```

- **HAL_Init():** This function initializes the Hardware Abstraction Layer (HAL) and configures the Flash interface and the Systick timer. It ensures that the microcontroller is in a known state before peripheral initialization begins.
- **SystemClock_Config():** This function configures the system clock, setting the clock source, prescalers, and other parameters to ensure the microcontroller operates at the desired frequency.

Peripheral Initialization

Following system initialization, the `main()` function proceeds to initialize various peripherals. Each peripheral typically has its own initialization function, which sets up the peripheral's configuration registers and prepares it for use.

```
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_LoRaWAN_Init();
```

```
4   MX_I2C2_Init();
5   MX_TIM1_Init();
6   MX_TIM2_Init();
```

- **MX_GPIO_Init():** This function initializes the General-Purpose Input/Output (GPIO) pins. It configures the pin modes (input, output, alternate function), pull-up/pull-down resistors, and initial output levels.
- **MX_LoRaWAN_Init():** This function initializes the LoRaWAN communication protocol, enabling the microcontroller to communicate over long-range wireless networks.
- **MX_I2C2_Init():** This function initializes the Inter-Integrated Circuit (I2C) interface, specifically the I2C2 peripheral, used for communication with other I2C-compatible devices such as sensors and memory modules.
- **MX_TIM1_Init() and MX_TIM2_Init():** These functions initialize the Timer peripherals TIM1 and TIM2, which are used for generating precise time delays, PWM signals, and event counting.

GPIO Pin Configuration After initializing the peripherals, specific GPIO pins are configured to ensure that connected sensors and modules are properly controlled. The following code sets various pins to a low state (reset):

```
1  /* USER CODE BEGIN 2 */
2  HAL_GPIO_WritePin(GPIOB, EN_SEN_3_Pin, GPIO_PIN_RESET);
3  HAL_GPIO_WritePin(GPIOC, EN_SEN_1_Pin, GPIO_PIN_RESET);
4  HAL_GPIO_WritePin(GPIOA, EN_SEN_4_Pin, GPIO_PIN_RESET);
5  HAL_GPIO_WritePin(GPIOB, EN_SEN_2_Pin, GPIO_PIN_RESET);
6  HAL_GPIO_WritePin(GPIOA, IMP_EN_Pin, GPIO_PIN_RESET);
7  HAL_GPIO_WritePin(GPIOC, EN_WT_Pin, GPIO_PIN_RESET);
8  CLEAR_BIT(DBGMCU->CR, DBGMCU_CR_DBG_STOP);
9  CLEAR_BIT(DBGMCU->CR, DBGMCU_CR_DBG_SLEEP);
10 CLEAR_BIT(DBGMCU->CR, DBGMCU_CR_DBG_STANDBY);
```

- **HAL_GPIO_WritePin():** This function sets the specified GPIO pin to the given state (reset in this case). It is used to control the enable pins for various sensors and modules.
- **CLEAR_BIT(DBGMCU->CR, DBGMCU_CR_DBG_STOP):** This macro clears the specified bits in the Debug MCU Configuration Register, ensuring that debugging features are disabled during low-power modes like stop, sleep, and standby.

LoRaWAN Initialization

As seen in previous section, LoRaWAN initialization is a process named as `MX_LoRaWAN_Init()`, its definition is in `app_lorawan.c`.

```
void MX_LoRaWAN_Init(void)
{
  /* USER CODE BEGIN MX_LoRaWAN_Init_1 */

  /* USER CODE END MX_LoRaWAN_Init_1 */
  SystemApp_Init();
  /* USER CODE BEGIN MX_LoRaWAN_Init_2 */

  /* USER CODE END MX_LoRaWAN_Init_2 */
  LoRaWAN_Init();
  /* USER CODE BEGIN MX_LoRaWAN_Init_3 */

  /* USER CODE END MX_LoRaWAN_Init_3 */
}
```

The functions in the process are two: `SystemApp_Init()` configures the system clock, initializes timers, RTC, debug and trace utilities, environmental sensors, and low power manager, ensuring the system is ready for operations while `LoRaWAN_Init()` is the one that ensures the initialization of the application. The latter is defined in `lora_app.c`. Following, there is a condensed explanation of the main processes.

```
void LoRaWAN_Init(void)
{
  /* USER CODE BEGIN LoRaWAN_Init_LV */
  uint32_t feature_version = 0UL;
  /* USER CODE END LoRaWAN_Init_LV */

  /* USER CODE BEGIN LoRaWAN_Init_1 */

  /* Get LoRaWAN APP version*/
  APP_LOG(TS_OFF, VLEVEL_M, "APPLICATION_VERSION: V%X.%X.%X\r\n
    ",
          (uint8_t)(APP_VERSION_MAIN),
          (uint8_t)(APP_VERSION_SUB1),
          (uint8_t)(APP_VERSION_SUB2));

```

```
15  /* Get MW LoRaWAN info */
16  APP_LOG(TS_OFF, VLEVEL_M, "MW_LORAWAN_VERSION:   V%X.%X.%X\r\n
      ",
17          (uint8_t)(LORAWAN_VERSION_MAIN),
18          (uint8_t)(LORAWAN_VERSION_SUB1),
19          (uint8_t)(LORAWAN_VERSION_SUB2));
20
21  /* Get MW SubGhz_Phy info */
22  APP_LOG(TS_OFF, VLEVEL_M, "MW_RADIO_VERSION:     V%X.%X.%X\r\n
      ",
23          (uint8_t)(SUBGHZ_PHY_VERSION_MAIN),
24          (uint8_t)(SUBGHZ_PHY_VERSION_SUB1),
25          (uint8_t)(SUBGHZ_PHY_VERSION_SUB2));
26
27  /* Get LoRaWAN Link Layer info */
28  LmHandlerGetVersion(LORAMAC_HANDLER_L2_VERSION, &
      feature_version);
29  APP_LOG(TS_OFF, VLEVEL_M, "L2_SPEC_VERSION:      V%X.%X.%X\r\n
      ",
30          (uint8_t)(feature_version >> 24),
31          (uint8_t)(feature_version >> 16),
32          (uint8_t)(feature_version >> 8));
33
34  /* Get LoRaWAN Regional Parameters info */
35  LmHandlerGetVersion(LORAMAC_HANDLER_REGION_VERSION, &
      feature_version);
36  APP_LOG(TS_OFF, VLEVEL_M, "RP_SPEC_VERSION:      V%X-%X.%X.%X\
      r\n",
37          (uint8_t)(feature_version >> 24),
38          (uint8_t)(feature_version >> 16),
39          (uint8_t)(feature_version >> 8),
40          (uint8_t)(feature_version));
```

Version Logging: Retrieves and prints on serial the logs of the application, LoRaWAN middleware, and SubGhz PHY versions using the `APP_LOG` macro and respective version macros.

```
1  MX_X_CUBE_AI_Init();
2  APP_LOG(TS_OFF, VLEVEL_M, "X-CUBE-AI initialized\r\n");
3
4  UTIL_TIMER_Create(&TxLedTimer, LED_PERIOD_TIME,
     UTIL_TIMER_ONESHOT, OnTxTimerLedEvent, NULL);
```

```
5   UTIL_TIMER_Create(&RxLedTimer, LED_PERIOD_TIME,
     UTIL_TIMER_ONESHOT, OnRxTimerLedEvent, NULL);
6   UTIL_TIMER_Create(&JoinLedTimer, LED_PERIOD_TIME,
     UTIL_TIMER_PERIODIC, OnJoinTimerLedEvent, NULL);
7
8   if (FLASH_IF_Init(NULL) != FLASH_IF_OK)
9   {
10    Error_Handler();
11  }
12
13  /* USER CODE END LoRaWAN_Init_1 */
14
15  UTIL_TIMER_Create(&StopJoinTimer, JOIN_TIME,
     UTIL_TIMER_ONESHOT, OnStopJoinTimerEvent, NULL);
```

Module Initialization: Initializes X-CUBE-AI, timers (`TxLedTimer`, `RxLedTimer`, `JoinLedTimer`, `StopJoinTimer`), and checks the Flash interface initialization. Here the **Join timer** begins running, it will be used to wait a period between the attempts.

```
1   UTIL_SEQ_RegTask((1 << CFG_SEQ_Task_LmHandlerProcess),
     UTIL_SEQ_RFU, LmHandlerProcess);
2
3   UTIL_SEQ_RegTask((1 <<
     CFG_SEQ_Task_LoRaSendOnTxTimerOrButtonEvent), UTIL_SEQ_RFU,
     SendTxData);
4   UTIL_SEQ_RegTask((1 << CFG_SEQ_Task_LoRaStoreContextEvent),
     UTIL_SEQ_RFU, StoreContext);
5   UTIL_SEQ_RegTask((1 << CFG_SEQ_Task_LoRaStopJoinEvent),
     UTIL_SEQ_RFU, StopJoin);
```

Task Registration: Registers tasks for LoRaWAN handler processing (`LmHandlerProcess`), transmission (`SendTxData`), context storage (`StoreContext`), and stop join event handling (`StopJoin`). The use of this tasks is explained in Section 3.5.2.

```
1   /* Init Info table used by LmHandler*/
2   LoraInfo_Init();
3
4   /* Init the Lora Stack*/
```

```
5   LmHandlerInit(&LmHandlerCallbacks, APP_VERSION);
6
7   LmHandlerConfigure(&LmHandlerParams);
```

LmHandler Initialization: Initializes the LmHandler with callbacks (`LmHandlerCallbacks`) and application version (`APP_VERSION`), then configures handler parameters (`LmHandlerParams`).

```
1   /* USER CODE BEGIN LoRaWAN_Init_2 */
2   UTIL_TIMER_Start(&JoinLedTimer);
3
4   /* USER CODE END LoRaWAN_Init_2 */
5
6   LmHandlerJoin(ActivationType, ForceRejoin);
```

Timer Start and Join: Starts the `JoinLedTimer` and initiates a join procedure with specified activation type (`ActivationType`) and force rejoin flag (`ForceRejoin`).

```
1   if (EventType == TX_ON_TIMER)
2   {
3     /* send every time timer elapses */
4     UTIL_TIMER_Create(&TxTimer, TxPeriodicity,
     UTIL_TIMER_ONESHOT, OnTxTimerEvent, NULL);
5     UTIL_TIMER_Start(&TxTimer);
6   }
7   else
8   {
9     /* USER CODE BEGIN LoRaWAN_Init_3 */
10
11    /* USER CODE END LoRaWAN_Init_3 */
12  }
13
14  /* USER CODE BEGIN LoRaWAN_Init_Last */
15
16  /* USER CODE END LoRaWAN_Init_Last */
17 }
```

Transmission Timer: If `EventType` is `TX_ON_TIMER`, creates and starts a timer (`TxTimer`) for periodic transmission events (`OnTxTimerEvent`). In this case, the variable is set, otherwise transmission can be started only by pressing the predefined button.

X-CUBE-AI Initialization

The process of initialization is launched in `MX_LoRaWAN_Init()`. Its definition is written in `app_x-cube-ai.c`.

```
void MX_X_CUBE_AI_Init(void)
{
    /* USER CODE BEGIN 5 */
//  RTC_TimeTypeDef start;
//  start.Hours = 0;
//  start.Minutes = 0;
//  start.Seconds = 0;
//  start.SubSeconds = 0;

//  HAL_RTC_SetTime(&hrtc, &start, RTC_FORMAT_BIN);
//  HAL_RTC_MspInit(&hrtc);
  /*Initialize network with default activations*/
  ai_boostrap(data_activations0);
    /* USER CODE END 5 */
}
```

There are some commented lines, they were used to debug the execution time of the network. The essential part is processed at `ai_bootstrap()` which allocates the pool where the activation data must go, after every run.

```
static int ai_boostrap(ai_handle *act_addr)
{
  ai_error err;

  /* Create and initialize an instance of the model */
  err = ai_plantnetwork_create_and_init(&plantnetwork, act_addr
   , NULL);
  if (err.type != AI_ERROR_NONE) {
    ai_log_err(err, "ai_plantnetwork_create_and_init");
    return -1;
  }

```

```
12    ai_input = ai_plantnetwork_inputs_get(plantnetwork, NULL);
13    ai_output = ai_plantnetwork_outputs_get(plantnetwork, NULL);
14
15  #if defined(AI_PLANTNETWORK_INPUTS_IN_ACTIVATIONS)
16    /* In the case where "--allocate-inputs" option is used,
       memory buffer can be
17     * used from the activations buffer. This is not mandatory.
18     */
19    for (int idx=0; idx < AI_PLANTNETWORK_IN_NUM; idx++) {
20      data_ins[idx] = ai_input[idx].data;
21    }
22  #else
23    for (int idx=0; idx < AI_PLANTNETWORK_IN_NUM; idx++) {
24        ai_input[idx].data = data_ins[idx];
25    }
26  #endif
27
28  #if defined(AI_PLANTNETWORK_OUTPUTS_IN_ACTIVATIONS)
29    /* In the case where "--allocate-outputs" option is used,
       memory buffer can be
30     * used from the activations buffer. This is no mandatory.
31     */
32    for (int idx=0; idx < AI_PLANTNETWORK_OUT_NUM; idx++) {
33      data_outs[idx] = ai_output[idx].data;
34    }
35  #else
36    for (int idx=0; idx < AI_PLANTNETWORK_OUT_NUM; idx++) {
37      ai_output[idx].data = data_outs[idx];
38    }
39  #endif
40
41    return 0;
42  }
```

This function is the link point between the fixed part of the firmware and the one that is updated every time NN changes. These methods are declared in files named the same as NN, because they contain the C model of the network, with all the weights and structure of the neurons.

3.5.2 Sequencer Loop

After all peripherals have been initialized and configured, the `main()` function enters the main program loop. This loop typically contains the application

logic, including tasks such as reading sensor data, processing inputs, and communicating with other devices.

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */
    MX_LoRaWAN_Process();

    /* USER CODE BEGIN 3 */
}
```

The `MX_LoRaWAN_Process()` function is called within the main loop to handle LoRaWAN-related tasks. The main loop ensures that the microcontroller continues to operate and respond to events indefinitely, or until it is reset or powered off. The definition of the function is contained in `app_lorawan.c`.

```
void MX_LoRaWAN_Process(void)
{
  /* USER CODE BEGIN MX_LoRaWAN_Process_1 */

  /* USER CODE END MX_LoRaWAN_Process_1 */
  UTIL_SEQ_Run(UTIL_SEQ_DEFAULT);
  /* USER CODE BEGIN MX_LoRaWAN_Process_2 */

  /* USER CODE END MX_LoRaWAN_Process_2 */
}
```

The process consists in a call to `UTIL_SEQ_Run()` which serves as scheduler based on polling. The function, a core component of STM32 utilities, serves as a polling-based scheduler crucial for managing task execution in embedded applications. Part of the sequencer module, it enables developers to define, prioritize, and execute multiple tasks in an organized and efficient manner. When invoked, `UTIL_SEQ_Run()` enters an infinite loop, continuously checking for and executing pending tasks based on their priorities. This approach ensures that higher-priority tasks are handled first, enhancing the responsiveness and performance of the firmware. Moreover, the sequencer manages task dependencies, ensuring proper order of execution and task completion before starting new ones.

The tasks involved in this project are listed in `utilities_def.c`.
The `CFG_SEQ_Task_Id_t` enumeration defines task identifiers used by the application for scheduling and managing tasks within the STM32 microcontroller environment.

Since `CFG_SEQ_Task_LoRaStoreContextEvent` is not used by the application, we describe the purpose and function of three specific tasks:

- `CFG_SEQ_Task_LmHandlerProcess`
- `CFG_SEQ_Task_LoRaSendOnTxTimerOrButtonEvent`
- `CFG_SEQ_Task_LoRaStopJoinEvent`

```
1  /**
2    * This is the list of task id required by the application
3    * Each Id shall be in the range 0..31
4    */
5  typedef enum
6  {
7    CFG_SEQ_Task_LmHandlerProcess,
8    CFG_SEQ_Task_LoRaSendOnTxTimerOrButtonEvent,
9    CFG_SEQ_Task_LoRaStoreContextEvent,
10   CFG_SEQ_Task_LoRaStopJoinEvent,
11   /* USER CODE BEGIN CFG_SEQ_Task_Id_t */
12
13   /* USER CODE END CFG_SEQ_Task_Id_t */
14   CFG_SEQ_Task_NBR
15 } CFG_SEQ_Task_Id_t;
```

CFG_SEQ_Task_LmHandlerProcess The LoRaWAN handler task is responsible for handling the LoRaWAN protocol stack. This task processes various related events, such as managing network join requests, uplink and downlink message handling, and other protocol-specific tasks. It ensures that the stack operates correctly, maintaining seamless communication between the end device and the LoRaWAN network server.

CFG_SEQ_Task_LoRaSendOnTxTimerOrButtonEvent This task manages the transmission of data via LoRaWAN based on specific triggers such as a timer or a button press event. This task is crucial for applications that require periodic data transmission (e.g., sensor readings at regular intervals) or event-driven data transmission (e.g., sending data when a user presses

a button). It ensures timely and appropriate data transmission over the LoRaWAN network.

CFG_SEQ_Task_LoRaStopJoinEvent The LoRaWAN join task is used to handle events related to stopping the LoRaWAN network join process. This could be necessary in scenarios where the device needs to abort an ongoing join procedure due to specific conditions or constraints (e.g., failed join attempts, network restrictions, or user intervention). This task ensures that the join process can be gracefully terminated when required.

Each of these tasks plays an essential role in managing the various aspects of LoRaWAN application.

3.5.3 Join Task

The join task in the LoRaWAN system is mandatory for managing the process of joining a LoRaWAN network. It involves the `OnStopJoinTimerEvent` and `StopJoin` functions, which handle the stopping of the join process and switching between activation types.

StopJoin Function

The `StopJoin` function is responsible for stopping the current join process. It first stops the transmission timer and attempts to stop the LoRaMAC handler. If the handler stops successfully, it logs the status and toggles between ABP (Activation By Personalization) and OTAA (Over-The-Air Activation) modes, reconfigures the handler, and reinitiates the join process. Finally, it restarts both the transmission and it stops join timers.

```
1  static void StopJoin(void)
2  {
3    /* USER CODE BEGIN StopJoin_1 */
4    //HAL_GPIO_WritePin(LED1_GPIO_Port, LED1_Pin, GPIO_PIN_SET);
       /* LED_BLUE */
5    //HAL_GPIO_WritePin(LED2_GPIO_Port, LED2_Pin, GPIO_PIN_SET);
       /* LED_GREEN */
6    //HAL_GPIO_WritePin(LED3_GPIO_Port, LED3_Pin, GPIO_PIN_SET);
       /* LED_RED */
7    /* USER CODE END StopJoin_1 */
8
9    UTIL_TIMER_Stop(&TxTimer);
10
```

```
11    if (LORAMAC_HANDLER_SUCCESS != LmHandlerStop())
12    {
13      APP_LOG(TS_OFF, VLEVEL_M, "LmHandler Stop on going ...\r\n"
       );
14    }
15    else
16    {
17      APP_LOG(TS_OFF, VLEVEL_M, "LmHandler Stopped\r\n");
18       if (LORAWAN_DEFAULT_ACTIVATION_TYPE == ACTIVATION_TYPE_ABP)
19       {
20          ActivationType = ACTIVATION_TYPE_OTAA;
21          APP_LOG(TS_OFF, VLEVEL_M, "LmHandler switch to OTAA mode\
      r\n");
22       }
23       else
24       {
25          ActivationType = ACTIVATION_TYPE_ABP;
26          APP_LOG(TS_OFF, VLEVEL_M, "LmHandler switch to ABP mode\r
     \n");
27       }
28       LmHandlerConfigure(&LmHandlerParams);
29       LmHandlerJoin(ActivationType, true);
30       UTIL_TIMER_Start(&TxTimer);
31    }
32    UTIL_TIMER_Start(&StopJoinTimer);
33    /* USER CODE BEGIN StopJoin_Last */
34
35    /* USER CODE END StopJoin_Last */
36  }
```

OnStopJoinTimerEvent Function

The `OnStopJoinTimerEvent` function is triggered when the stop join timer event occurs. It checks the current activation type and, if it matches the default LoRaWAN activation type, it schedules the `CFG_SEQ_Task_LoRaStopJoinEvent` task with priority 0.

```
1   static void OnStopJoinTimerEvent(void *context)
2   {
3     /* USER CODE BEGIN OnStopJoinTimerEvent_1 */
4
5     /* USER CODE END OnStopJoinTimerEvent_1 */
6     if (ActivationType == LORAWAN_DEFAULT_ACTIVATION_TYPE)
```

```
7   {
8     UTIL_SEQ_SetTask((1 << CFG_SEQ_Task_LoRaStopJoinEvent),
      CFG_SEQ_Prio_0);
9   }
10  /* USER CODE BEGIN OnStopJoinTimerEvent_Last */
11  //HAL_GPIO_WritePin(LED1_GPIO_Port, LED1_Pin, GPIO_PIN_RESET)
      ; /* LED_BLUE */
12  //HAL_GPIO_WritePin(LED2_GPIO_Port, LED2_Pin, GPIO_PIN_RESET)
      ; /* LED_GREEN */
13  //HAL_GPIO_WritePin(LED3_GPIO_Port, LED3_Pin, GPIO_PIN_RESET)
      ; /* LED_RED */
14  /* USER CODE END OnStopJoinTimerEvent_Last */
15 }
```

The join task involves stopping the current join process and managing the activation type switching between OTAA and ABP. The `OnStopJoinTimerEvent` function schedules the stop join task, while the `StopJoin` function handles the stopping process, logging, and reconfiguring of the LoRaMAC handler to ensure a smooth and efficient join process.

3.5.4 Transmission Task

The transmission task in the LoRaWAN system demands the `SendTxData` function, which is responsible for collecting sensor data, predicting network conditions, buffering the data, and transmitting it over the LoRaWAN network. This process is divided into four main phases: Measurement, Network Prediction, Buffering, and Transmission.

Measurement

In the Measurement phase, various sensors are initialized, and data is collected. This involves reading temperature, relative humidity, light intensity, impedance, and watermark levels from connected sensors.

Initially, the function initializes the ADC and I2C peripherals and reads the temperature and humidity values using the HDC2080 sensor. It also checks for the presence of the OPT3001 light sensor and reads the light intensity if the sensor is connected.

```
1 static void SendTxData(void)
2 {
3   uint16_t rh_int = 0;
```

```
4   uint16_t lux_int = 0;
5   uint16_t temp_int = 0;
6   uint32_t *output_value_ptr;
7
8   /* ADC and I2C Initialization */
9   MX_ADC_Init();
10  MX_I2C2_Init();
11  MX_TIM1_Init();
12  MX_TIM2_Init();
13
14  /* Read temperature and humidity */
15  temp_int = hdc2080_readtemp();
16  rh_int = hdc2080_readrh();
17
18  /* Check if the OPT3001 sensor is connected */
19  if (read_devid() == 12289) {
20    if (opt3001_init() == 1) {
21      lux_int = opt3001_readdata();
22    }
23  } else {
24    lux_int = 0;
25  }
```

To measure impedance and watermark levels, the function reconfigures and uses timers. Timer 1 (TIM1) is set up for input capture, and Timer 2 (TIM2) is set up for output compare. These timers are started to measure the time interval between signal transitions, which is used to calculate the frequency of the signal corresponding to the impedance and watermark levels.

For impedance measurement, the impedance module is enabled, and the timers are started. The function waits until the measurement is complete, then stops the timers and disables the impedance module.

```
1   /* Start and stop timers for impedance measurement */
2   HAL_TIM_IC_Start_IT(&htim1, TIM_CHANNEL_1);
3   HAL_TIM_OC_Start_IT(&htim2, TIM_CHANNEL_1);
4   HAL_GPIO_WritePin(GPIOA, IMP_EN_Pin, GPIO_PIN_SET);
5   while (flag_meas == 0) {}
6   HAL_TIM_IC_Stop_IT(&htim1, TIM_CHANNEL_1);
7   HAL_TIM_OC_Stop_IT(&htim2, TIM_CHANNEL_1);
8   HAL_GPIO_WritePin(GPIOA, IMP_EN_Pin, GPIO_PIN_RESET);
9
10  /* Evaluate impedance */
```

```
11   impedance = eval_freq();
12   flag_meas = 0;
```

Similarly, for watermark measurement, the watermark module is enabled and the timers are started. The function waits until the measurement is complete, then stops the timers and disables the watermark module.

```
1   /* Start and stop timers for watermark measurement */
2   HAL_TIM_IC_Start_IT(&htim1, TIM_CHANNEL_2);
3   HAL_TIM_OC_Start_IT(&htim2, TIM_CHANNEL_1);
4   HAL_GPIO_WritePin(GPIOC, EN_WT_Pin, GPIO_PIN_SET);
5   while (flag_meas == 0) {}
6   HAL_TIM_IC_Stop_IT(&htim1, TIM_CHANNEL_2);
7   HAL_TIM_OC_Stop_IT(&htim2, TIM_CHANNEL_1);
8   HAL_GPIO_WritePin(GPIOC, EN_WT_Pin, GPIO_PIN_RESET);
9
10  /* Evaluate watermark */
11  watermark = (uint16_t) eval_freq();
```

The `eval_freq` function is used to calculate the frequency based on the captured timer values, which represents the impedance and watermark levels. It calculates the time difference between two consecutive timer captures and uses this difference to compute the frequency.

```
1  uint32_t eval_freq()
2  {
3    uint32_t freq = 0;
4    float fr_tmp;
5
6    if(old_capture != new_capture)
7    {
8      if(new_capture > old_capture)
9      {
10       diffCapture = new_capture - old_capture;
11       fr_tmp = 2000000 / diffCapture;
12       freq = (uint32_t) fr_tmp;
13     }
14     else
15     {
16       diffCapture = 65535 - old_capture + new_capture;
17       fr_tmp = 2000000 / diffCapture;
18       freq = (uint32_t) fr_tmp;
```

```
19      }
20    }
21
22    return freq;
23  }
```

The `eval_freq` function calculates the frequency by considering the difference between the new and old capture values. If the new capture value is greater than the old value, the difference is calculated directly. If the new capture value has been wrapped, the difference is adjusted accordingly. The calculated frequency is then returned as the result.

Network Prediction

In the Network Prediction phase, the system processes the collected data using a neural network to predict network conditions. This is achieved using the X-CUBE-AI library. The `MX_X_CUBE_AI_Process` function orchestrates the neural network processing, involving three main steps: acquiring and preprocessing input data, running the inference engine, and post-processing the predictions.

```
1  void MX_X_CUBE_AI_Process(void)
2  {
3    int res = -1;
4
5    APP_PRINTF("\r\nNNetwork> Starting neural network process...\
       r\n");
6
7    if (plantnetwork) {
8        /* 1 - acquire and pre-process input data */
9        res = acquire_and_process_data(data_ins);
10       /* 2 - process the data - call inference engine */
11       if (res == 0){
12         res = ai_run();
13       }
14       /* 3- post-process the predictions */
15       if (res == 0)
16         res = post_process(data_outs);
17   }
18
19   if (res) {
20     ai_error err = {AI_ERROR_INVALID_STATE,
       AI_ERROR_CODE_NETWORK};
21     ai_log_err(err, "Process has FAILED");
```

```
22    }
23
24    APP_PRINTF("NNetwork> Neural network process ended\r\n\n");
25  }
```

The `acquire_and_process_data` function prepares the input data for the neural network. It collects the impedance and watermark values, normalizes them, and formats them as inputs for the neural network model.

```
1  int acquire_and_process_data(ai_i8* data[])
2  {
3    static ai_float f[AI_PLANTNETWORK_IN_1_SIZE];
4    f[1] = (ai_float)impedance;
5    f[0] = (ai_float)watermark;
6
7    normalize(f);
8
9    for (int idx = 0; idx < AI_PLANTNETWORK_IN_NUM; idx++)
10       for(int i = 0; i < AI_PLANTNETWORK_IN_1_SIZE; i++)
11         ((ai_float*)(data[idx]))[i] = f[i];
12
13   APP_PRINTF("NNetwork> Received: %f\t%f\r\n", (float)(((
      ai_float*)*data)[0]), (float)(((ai_float*)*data)[1]));
14   return 0;
15 }
```

The `ai_run` function executes the neural network inference. It passes the preprocessed input data to the neural network and retrieves the output predictions.

```
1  static int ai_run(void)
2  {
3    ai_i32 batch;
4
5    batch = ai_plantnetwork_run(plantnetwork, ai_input, ai_output
      );
6    if (batch != 1) {
7      ai_log_err(ai_plantnetwork_get_error(plantnetwork), "
     ai_plantnetwork_run");
8      return -1;
9    }
10
```

```
11    return 0;
12  }
```

The `post_process` function interprets the neural network output, determining the prediction result. It processes the output to determine whether the prediction exceeds a certain threshold, which indicates the network condition.

```
1  int post_process(ai_i8* data[])
2  {
3    ai_float output;
4
5    output = ((ai_float*)*data)[0];
6    prediction = output >= 0.5;
7    APP_PRINTF("NNetwork> Network Output:%.4e\r\n", *((ai_float*)
       &output));
8    APP_PRINTF("NNetwork> Predicted: %d\r\n", prediction);
9    return 0;
10 }
```

Buffering

During the Buffering phase, the sensor data and prediction results are formatted and stored in the transmission buffer.

```
1    uint32_t i = 0;
2    AppData.Buffer[i++] = (uint8_t)((rh_int >> 8) & 0xFF);
3    AppData.Buffer[i++] = (uint8_t)(rh_int & 0xFF);
4    AppData.Buffer[i++] = (uint8_t)((lux_int >> 8) & 0xFF);
5    AppData.Buffer[i++] = (uint8_t)(lux_int & 0xFF);
6    AppData.Buffer[i++] = (uint8_t)((temp_int >> 8) & 0xFF);
7    AppData.Buffer[i++] = (uint8_t)(temp_int & 0xFF);
8    AppData.Buffer[i++] = (uint8_t)((impedance >> 16) & 0xFF);
9    AppData.Buffer[i++] = (uint8_t)((impedance >> 8) & 0xFF);
10   AppData.Buffer[i++] = (uint8_t)(impedance & 0xFF);
11   AppData.Buffer[i++] = (uint8_t)((watermark >> 8) & 0xFF);
12   AppData.Buffer[i++] = (uint8_t)(watermark & 0xFF);
13   AppData.Buffer[i++] = (uint8_t)(prediction & 0xFF);
14   AppData.Buffer[i++] = (uint8_t)(((uint8_t*)output_value_ptr)
       [3] & 0xFF);
15   AppData.Buffer[i++] = (uint8_t)(((uint8_t*)output_value_ptr)
       [2] & 0xFF);
```

```
16    AppData.Buffer[i++] = (uint8_t)(((uint8_t*)output_value_ptr)
      [1] & 0xFF);
17    AppData.Buffer[i++] = (uint8_t)(((uint8_t*)output_value_ptr)
      [0] & 0xFF);
18    AppData.BufferSize = i;
```

Transmission

Finally, in the Transmission phase, the buffered data is sent over the LoRaWAN network. The function checks if the network is ready and then sends the data. It also handles duty cycle restrictions by adjusting the transmission timer.

```
1   LmHandlerErrorStatus_t status = LORAMAC_HANDLER_ERROR;
2   UTIL_TIMER_Time_t nextTxIn = 0;
3
4   if (LmHandlerIsBusy() == false)
5   {
6     status = LmHandlerSend(&AppData, LmHandlerParams.
      IsTxConfirmed, false);
7     if (LORAMAC_HANDLER_SUCCESS == status)
8     {
9       APP_LOG(TS_ON, VLEVEL_L, "SEND REQUEST\r\n");
10    }
11    else if (LORAMAC_HANDLER_DUTYCYCLE_RESTRICTED == status)
12    {
13      nextTxIn = LmHandlerGetDutyCycleWaitTime();
14      if (nextTxIn > 0)
15      {
16        APP_LOG(TS_ON, VLEVEL_L, "Next Tx in  : ~%d second(s)\r
      \n", (nextTxIn / 1000));
17      }
18    }
19  }
20
21  if (EventType == TX_ON_TIMER)
22  {
23    UTIL_TIMER_Stop(&TxTimer);
24    UTIL_TIMER_SetPeriod(&TxTimer, MAX(nextTxIn, TxPeriodicity)
      );
25    UTIL_TIMER_Start(&TxTimer);
26  }
27}
```

3.5.5 Reception Task

The reception task in the LoRaWAN system is responsible for handling incoming data packets received via the LoRaWAN network. This task is invoked by a LoRaWAN interrupt and is implemented in the `OnRxData` function. The function processes the received data, identifies the port on which the data was received, and takes appropriate actions based on the content of the received data.

```
1  static void OnRxData(LmHandlerAppData_t *appData,
       LmHandlerRxParams_t *params)
2  {
3    uint8_t RxPort = 0;
4
5    if (params != NULL)
6    {
7      UTIL_TIMER_Start(&RxLedTimer);
8
9      if (params->IsMcpsIndication)
10     {
11       if (appData != NULL)
12       {
13         RxPort = appData->Port;
14         if (appData->Buffer != NULL)
15         {
16           switch (appData->Port)
17           {
18             case LORAWAN_SWITCH_CLASS_PORT:
19               if (appData->BufferSize == 1)
20               {
21                 switch (appData->Buffer[0])
22                 {
23                   case 0:
24                     LmHandlerRequestClass(CLASS_A);
25                     break;
26                   case 1:
27                     LmHandlerRequestClass(CLASS_B);
28                     break;
29                   case 2:
30                     LmHandlerRequestClass(CLASS_C);
31                     break;
32                   default:
33                     break;
34                 }
```

```
35                }
36                break;
37
38              case LORAWAN_USER_APP_PORT:
39                if (appData->BufferSize == 1)
40                {
41                  if (appData->Buffer[0] == 0)
42                  {
43                    NVIC_SystemReset();
44                  }
45                }
46                if (appData->BufferSize == 2)
47                {
48                  new_lora_dc = (((appData->Buffer[0]) << 8) | (
    appData->Buffer[1])) * 1000;
49                  OnTxPeriodicityChanged(new_lora_dc);
50                }
51                break;
52
53              default:
54                break;
55            }
56          }
57        }
58      }
59
60      if (params->RxSlot < RX_SLOT_NONE)
61      {
62        APP_LOG(TS_OFF, VLEVEL_H, "###### D/L FRAME:%04d | PORT:%
    d | DR:%d | SLOT:%s | RSSI:%d | SNR:%d\r\n",
63                params->DownlinkCounter, RxPort, params->Datarate
    , slotStrings[params->RxSlot],
64                params->Rssi, params->Snr);
65      }
66    }
67  }
```

Overview The `OnRxData` function is triggered by a LoRaWAN interrupt when data is received. It processes the received data packet, evaluates the port on which the data was received, and performs specific actions based on the content and size of the data.

Processing Received Data When the `OnRxData` function is invoked, it first checks if the `params` structure is not null. If valid parameters are available, it starts the reception LED timer using `UTIL_TIMER_Start(&RxLedTimer)` to indicate that data reception is in progress.

MCPS Indication Handling The function then checks if the `IsMcpsIndication` flag in the `params` structure is set, indicating that a MAC-layer indication has occurred. If so, it proceeds to evaluate the received application data (`appData`).

Port-Based Data Handling The function identifies the port on which the data was received (`RxPort = appData->Port`). Based on the port number, it performs different actions:

- `LORAWAN_SWITCH_CLASS_PORT`: This port is used for switching the LoRaWAN device class. If the received data size is 1 byte, it switches the class based on the value of the byte (0 for Class A, 1 for Class B, and 2 for Class C).
- `LORAWAN_USER_APP_PORT`: This port is used for user-defined application commands. If the received data size is 1 byte and the value is 0, the microcontroller is reset using `NVIC_SystemReset()`. If the received data size is 2 bytes, it updates the LoRa duty cycle based on the received value and calls `OnTxPeriodicityChanged` with the new duty cycle.

Logging Received Data If the data was received in a valid Rx slot (`params->RxSlot < RX_SLOT_NONE`), the function logs detailed information about the received frame, including the downlink frame counter, port number, data rate, slot type, RSSI, and SNR.

3.6 LoRaWAN Configuration

Configuring the credentials for a LoRaWAN node is a mandatory step in the setup of the FW and it ensures secure and reliable communication within the LoRaWAN network. Properly setting these credentials helps to establish the identity of the device, allowing it to join and communicate with the network securely. This section highlights the significance of this process and illustrates it with specific lines from the `se-identity.h` file.

3.6.1 Overview of Credentials

In a LoRaWAN network, each node must have a unique set of credentials that include a Device EUI (Extended Unique Identifier), Join EUI (formerly known as App EUI), and application keys. These credentials ensure that the device can be authenticated by the network server and that the data transmitted and received is encrypted.

Device EUI

The Device EUI is a unique identifier for each LoRaWAN device. It is typically a 64-bit address that is used to uniquely identify the device within the network.

```
/* Device EUI */
#define LORAWAN_DEVICE_EUI                                          { 0
    x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 }
```

Join EUI

The Join EUI (formerly App EUI) is a global application identifier used during the join procedure to identify the application provider of the device.

```
/* Join EUI */
#define LORAWAN_JOIN_EUI                                            { 0
    x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 }
```

Application Key

The application key (AppKey) is a 128-bit AES encryption key specific to the device, used to secure the communication between the device and the network server.

```
/* Application root key */
#define LORAWAN_APP_KEY                                             { 0
    x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , \
                                                                      0
    x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 }
```

3.6.2 Security and Authentication

Setting these credentials correctly is crucial for the following reasons:

- **Authentication:** The Device EUI and Join EUI are used to authenticate the device during the join process. This ensures that only legitimate devices can join the network.
- **Data Integrity and Confidentiality:** The AppKey is used to encrypt the data transmitted between the device and the network server, ensuring that the data is not tampered with or intercepted by unauthorized entities.
- **Network Management:** Proper credential configuration helps network servers manage and identify devices efficiently, facilitating network scalability and maintenance.

3.6.3 Example Configuration

Below is an example configuration from the `se-identity.h` file, which shows how the credentials are defined and initialized.

```
/* Device EUI */
#define LORAWAN_DEVICE_EUI                                          { 0
    x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 }

/* Join EUI */
#define LORAWAN_JOIN_EUI                                            { 0
    x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 }

/* Application root key */
#define LORAWAN_APP_KEY                                             { 0
    x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , \
                                                                      0
    x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 }

/* Network root key */
#define LORAWAN_NWK_KEY                                             { 0
    x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , \
                                                                      0
    x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 }
```

Chapter 4

Toolchain Description

In this chapter, we describe the toolchain used for the development and deployment of neural network firmware designed to monitor plant health by impedance analysis. The process encompasses several stages, each critical to ensuring the firmware's accuracy and functionality.

The toolchain is divided into three main phases: Training, Building, and Flashing, as illustrated in Figure 4.1.

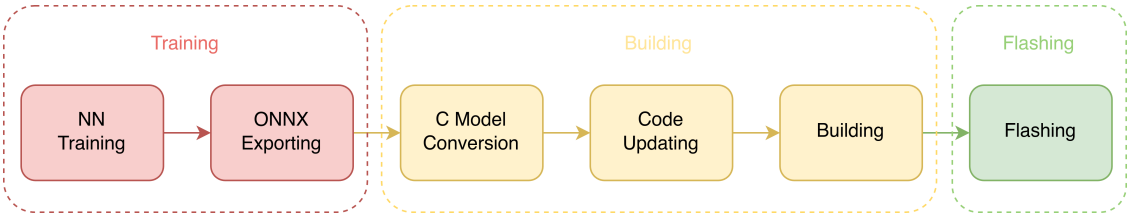


Figure 4.1: Toolchain for neural network firmware development

Dependencies

In order to run the toolchain, some dependency must be satisfied.

Python v3.12 Python language interpreter

Anaconda Data science Python environment manager

GNU Make standard build tool to automate the process

gcc-arm-none-eabi GNU Arm embedded toolchain to build Arm binaries

X-CUBE-AI STM software to create C models from ONNX

STM32CubeProgrammer official STM microcontrollers flasher tool

4.1 Training

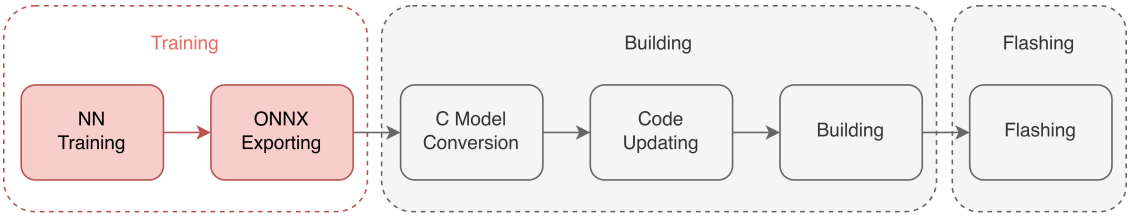


Figure 4.2: Toolchain for neural network firmware development - Training highlighted

The training phase involves the preparation of the neural network model. This begins with the neural network (NN) training, where the model learns from the data collected. The trained model is then exported in the ONNX format, which facilitates compatibility with various hardware and software environments.

4.1.1 NN Training

A framework developed previously is used to train the NN. The latter is *sw-ml-framework* developed by eLiONS doctoral students. This one is written in Python language, using a popular ML library called PyTorch. [27][28]

Not only that, a series of support libraries have been used, all listed in following description:

PyTorch
: PyTorch is an open-source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing. In this project, PyTorch is employed for developing and training neural network models due to its flexibility and dynamic computational graph support.

NumPy
: NumPy is a fundamental package for scientific computing in Python. It provides support for arrays, matrices, and many mathematical functions to operate on these data structures efficiently. In this project, NumPy is used extensively for numerical calculations and data manipulation.

Pandas

Pandas is a powerful data manipulation and analysis library for Python. It offers data structures like DataFrame, which allows for easy manipulation, cleaning, and analysis of structured data. In this project, Pandas is used to handle large datasets, perform data cleaning, and prepare data for analysis.

Scikit-learn
Scikit-learn is a Python machine learning library. In this project, Scikit-learn is used to implement and train neural network models used in the plant health monitoring system.

Matplotlib
Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. In this project, Matplotlib is used to visualize the data and the results of the analysis.

SciPy
SciPy is an open-source Python library used for scientific and technical computing. In this project, SciPy is used for advanced mathematical and statistical computations.

File description

Figure 4.3 presents a comprehensive analysis of the structure of *sw-ml-framework*. Following this, an in-depth explanation of the roles of these files is given:

Data is the directory in which input *.csv* files are stored to be processed by the framework

ml_framework_env[_cuda].yml are files to reproduce the exact Python environment in Anaconda, a scientific environment manager for Python.

README.md simple README file, generated by GitLab, an online software versioning server

results output folder, it contains a snapshot of each simulation launched

calibration is used by DataManipulation.py, in case of *SWP* feature selected it does the conversion from raw data to kPa, using its LUTs written in *.csv* file format.

DataManipulation.py/Dataset.py contain function to manage and process Pandas DataFrames, in order to prepare them for the convertion to input tensors.

single_simulation.py/single.py define all routines to launch training loops, manage batch subsets, measure evaluation metrics and export ONNX net.

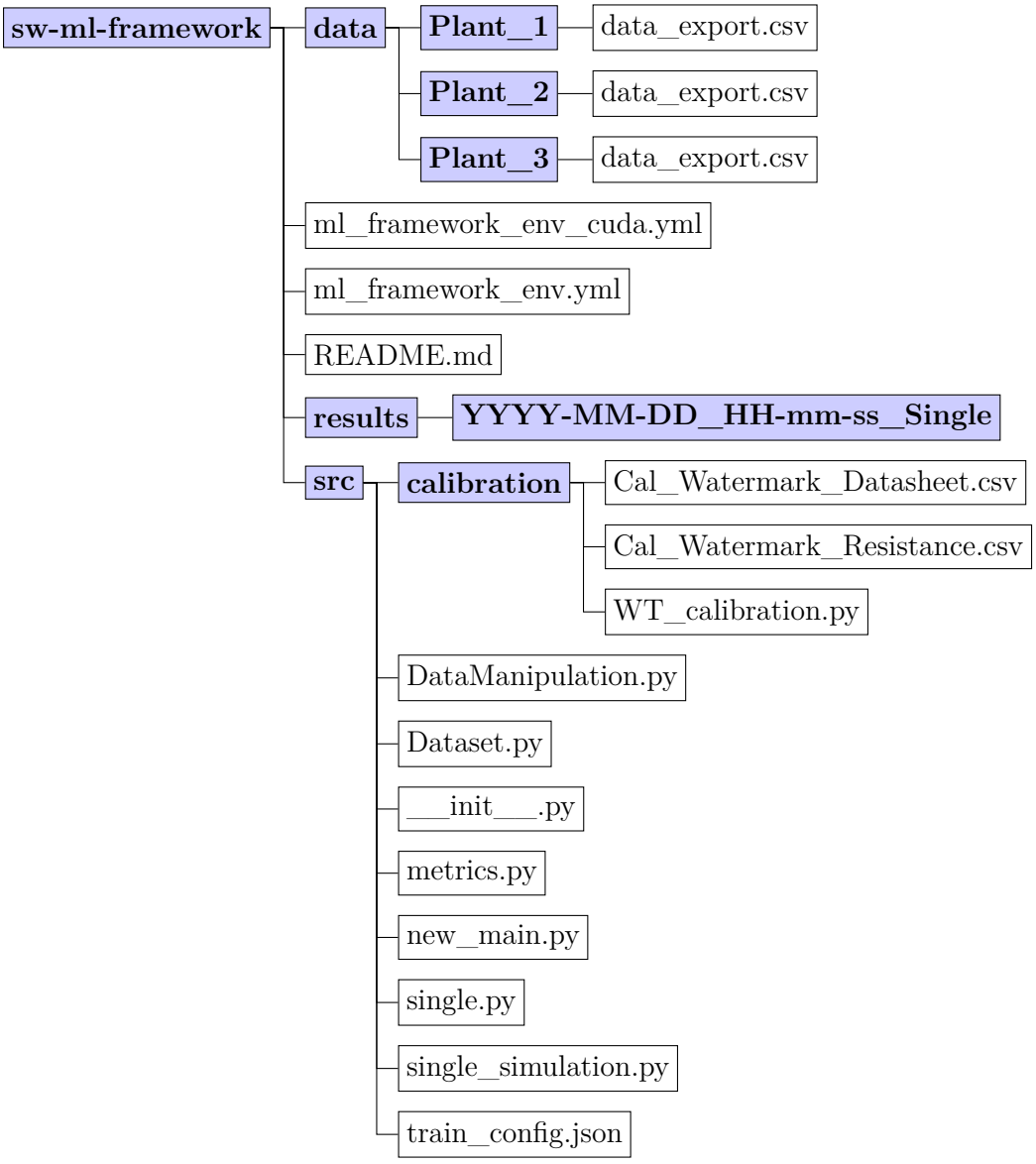


Figure 4.3: sw-ml-framework file tree

metrics.py permit to define user metrics formulae.

train_config.json is the file in which all settings are defined, such as plant IDs to use, the date of the subsets and the type of simulation to run.

Anaconda setup

The process of installing all the previously listed libraries is automated using Anaconda. In files *.yml* the lists of packages that the software needs to install

are contained to get the framework ready to work. The environment can be selected between a CPU or GPU based, but the latter is still WIP. Once the suited configuration is chosen, it can be installed through command:

Listing 4.1: Example of Anaconda environment installation command

```
conda env create -n <environment-name> ml_framework_env[_cuda].yml
```

where *<environment-name>* is an arbitrary name.

Training flow

Figure 4.4 illustrates the ASM chart of the training workflow. To summarize, the process starts with *DataManipulation.py*, which reads and preprocesses data from *train_config.json* to ensure it meets analytical criteria. Subsequently, *Dataset.py* defines a custom dataset class to enable efficient data handling within the PyTorch ecosystem. *new_main.py* then coordinates the entire workflow, including data loading, model training, and evaluation. Finally, *single.py* contains the neural network architecture and executes the training loop, leveraging PyTorch's functions to optimize the model with the processed data.

CUDA optimization

During the manipulation, some improvements have been tried to test the potential of the framework. One of these was CUDA optimizations, since PyTorch has embedded supports for the devices. After many trial and errors the code is ready to use, but with the following descripted issues. Inside the inner loop, called "_mini_batch()" there is a call to a function "predict()". This function moves the entire tensor content back to the CPU, causing a large overhead on high values of epochs ($N_{epochs} > 100$). This high latency is caused by transport, which is doubled due to this fact. Despite this fact, a positive improvement signal has been reported in Figures 4.5 and 4.6, to convince the future of this feature.

The graphs show the reporting of a *NeuronSweep* simulation. The task was to vary the number of neurons logarithmically in a structure like $[2, N, M, 1]$ where:

$$N, M \in \{10^1, 10^2, 10^3, 10^4\} \tag{4.1}$$

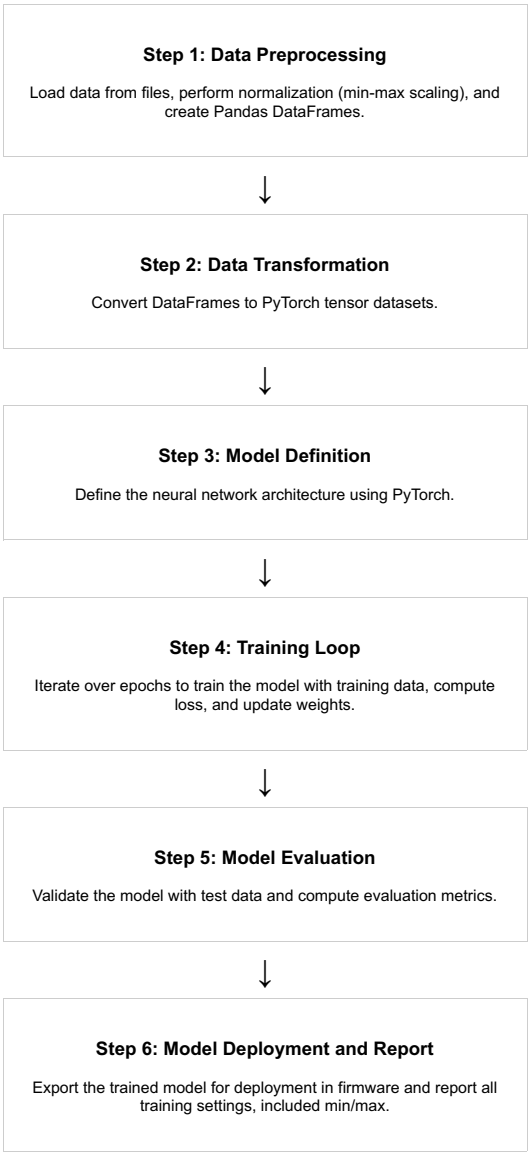


Figure 4.4: NN training ASM chart

2 Is the number of input of the NN
N Is the number of neurons of the first layer
M Is the number of neurons of the second layer
1 Is the number of outputs

However, the results are promising for training with a large number of neurons|

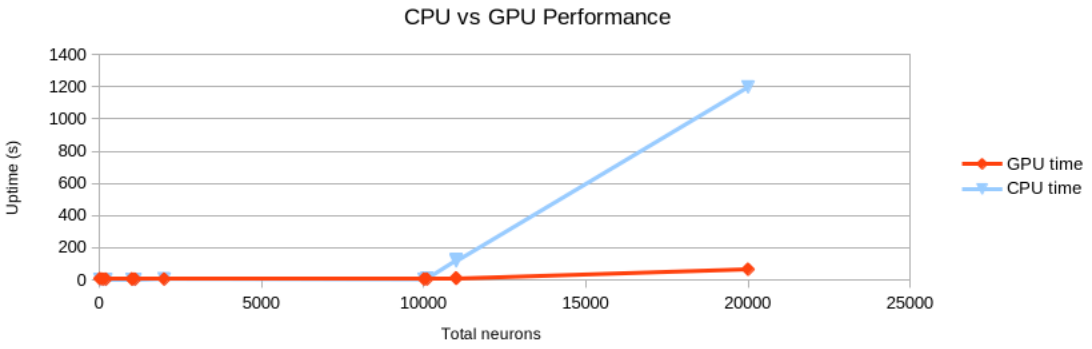


Figure 4.5: CPU vs GPU memory allocation

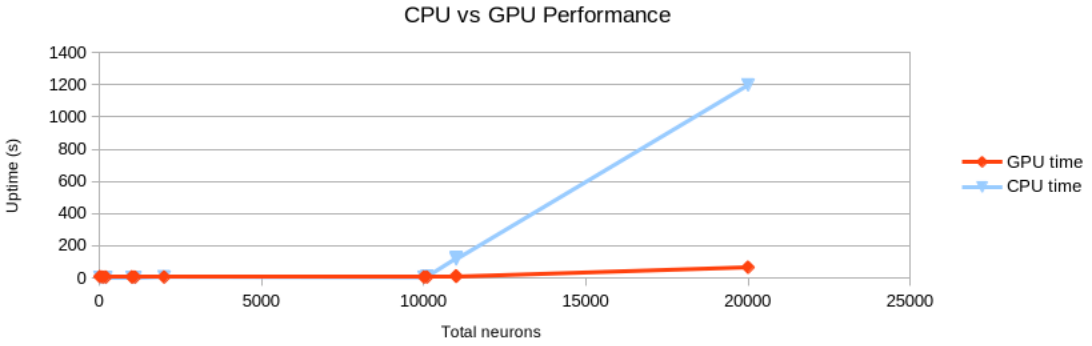


Figure 4.6: CPU vs GPU uptime

This is possible only for high performance HW, but it consumes more power and this is not the object of the thesis.

Normalization bounds reporting

The normalization of minimum and maximum values is a crucial step in the preprocessing phase, ensuring that all data features are on a similar scale and enhancing the performance of the neural network. This normalization process is carried out within the *DataManipulation.py* script. Specifically, the script reads the raw data from the specified files and computes the minimum and maximum values for each feature. The data is then scaled to a range between 0 and 1 using the formula:

$$\text{normalized\_value} = \frac{\text{value} - \text{min}}{\text{max} - \text{min}}$$

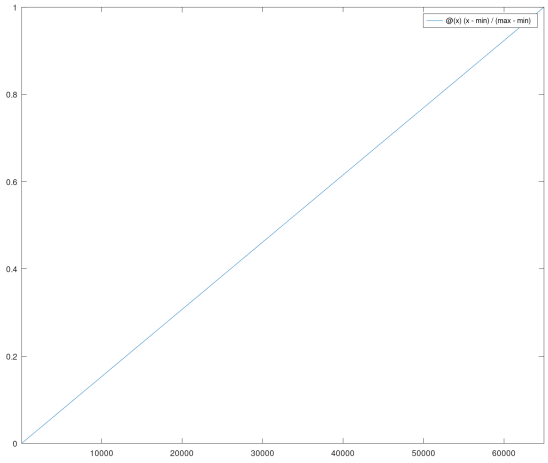


Figure 4.7: Example of normalization applied to $[min, max] = [32{,}64963]$

This transformation is applied to each feature in the dataset, producing normalized DataFrames that are subsequently loaded into the PyTorch tensorLoadset. This ensures that the data fed into the neural network during the *Simple-Training()* algorithm is appropriately scaled, facilitating more efficient training and improved model accuracy. The flow proceeds from reading the data and normalizing in *DataManipulation.py* to loading and handling by the custom data set class in *Dataset.py*, and finally to model training and evaluation in *new_main.py*. During this phase, the normalization bounds are stored in a variable.

4.1.2 ONNX exporting

Once the training is finished, the model needs to be exported, in order to get saved on a file. The target format for NN is ONNX.

Figure 4.8: ONNX logo [29][30]

Open Neural Network Exchange (ONNX) is an open source format designed to facilitate the interchangeability of machine learning models between different

frameworks. Developed by Microsoft and Facebook in 2017, ONNX aims to streamline the deployment process by providing a universal standard that supports interoperability among various deep learning tools such as PyTorch, TensorFlow, and Caffe2. By adopting ONNX, developers can leverage the strengths of multiple platforms throughout the model lifecycle, from training to inference, without the need for extensive conversion or redevelopment.

In order to export to this specific format, some changes have been applied to the original code. Initially, there was an individual Python script to achieve this feature, but all it was doing was reloading the model from the standard PyTorch one which is *.pth*. This implies using two rendundant disk accesses, since the scope of the framework is to produce ONNX from the beginning. Then a new method for the class *Simple* in file *single.py* of the framework was created (*export_to_onnx* in order to take the model, already loaded in memory, and export directly to target format.

Listing 4.2: Example of *minmax.json* report, with SWP and Impedance as features

```
[
    {
        "SWP": {
            "max": 2.2823558463946396,
            "min": -78.76520129153818
        }
    },
    {
        "Impedance": {
            "max": 51891.0,
            "min": 14883.0
        }
    }
]
```

All final files are stored in a specific folder under *result*, which reports data and time of the beginning of the training. All settings from *train_config.json*, the values of the minimum and maximum of the data set and a small report (in the case of *NeuronSweep* mode) are saved for the estimation of the results.

In Figure 4.9 there is an example of a typical output of the framework. In folder *plots* there are all metrics plots, with the number of epoch iterations on x-axis. There is also a preview of the predicted value of test plants.

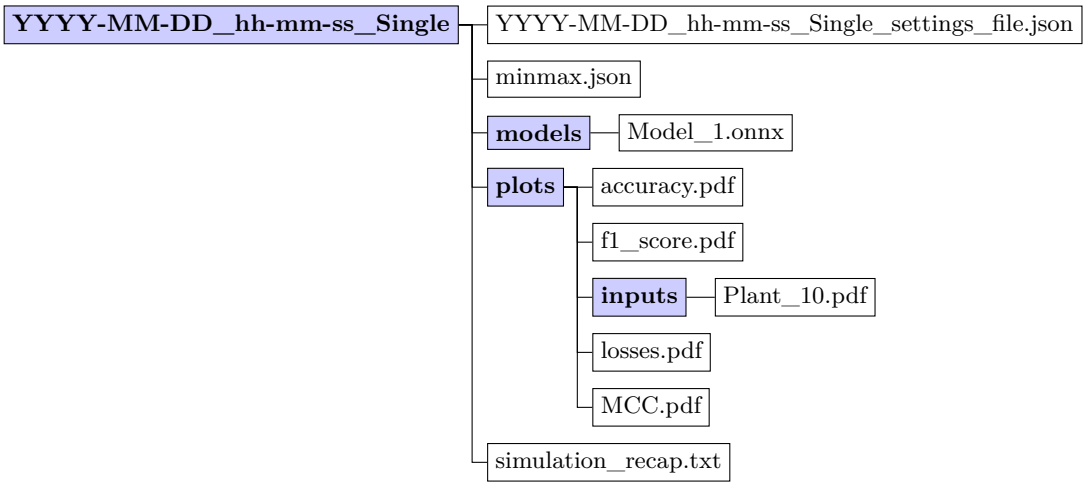


Figure 4.9: sw-ml-framework result output folder file tree

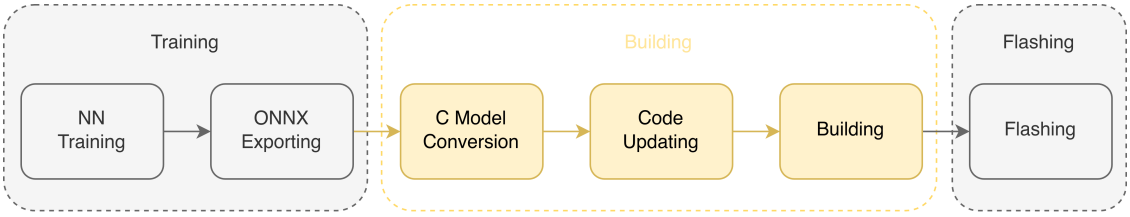


Figure 4.10: Toolchain for neural network firmware development - Building highlighted

4.2 Building

The building phase translates the trained neural network model into a format suitable for the microcontroller. The ONNX model is first converted into a C model. Following this, the code is updated to integrate the neural network model into the firmware. The final step in this phase is building the firmware, which prepares it for deployment on the hardware.

In Figure 4.11 the scripts that are involved in the building stage are shown. They serve for the following scopes:

set-env.sh is **mandatory to set**, it contains all shortcuts to system executables, listed below:

- STM32CubeProgrammer
- X-CUBE-AI

build.sh launches all scripts sequentially. Its default behaviour is to prompt for every feature, but it can be bypassed using the option *-y*.

update-network.sh uses X-CUBE-AI to generate the updated NN model written in C language.

update-normalization.sh updates minimum and maxim boundaries of the input features. It is launched automatically by *update-network.sh*.

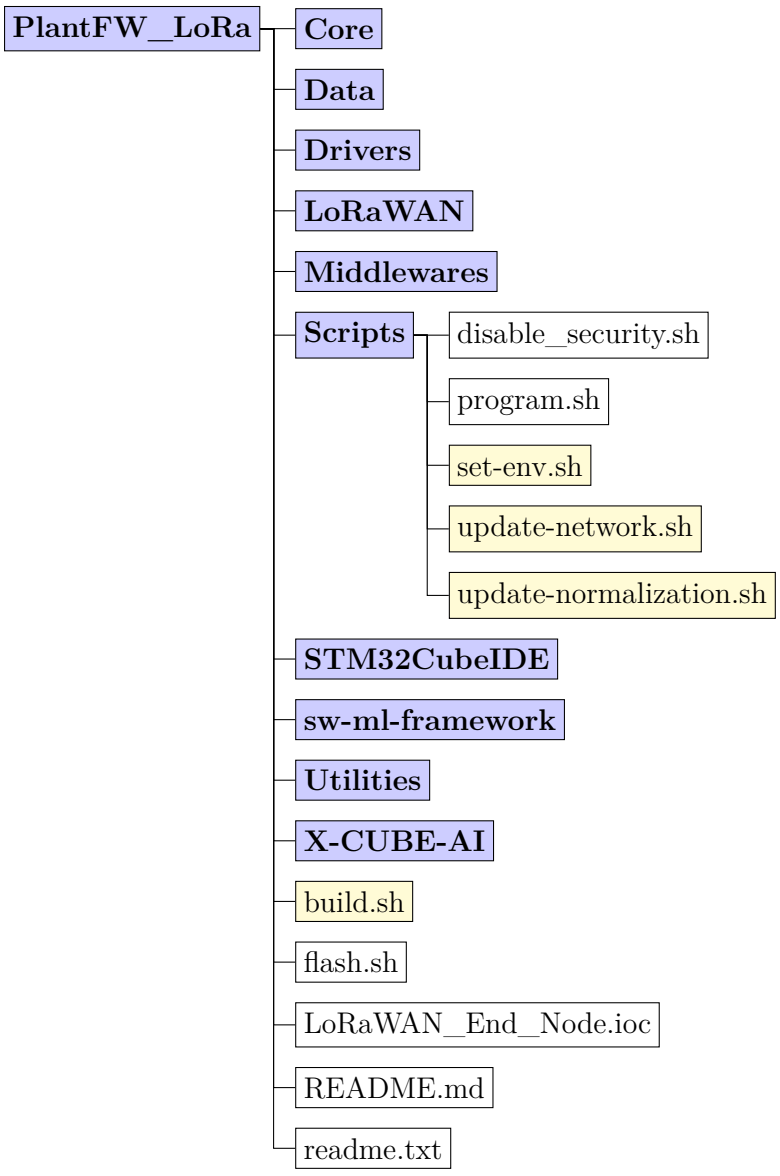


Figure 4.11: Root directories, building scripts focused

In order to simplify the needed steps, a bash script *build.sh* has been written and it is shown below in Listing A.1.

Describing the code from the top: first all default paths are set if something

needs to be customized, then the option from CLI are parsed in case all prompts need to be skipped, execution pass to following steps.

In order to get the code working, all executable paths must be set in *set-env.sh*, which is written in bash as following listed:

Listing 4.3: *set-env.sh* listing

```
1 #!/bin/bash
2 XCUBEAI_PATH=$HOME/STM32Cube/Repository/Packs/
      STMicroelectronics/X-CUBE-AI/8.0.1/Utilities/linux/stm32ai
3 STM_CUBEPROG_PATH=$HOME/Applications/stm32cubeclt_1.12.0/
      STM32CubeProgrammer/bin/STM32_Programmer_CLI
```

4.2.1 C Model Conversion

If the neural network respects the chosen metrics standards, since FW is written in C language, a conversion to that language model is needed. This process is automated by X-CUBE-AI which outputs an optimized version for STM32 MCUs, which is suitable for the case.

All input files, which are *ModelX.onnx* and *minmax.json*, must be manually moved from the *sw_ml_framework* folder to *Data/NNmodel/* folder, as highlighted in Figure 4.12. Then when script *build.sh* launches *update-network.sh*, it takes the executable path of X-CUBE-AI and set the custom variables that are by default as listed below, from listing:

Listing 4.4: *update-network.sh* settings listing

```
1 NNMODEL_PATH=../Data/NNmodel/*.onnx
2 OUTPUT_PATH=../Data/Cmodel
3 FIRMWARE_CODE_PATH=../STM32CubeIDE/Application/User/X-CUBE-AI/
      App
4 MINMAX_FILEPATH=../Data/NNmodel/minmax.json
5 NAME=plantnetwork
```

When the execution of X-CUBE-AI is finished, all output files are stored in *Data/Cmodel/* and the console shows some reports about the just created model. An example of which files are produced in *Cmodel* folder is drawn in Figure 4.13.

It is included *normalization.c* which is produced by *update-normalization.sh* which is launched after the C model is created. It is essential for the next step, since the firmware needs the new boundaries of the network to be updated.

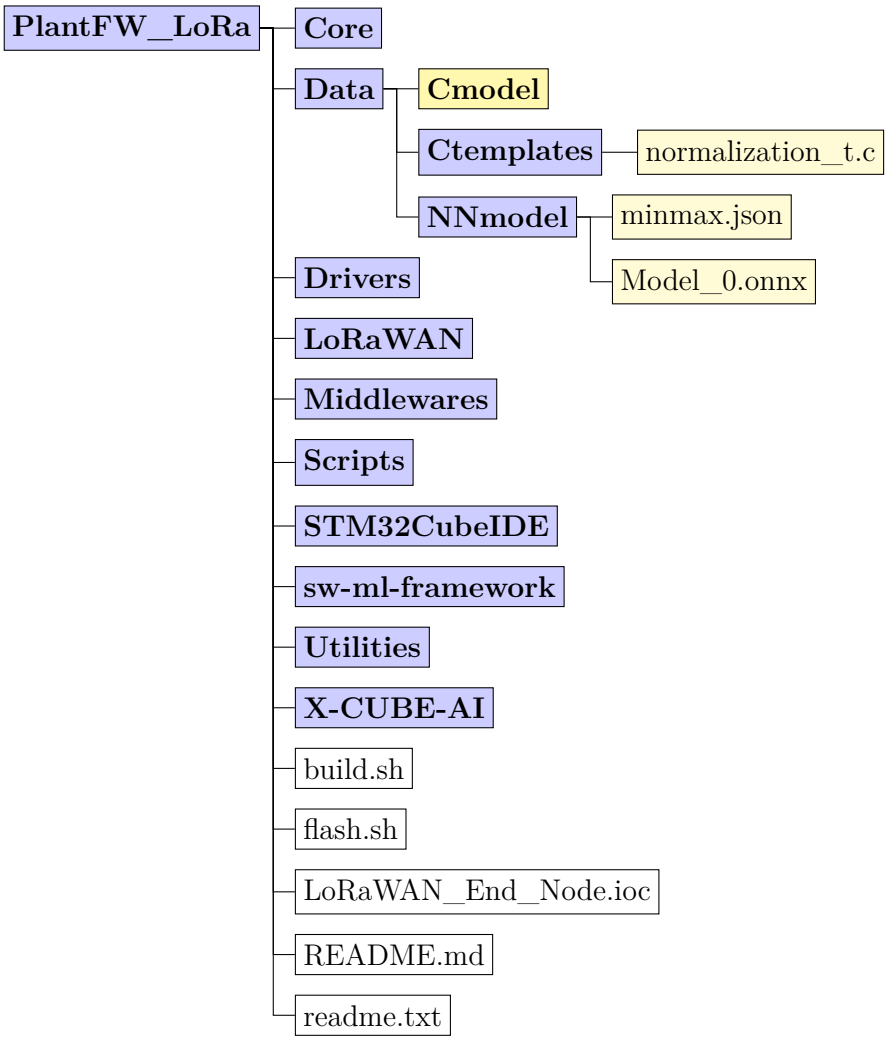


Figure 4.12: Root directories, data focused

4.2.2 Code Updating

The files contained in *Cmodel* folder needs to be substitute in target directory. In the last part of *update-nertwork.sh* the user is prompted to choose if copy the code. This function is explicited in following lines:

```
function update_code () {
    echo "Updating code on firmware..."
    mkdir -p ${FIRMWARE_CODE_PATH}
    cp -t ${FIRMWARE_CODE_PATH} ${OUTPUT_PATH}/*.[ch]
}
```

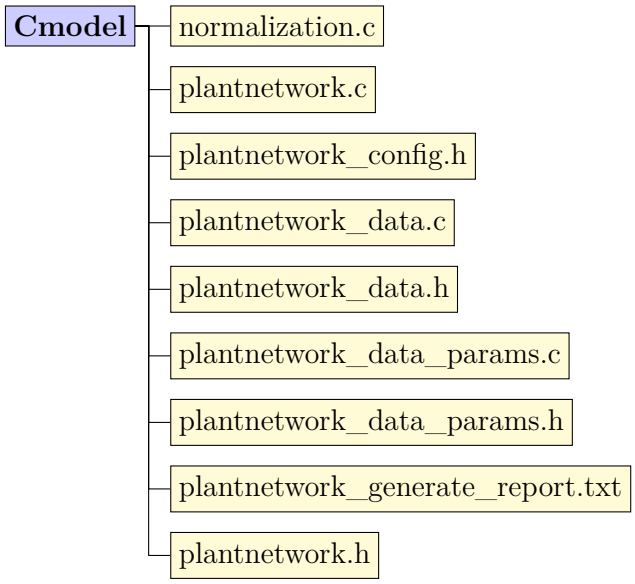


Figure 4.13: Output files inside Cmodel directory

Which moves files in the directory shown in Figure 4.14, replacing the ones highlighted in yellow.

4.2.3 Building

Once all source files are ready, *build.sh* launches

```
cd $MAKEFILE_PATH
```

```
make -j4 all
```

which compiles using the Makefile inside *STM32CubeIDE*. If there is any issue with this method, the IDE from STM can be installed to debug the code or simply regenerate Makefiles. However, this step is not required to simply reproduce the toolchain of this thesis.

4.3 Flashing

The final phase is flashing, where the built-in firmware is uploaded to the microcontroller. This step ensures that the firmware is correctly installed and

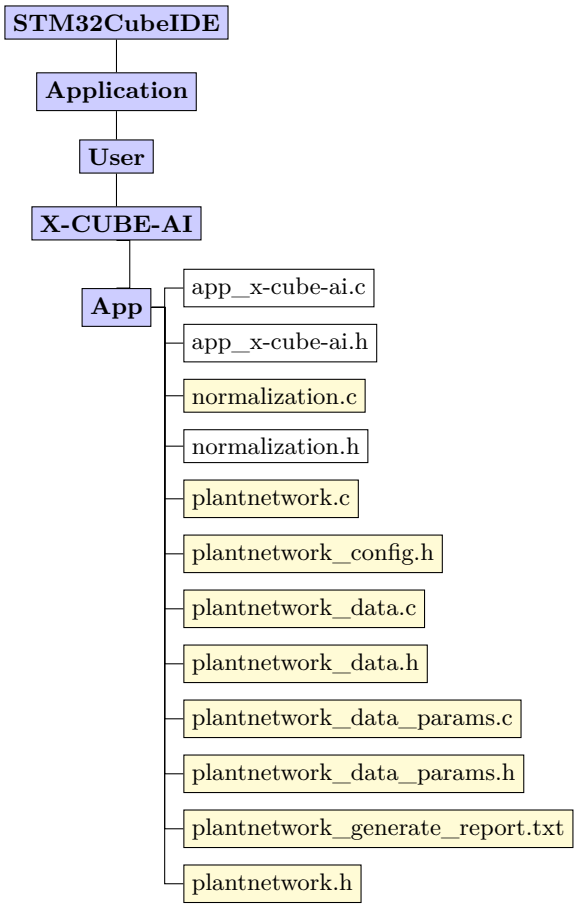


Figure 4.14: X-CUBE-AI files inside FW source code

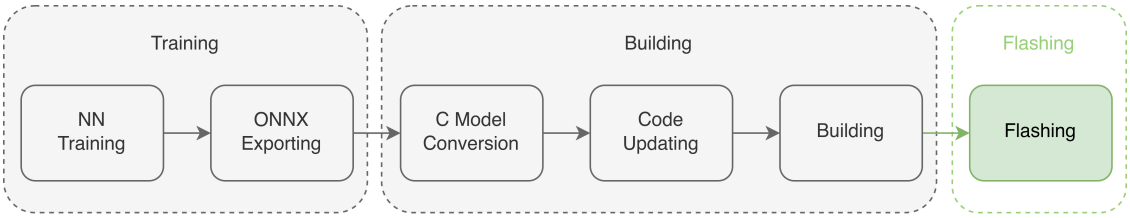


Figure 4.15: Toolchain for neural network firmware development - Flashing highlighted

ready to perform real-time plant health monitoring. The files involved in this step are highlighted in Figure 4.16.

The following sections will provide a detailed explanation of each phase, highlighting the tools and methodologies used to achieve efficient and reliable firmware development.

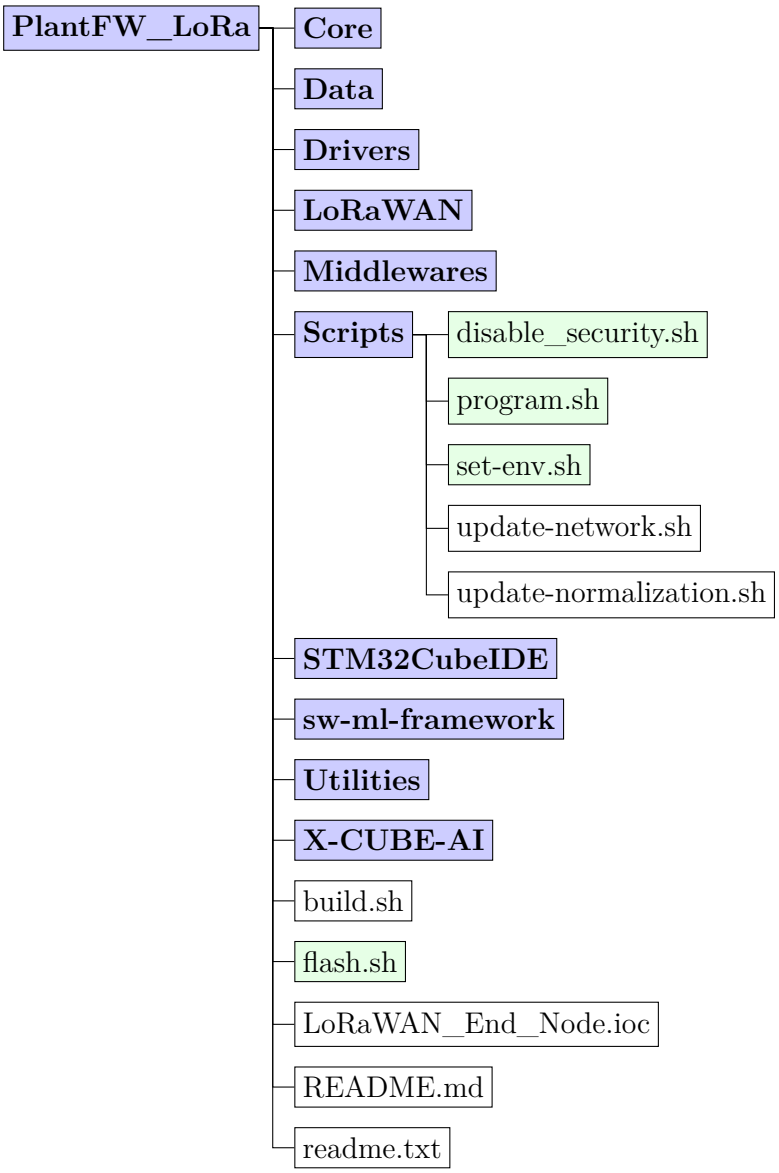


Figure 4.16: Root directories, flashing scripts focused

User can launch *flash.sh* to start the downloading process. Also in this case, user must set *set-env.sh* to make sure the executable for STMCubeProgrammer is set. The script run through two main steps:

- Security disabling
- Downloading phase

Security disabling

This process has solved a common bug found during several attempts, by the candidate and the rest of the eLiONS group, to download FW with STM32CubeIDE, which is the official development environment provided shipped with STM Nucleo boards. In particular the bug emerges in a non-deterministic way, launching multiple downloading runs. Sometimes the board behave like its locked: STLink, the default programmer on Nucleo board, can be seen but once it tries to reach the MCU the latter does not respond.

The solution was found during the development of FUOTA firmware (Section 5). A batch script was found in the folder where all building and flashing ones were contained. This batch file contained an example of the procedure developed by STM to unlock MCU. After using this file, the download process has not failed a single time.

Some improvements must be done before using it in the project; the batch file was not cross-compatible, so it has been ported in a bash script. The result is listed in Listing A.5. The exact steps are show in sequence in Figure 4.17 and they are described in following paragraphs.

Read Out Protection (RDP) The script begins by setting the Read Out Protection (RDP) level. RDP is a security feature that protects the firmware from being read or copied, thus preventing intellectual property theft and unauthorized tampering. The levels of protection are:

- **Level 1 (0xBB):** This level enables read-out protection, preventing external access to the flash memory. It ensures that the contents of the microcontroller's flash memory cannot be read or copied by unauthorized users.

Read Out Protection and Security Disable (ESE) Next, the script sets the RDP to Level 0 and disables security:

- **Level 0 (0xAA) + Security disabled (ESE 0x0):** This configuration disables read-out protection and certain security features, allowing unrestricted access to the flash memory. This setting is typically used during development when unrestricted access to the microcontroller is needed for debugging and programming.

Write Protection (WRP) The script then disables write protection for specific flash memory areas:

- **WRP disabled:** This configuration allows writing to the entire flash memory. Disabling write protection is often necessary during development or updates when the firmware needs to be modified.

User Configuration This section sets various user-specific configurations, allowing customization of the microcontroller's behavior:

- **nRST:** Configures the behavior of the microcontroller during low-power modes. Setting nRST to '1' means that no reset is generated when entering Stop, Standby, or Shutdown modes, which can be important for certain low-power applications.
- **WDG_SW:** Switches the watchdog timers to software control, allowing more flexibility in handling system resets and monitoring system health.
- **IWDG:** Ensures the independent watchdog counter is frozen in Stop/Standby modes, which helps to prevent unwanted resets during low-power operations.
- **BOOT:** Disables boot lock for CPU1 and CPU2, allowing for more flexible boot configurations, which is useful during development and debugging.

Security Configuration This section configures additional security settings to further protect the microcontroller and its data:

- **HDPAD:** Disables the hide protection area for user flash, ensuring that all parts of the user flash are accessible.
- **SPISD:** Disables SPI3 security, allowing SPI3 to operate without additional security restrictions, which may be necessary for certain communication applications.
- **SBRSA:** Resets the default value of the SRAM start address to a secure setting, ensuring that sensitive data stored in SRAM is protected.
- **SBRV:** Resets the default value of the CPU2 boot start address, ensuring secure booting of the secondary CPU.

Downloading phase

In the previous section, the steps involved in the disabling of security on the STM32 microcontroller have been detailed, ensuring that the device is ready for subsequent operations. Disabling security is a crucial step that allows us to perform actions such as memory erasure and binary downloads without

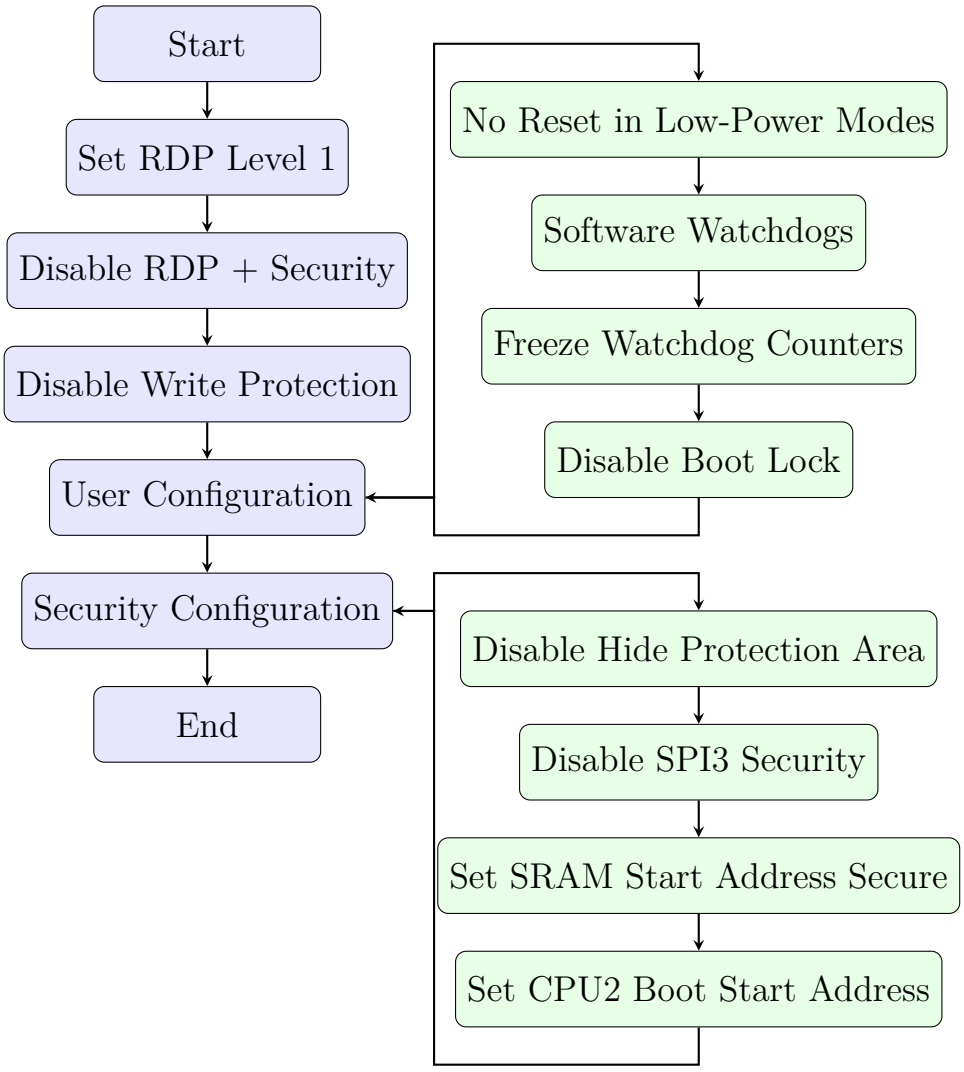


Figure 4.17: ASM of disable-security

encountering security restrictions previously descripted that could hinder these processes. Having successfully completed the security disabling phase, we now move on to the next critical phase: the Download Phase.

The Download Phase encompasses the process of erasing the existing memory content on the microcontroller and subsequently downloading the new binary files. This phase is essential for updating the firmware or deploying new applications on the microcontroller. The operations in this phase ensure that the microcontroller has a clean memory slate and the latest firmware version, which is crucial to maintaining the functionality and security of the system. The following steps are synthetized from Listing A.6.

Erase Memory The next step is to erase the entire microcontroller memory. This is done using the STM32CubeProgrammer tool, specified by the `$STM_CUBEPROG_PATH` variable, with the `-c port=SWD mode=UR -e all` options to connect via SWD and erase all memory.

The script checks the exit status of the memory erasure command. If the command fails, the script terminates with an error message.

Download Binaries This step involves downloading the binary files to the microcontroller. A function named `download_file` is defined to handle this process. The function checks for the existence of the binary file and then uses the STM32CubeProgrammer to download the binary.

The function first checks if the binary file exists. If not, it exits with an error message. It then downloads the binary using the STM32CubeProgrammer, verifies the download, and performs a hard reset on the microcontroller. If any of these steps fail, the function exits with an error message.

Power Cycle After successfully downloading the binaries, the script prompts the user to power cycle the board to apply the BFU security mechanisms.

Conclusion

This phase, coupled with the preceding security disabling phase, forms a comprehensive approach to preparing the microcontroller for its intended applications. The flashing step is the most crucial of the toolchain, since as proved by practical attempt, there is not effective way to know the exact size of the binary and sometimes it can overflow the size of flash memory dedicated to code.

Chapter 5

Challenges in Implementing FUOTA

FUOTA (firmware update over-the-air) is a features that permits to remotely update the firmware online, while running. The implementation of FUOTA ensures that the devices remain up-to-date with the latest enhancements and security patches, thereby extending their operational lifespan and reducing maintenance costs. Since a flexible toolchain was developed to update the firmware, this also consents to improve NN performance by launching new trainings with fresh values, an attempt was made to port the code.

5.1 Introduction to FUOTA for WL55JC1 Series

According to STMicroelectronics, the STM32WL series is designed to support over-the-air firmware updates, ensuring that devices can be updated with minimal disruption to their operation [16].

5.1.1 Memory Mapping

Understanding the memory mapping of the STM32WL55JC1 series is essential for implementing FUOTA. The most important information to be deducted is the download memory size, since this is limiting for the firmware. In Figure 5.1 there is a detailed description of the memory sectors.

Start address	End address	Flash memory region
0x0800 0000	0x0800 67FF	Secure Boot CM4
0x0800 6800	0x0800 6FFF	Download slot (2 Kbytes) for KMS blob
0x0800 7000	0x0800 71FF	Reserved
0x0800 7200	0x0801 3FFF	Active image #2 End_Node_DualCore_CM4 (52 Kbytes)
0x0801 4000	0x0801 4FFF	LoRaWAN NVM
0x0801 5000	0x0801 51FF	Reserved
0x0801 5200	0x0802 97FF	Active image #1 End_Node_DualCore_CM0+ (82 Kbytes)
0x0802 9800	0x0802 B7FF	KMS Data Storage (8 Kbytes)
0x0802 B800	0x0802 CBFF	SE interface Cortex-M0+
0x0802 CC00	0x0803 5FFF	SBSFU Cortex-M0+
0x0803 6000	0x0803 61FF	SBSFU CM0+ vector table
0x0803 6200	0x0803 E4FF	SE Cortex-M0+
0x0803 E500	0x0803 E7FF	User keys
0x0803 E800	0x0803 EFFF	SE keys
0x0803 F000	0x0803 F7FF	Active image #2 header
0x0803 F800	0x0803 FFFF	Active image #1 header

Start address	End address	External flash memory region
0x9000 0000	0x9000 01FF	Download image header
0x9000 0200	0x9001 4FFF	Download image (84 Kbytes)

Start address	End address	RAM region
0x2000 0000	0x2000 0CDF	Secure Boot CM4
0x2000 0CE0	0x2000 0CFF	Cortex-M0+/M4 sync flag
0x2000 0D00	0x2000 6FFF	End_Node_DualCore_CM4
0x2000 7000	0x2000 73FF	Mapping table Mailbox MEM1 Cortex-M4
0x2000 7400	0x2000 7FFF	Mailbox MEM2 Cortex-M0+
0x2000 8000	0x2000 C7FF	SBSFU Cortex-M0+ End_Node_DualCore_CM0+
0x2000 C800	0x2000 FFEF	SE Cortex-M0+
0x2000 FFF0	0x2000 FFFF	KMS DataStorage key (encrypt/decrypt blob)

DT71516V1

Figure 5.1: Memory Mapping for `LoRaWAN_FUOTA_DualCore_ExtFlash` [31]

5.1.2 Example Calculation: Transmission Time

To estimate the transmission time for a firmware update, a scenario with a spread factor (SF) of 7, a bandwidth of 125 kHz and a firmware size of 84kB is considered. The spread factor affects the data rate and the air time for each transmission.

The data rate for SF7 in LoRaWAN typically corresponds to a payload of 235 bytes per packet. Given the overhead for each packet estimated for 13, we can assume an effective payload of approximately 222 bytes.

$$\text{Total packets required} = \left\lceil \frac{84 \times 1024\,\text{B}}{222\,\text{B/packet}} \right\rceil = \lceil 387{,}459459459 \rceil = 388 \text{ packets} \tag{5.1}$$

Since it has been used the maximum bytes per hour, now the only constraints is the duty cycle. In the European band 863 MHz limits the transmission period to 1%. Considering ToA (Time over Air), calculated by `Airtime calculator for LoRaWAN`, of 368.9 ms, the next transmitted packet should wait:

$$\text{Wait time} = \lceil 368.9\,\text{ms}/1\% \rceil = 36.9\,\text{s} \tag{5.2}$$

$$\text{Total transmission time} = 388 \text{ packets} \times 36.9\,\text{s/packet} = 14{,}317.2\,\text{s} \approx 4\,\text{h} \tag{5.3}$$

Therefore, the estimated transmission time for an 84kB firmware update with a spread factor of 7 is approximately 4 hours. This calculation demonstrates the importance of optimizing firmware size and selecting appropriate transmission parameters to minimize update time.

5.2 State of Development

FUOTA End Node sits in the folder *Examples* of SDK. Since it is an empty example, the code must be re-adapted.

The build step is processed through a script which is programmed in Batch language, which is compatible only for Windows systems. A Bash port has been realised in order to run the build with the Unix systems. The project generates four key binaries:

1. **2_Images_KMS_Blob:** This binary is built to include the Key Management Services (KMS) functionalities.It ensures secure handling and storage of cryptographic keys used during the firmware update process.
2. **2_Images_SECoreBin:** The SE core binary is compiled to provide secure boot and secure firmware update capabilities. It ensures that only authenticated firmware updates are applied to the device.
3. **2_Images_SBSFU:** The SBSFU binary manages the secure boot process and the firmware update operations. It coordinates the download and verification of new firmware images.
4. **LoRaWAN_End_Node_DualCore:** This binary contains the main application, including the LoRaWAN protocol stack and the application logic. It handles communication with the network and the execution of the device's primary functions.

Each binary has a postbuild script that combines the previous blob with the current. The final flash script has to upload only the last one produced.

5.2.1 Dependencies

Some pre-built executables are provided with the repository, but these are compatible only with Windows. However, some research in the `postbuild.sh` and `prebuild.sh` scripts contained in some project folders reveals some references to a Python script that performs the same processes. The latter is located in the path *Lib/Middlewares/ST/STM32_Secure_Engine/Utilities/KeysAndImages/*. Some Python libraries need to be added to get the script `keys.py` working, they are reported below:

- **PyCryptodome** contains some common cryptography algorithms
- **PyElftool** has tools to manipulate .elf binary files.
- **ecdsa** contains some tools to use ECC (Elliptic Curve Cryptography).

5.2.2 Oversize Issues

Once all the firmware has been ported, *build.sh* has been launched. The building process failed with the following log message:

Listing 5.1: Result log from *build.sh*

```
20:01:47 **** Build of configuration Debug for project 2
    _Images_SBSFU_CM0PLUS ****
make -j12 all
arm-none-eabi-gcc -o "SBSFU.elf" @"objects.list"   -mcpu=cortex-m0plus -T"/
    home/repex/Scrivania/PlantFW_FUOTA_DualCore_ExtFlash/2_Images_SBSFU/
    STM32CubeIDE/CM0PLUS/SBSFU_cm0plus.ld" --specs=nosys.specs -Wl,-Map="
    SBSFU.map" -Wl,--gc-sections -static -L ../../../../Linker_Common/
    STM32CubeIDE --specs=nano.specs -mfloat-abi=soft -mthumb -Wl,--start-
    group -lc -lm -Wl,--end-group
/opt/st/stm32cubeide_1.13.2/plugins/.../arm-none-eabi/bin/ld: SBSFU.elf
    section '.SE_IF_Code' will not fit in region 'SE_IF_ROM_region'
/opt/st/stm32cubeide_1.13.2/plugins/.../arm-none-eabi/bin/ld: section .text
    LMA [000000000802d000,0000000008033ecf] overlaps section .SE_IF_Code LMA
     [000000000802b800,000000000802d027]
/opt/st/stm32cubeide_1.13.2/plugins/.../arm-none-eabi/bin/ld: region '
    SE_IF_ROM_region' overflowed by 40 bytes
collect2: error: ld returned 1 exit status
make[1]: *** [makefile:72: SBSFU.elf] Error 1
make: *** [makefile:65: all] Error 2
"make -j12 all" terminated with exit code 2. Build might be incomplete.

20:01:48 Build Failed. 5 errors, 0 warnings. (took 262ms)
```

That indicates an oversize problem with the declared configuration. The region named *SE Interface Cortex-M0+* should begin at the address `0x0802 B800` and end at `0x0802 CBFF`, but it results in an overlap with the next region up to `0x0802 D027`.

The issue has been fixed, giving more room in memory mapping to the incriminated region. The current boundaries are set now to `0x0802 B800` -> `0x0802 D7FF`.

Then the execution could proceed over this stage, but it has got stuck in this new error:

Listing 5.2: Result log from *build.sh*

```
1 /opt/st/stm32cubeide_1.13.2/plugins/com.st.stm32cube.ide.mcu.externaltools.
    gnu-tools-for-stm32.11.3.rel1.linux64_1.1.1.202309131626/tools/bin/../
    lib/gcc/arm-none-eabi/11.3.1/../../../../arm-none-eabi/bin/ld: SB.elf
    section `._user_heap_stack' will not fit in region `M4_SB_RAM_region'
2 /opt/st/stm32cubeide_1.13.2/plugins/com.st.stm32cube.ide.mcu.externaltools.
    gnu-tools-for-stm32.11.3.rel1.linux64_1.1.1.202309131626/tools/bin/../
    lib/gcc/arm-none-eabi/11.3.1/../../../../arm-none-eabi/bin/ld: region `
    M4_SB_RAM_region' overflowed by 240 bytes
3 collect2: error: ld returned 1 exit status
4 make: *** [makefile:67: SB.elf] Error 1
5 "make -j12 all" terminated with exit code 2. Build might be incomplete.
6
7 14:21:01 Build Failed. 3 errors, 0 warnings. (took 1s.435ms)
```

Which indicates another overlap in assigned RAM region. The error indicates an insufficiency of space to run the Cortex M4 Secure Boot execution. Another time is has been solved by extending the region, shrinking the User Application RAM section. The results is reported in Table 5.1:

Name	Start	End
Cortex M4 SB	0x2000 0000	0x2000 181F
Cortex M0+/M4 sync flag	0x2000 1820	0x2000 183F
Cortex M4 User Space	0x2000 1840	0x2000 6FFE

Table 5.1: Modified Memory Map

This modification brings up to the last part of the current development state. The entire project folder is a standalone project. It can be built or flashed using the scripts or the IDE. Nevertheless, additional efforts remain requisite to fully get operational the X-CUBE-AI functionalities.

Chapter 6

Conclusion and Future Perspective

The development and deployment of AI firmware for remote wireless plant health monitoring, as presented in this thesis, represent a significant advancement in agricultural technology. The integration of neural networks into plant health monitoring systems offers a robust, efficient, and scalable solution to the challenges faced by traditional methods. By leveraging the intrinsic characteristics of plants through impedance analysis, the developed firmware provides accurate assessments of plant health, which is crucial for ensuring agricultural productivity and sustainability.

The implementation process involved a thorough examination of neural network architectures to identify the most suitable design for plant health monitoring. The integration of the LoRaWAN protocol ensured low power consumption and extensive range, making the system ideal for large-scale agricultural applications.

Key contributions of this research include:

- **Neural Network Integration:** The integration of neural networks into the firmware allowed for the analysis of complex plant health data, enabling precise and real-time monitoring.
- **Efficient Data Collection:** The developed system effectively collected and processed data from various sensors, providing valuable insights into plant health.

- **Low Power Consumption:** The use of the LoRaWAN protocol ensured that the system operated with minimal power consumption, which is critical for battery-operated devices in remote locations.
- **Scalability:** The system's design allows for scalability, making it suitable for deployment in large agricultural fields and various environmental conditions.

The results demonstrated that the AI firmware could accurately monitor plant health by analyzing impedance and soil water potential (SWP). The system's ability to predict plant health conditions based on these parameters highlights its potential for widespread application in agriculture, horticulture, and environmental science.

6.1 Future Perspectives

The research presented in this thesis opens several avenues for future work:

- **Enhanced Neural Network Models:** Future work could explore the use of more advanced neural network models and machine learning techniques to improve the accuracy and reliability of plant health assessments.
- **Integration with IoT Platforms:** Integrating the developed system with existing IoT platforms and cloud services can enhance data analysis, visualization, and decision-making processes.
- **Field Trials:** Conducting extensive field trials in diverse agricultural settings will help validate the system's performance and identify areas for improvement.
- **Multi-Parameter Analysis:** Expanding the system to monitor additional parameters such as soil nutrients and others can provide a more comprehensive assessment of plant health.
- **Energy Harvesting:** Investigating energy harvesting techniques, such as solar or wind power, can further enhance the system's sustainability by reducing dependence on batteries.

In conclusion, this thesis demonstrates the feasibility and effectiveness of using AI firmware for remote wireless plant health monitoring. The integration of neural networks and low-power communication protocols has the potential to revolutionize agricultural practices, ensuring food security and promoting sustainable farming. Continued research and development in this field will contribute to the advancement of smart agriculture and environmental monitoring technologies.

Appendix A

Bash scripts listings

A.1 Building Scripts

A.1.1 C Model Conversion

Listing A.1: *build.sh*

```
#!/bin/bash
MAKEFILE_PATH="STM32CubeIDE/Release/"
UPDATE_SCRIPTS_PATH="Scripts/"
UPDATE_SCRIPTS_FILENAME="./update-network.sh"
CURRENT_DIR=$PWD

function help_func () {
    cat << EOF
Usage: ./build.sh [-y|-h]

Description:
    Simple script to update c model of neural network and/or
   build the firmware

Options:
    -y          Set all answer to yes
    -h|--help   Show this message
EOF
}

# Parse command line arguments
bypass=""
```

```
22 while [[ $# -gt 0 ]]; do
23     case "$1" in
24         -y)
25             bypass=Y
26             ;;
27         -h)
28             help_func
29             exit 1
30             ;;
31         --help)
32             help_func
33             exit 1
34             ;;
35         *)
36             echo "Unknown option: $1"
37             help_func
38             exit 1
39             ;;
40     esac
41     shift
42 done
43
44 # 1. Update CModel
45 answer=$bypass
46 while true; do
47     if [[ $answer == "" ]]; then
48         read -r -p "Do you wish to update the code on firmware?
       (Y/N): " answer
49     fi
50     case $answer in
51         [Yy]* )
52             echo "Updating C model of neural network..."
53             cd $UPDATE_SCRIPTS_PATH
54             $UPDATE_SCRIPTS_FILENAME -y
55             cd $CURRENT_DIR
56             break;;
57         [Nn]* )
58             break;;
59         * )
60             answer=""
61             echo "Please answer Y or N.";;
62     esac
63 done
64
65 # 2. Clean prompt
```

```
answer=$bypass
cd $MAKEFILE_PATH
while true; do
    if [[ $answer == "" ]]; then
        read -r -p "Do you wish to clean the project folder first? (Y/N): " answer
    fi
    case $answer in
        [Yy]* )
            echo "Launching make clean..."
            make -j4 clean
            break;;
        [Nn]* )
            break;;
        * )
            answer=""
            echo "Please answer Y or N.";;
    esac
done

# 3. Build FW
echo "Building the firmware"
make -j4 all
cd $CURRENT_DIR
```

Listing A.2: *update-network.sh*

```
#!/bin/bash
source ./set-env.sh
NNMODEL_PATH=../Data/NNmodel/*.onnx
OUTPUT_PATH=../Data/Cmodel
FIRMWARE_CODE_PATH=../STM32CubeIDE/Application/User/X-CUBE-AI/App
MINMAX_FILEPATH=../Data/NNmodel/minmax.json
NAME=plantnetwork

answer=""
while [[ $# -gt 0 ]]; do
    case "$1" in
        -y)
            answer=Y
            ;;
        *)
            echo "Unknown option: $1"
            exit 1
```

```
18              ;;
19      esac
20      shift
21  done
22
23  function update_code () {
24      echo "Updating code on firmware..."
25      mkdir -p ${FIRMWARE_CODE_PATH}
26      cp -t ${FIRMWARE_CODE_PATH} ${OUTPUT_PATH}/*.[ch]
27  }
28
29  ${XCUBEAI_PATH} generate --type onnx -n $NAME -m ${NNMODEL_PATH
        } -o ${OUTPUT_PATH}
30
31  if [[ -f $MINMAX_FILEPATH ]]; then
32      ./update-normalization.sh
33  fi
34
35  while true; do
36      if [[ $answer == "" ]]; then
37          read -r -p "Do you wish to update the code on firmware?
        (Y/N): " answer
38      fi
39      case $answer in
40          [Yy]* )
41              update_code
42              break;;
43          [Nn]* )
44              exit;;
45          * )
46              answer=""
47              echo "Please answer Y or N.";;
48      esac
49  done
```

Listing A.3: *update-normalization.sh*

```
1  #!/bin/bash
2  MINMAX_FILEPATH="../Data/NNmodel/minmax.json"
3  NORMALIZATION_T_FILEPATH="../Data/Ctemplates/normalization_t.c"
4  NORMALIZATION_OUTPUT="../Data/Cmodel/normalization.c"
5
6  # Parse JSON file and extract values
7  mapfile -t max_values < <(jq -r '.[].[].max' $MINMAX_FILEPATH)
8  mapfile -t min_values < <(jq -r '.[].[].min' $MINMAX_FILEPATH)
```

```
mapfile -t features   < <(jq -r '.[] | keys[]' $MINMAX_FILEPATH
    )

# Create the replacement text
echo -e "Normalization set points:\n\n Features    | Min
    | Max         \n-----------------------------------------"
for ((i=0; i<${#max_values[@]}; i++)); do
    if [ $i -eq 0 ]; then
        text="    {${max_values[i]}, ${min_values[i]}}"
    else
        text+=",\n    {${max_values[i]}, ${min_values[i]}}"
    fi
    printf " %-10s | %-10s | %-10s \n" ${features[i]} ${
    min_values[i]} ${max_values[i]}
done

# Create a new file with substituted values
cp $NORMALIZATION_T_FILEPATH $NORMALIZATION_OUTPUT

# Replace the placeholder with the replacement text
sed -i "s|\s*\/\/INSERT HERE\s*|$text|" $NORMALIZATION_OUTPUT

echo -e "\nUpdated in $NORMALIZATION_OUTPUT"
```

A.1.2 Flashing Scripts

Listing A.4: *flash.sh*

```
#!/bin/bash
CURRENT_DIR=$PWD

cd Scripts/
./program.sh
cd $CURRENT_DIR
```

Listing A.5: *disable-security.sh*

```
#!/bin/bash

current_dir="$(pwd)/"

echo "################################################"
echo "# 0- Set all global variables"
echo "################################################"
source ./set-env.sh
```

```
9
10 echo "#############################################"
11 echo "# 1- Disable security"
12 echo "#############################################"
13
14 function write_ob () {
15     len_ob=0
16     option_byte_name=()
17     option_byte_val=()
18     while [[ $# -gt 1 ]]; do
19         echo $1 $2
20         if [[ ! ( -z "$1" || -z "$2" ) ]]
21         then
22             option_byte_name+=($1)
23             option_byte_val+=($2)
24             len_ob=$(($len_ob+1))
25         fi
26         shift 2
27     done
28     # Check if the number of arguments is odd
29     if [[ $# -eq 1 ]]; then
30         echo "The last argument ($1) is not paired and will be
    ignored."
31     fi
32     len_ob=$(($len_ob-1))
33     obj_str=""
34     for i in $(seq 0 $len_ob)
35     do
36         obj_str="${option_byte_name[$i]}=${option_byte_val[$i]}
    $obj_str"
37     done
38     echo $obj_str
39     #Writing byte
40     $STM_CUBEPROG_PATH -c port=SWD mode=UR -q -ob "$obj_str" >
    /dev/null 2>&1
41     if [[ ${?} -ne 0 ]]; then
42         echo "Error: Command write -ob $obj_str Failed"
43         exit 1
44     fi
45     #Checking byte
46     $STM_CUBEPROG_PATH -c port=SWD mode=UR -q -ob displ > temp.
    txt
47     for i in $(seq 0 $len_ob)
48     do
```

```
49          for read_ob in $(grep -oP "\s${option_byte_name[$i]}\s
    *:\s*0x[0-9A-Fa-f]+" temp.txt | awk '{print $3}')
50          do
51              if [[ $read_ob -ne ${option_byte_val[$i]} ]]
52              then
53                  echo -e "Error: Option Byte ${option_byte_name[
    $i]} not modified as expected: read:$read_ob expected:${
    option_byte_val[$i]}"
54                  rm -f temp.txt  > /dev/null 2>&1
55                  exit 1
56              fi
57          done
58      done
59
60      rm -f temp.txt  > /dev/null 2>&1
61  }
62
63  echo "RDP: Read Out protection Level 1"
64  write_ob RDP 0xBB
65
66  echo "RDP+ESE: Read Out protection Level 0 + Security disabled"
67  write_ob RDP 0xAA ESE 0x0
68
69  echo "WRP: Write Protection disabled"
70  write_ob WRP1A_STRT 0x7F WRP1A_END 0x0 WRP1B_STRT 0x7F
    WRP1B_END 0x0
71
72  echo "------- User Configuration -------"
73  echo "nRST: No reset generated when entering the Stop/Standby/
    Shutdown modes"
74  write_ob nRST_STOP 0x1 nRST_STDBY 0x1 nRST_SHDW 0x1
75
76  echo "WDG_SW: Software window/independent watchdogs"
77  write_ob WWDG_SW 0x1 IWDG_SW 0x1
78
79  echo "IWDG: Independent watchdog counter frozen in Stop/Standby
      modes"
80  write_ob IWGD_STDBY 0x0 IWDG_STOP 0x0
81
82  echo "BOOT: CPU1+CPU2 CM0+ Boot lock disabled"
83  write_ob BOOT_LOCK 0x0 C2BOOT_LOCK 0x0
84
85  echo "------- Security Configuration -------"
86  echo "HDPAD: User Flash hide protection area access disabled"
87  write_ob HDPAD 0x1
```

```
88
89 echo "SPISD: SPI3 security disabled"
90 write_ob SUBGHSPISD 0x1
91
92 echo "SBRSA: Reset default value of SRAM Start address secure"
93 write_ob SNBRSA 0x1F SBRSA 0x1F
94
95 echo "SBRV: Reset default value of CPU2 Boot start address"
96 write_ob SBRV 0x8000
```

Listing A.6: *program.sh* listing

```
1 #!/bin/bash
2
3 current_dir="$(pwd)/"
4 binary_file="$current_dir../STM32CubeIDE/Release/*.elf"
5
6 echo "################################################"
7 echo "# 0- Set all global variables"
8 echo "################################################"
9 source ./set-env.sh
10
11 echo "################################################"
12 echo "# 1- Disable security"
13 echo "################################################"
14 source ./disable_security.sh
15 if [[ $? -ne 0 ]]; then
16     echo "Error disabling security"
17     exit 1
18 fi
19
20 echo "################################################"
21 echo "# 2- Erase Memory"
22 echo "################################################"
23 $STM_CUBEPROG_PATH -c port=SWD mode=UR -e all
24 if [[ $? -ne 0 ]]; then
25     echo "Error: Full Memory Erase Failure"
26     exit 1
27 fi
28
29 echo "################################################"
30 echo "# 3- Download binaries"
31 echo "################################################"
32
33 function download_file () {
```

```
34      if [[ ! -f $1 ]]; then
35          echo "Error: $1 File not found. Check your build log...
    "
36          exit 1
37      fi
38
39      if [[ $3 = "" ]]
40      then
41          echo "Downloading $2 binary ..."
42          $STM_CUBEPROG_PATH -c port=SWD mode=UR -d $1 -v
43      else
44          echo "Downloading $2 binary @$3 ..."
45          $STM_CUBEPROG_PATH -c port=SWD mode=UR -d $1 $3 -v
46      fi
47
48      if [[ $? -ne 0 ]]; then
49          echo "Error: Download failed"
50          exit 1
51      fi
52
53      $STM_CUBEPROG_PATH -c port=SWD mode=HOTPLUG -hardRst
54      if [[ $? -ne 0 ]]; then
55          echo "Error: Reset after download failed"
56          exit 1
57      fi
58
59      echo "Done!"
60  }
61
62  download_file $binary_file $(basename $binary_file)
63
64  echo "Power cycle the board (unplug/plug USB cable) to apply
    the BFU security mechanisms..."
65  exit 0
```

Bibliography

[1] Food and Agriculture Organization of the United Nations. *The future of food and agriculture: Trends and challenges*. FAO, 2017 (cit. on pp. 1, 2).

[2] Mingqiu Zhang, Botong Li, and Yang Heng. «Plant phenotyping: past, present, and future». In: *The Crop Journal* 8.3 (2020), pp. 329–342 (cit. on p. 2).

[3] Paul Mooney and Kevin Peairs. «Smart agricultural monitoring based on solar-powered wireless sensor network». In: *IEEE Sensors Journal* 10.10 (2010), pp. 1571–1583 (cit. on p. 3).

[4] Aloïs Augustin, Jianmin Yi, Thomas Hee Clausen, and William Townsley. «A study of LoRa: Long range & low power networks for the internet of things». In: *Sensors* 16.9 (2016), p. 1466 (cit. on pp. 3, 19).

[5] eLiONS Research Group. *eLiONS - Plant Health Monitoring Project*. `https://elions.polito.it/home/agrifood-electronics/plant-health-monitoring/`. Accessed: 2024-06-23. 2019 (cit. on p. 3).

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016 (cit. on p. 5).

[7] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009 (cit. on p. 6).

[8] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. «Deep Learning». In: *Nature* 521.7553 (2015), pp. 436–444 (cit. on p. 6).

[9] Christopher M Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006 (cit. on p. 6).

[10] Vinod Nair and Geoffrey E Hinton. «Rectified Linear Units Improve Restricted Boltzmann Machines». In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 2010, pp. 807–814 (cit. on p. 7).

[11] Calvo Stefano, Barezzi Mattia, Demarchi Danilo, and Garlando Umberto. «In-vivo proximal monitoring system for plant water stress and biological activity based on stem electrical impedance». In: *9th International Workshop on Advances in Sensors and Interfaces* (2023), pp. 80–85. URL: https://hdl.handle.net/11583/2980510 (cit. on p. 8).

[12] Guojie Hu et al. «Water and heat coupling processes and its simulation in frozen soils: Current status and future research directions». In: *CATENA* 222 (2023), p. 106844. ISSN: 0341-8162. DOI: https://doi.org/10.1016/j.catena.2022.106844. URL: https://www.sciencedirect.com/science/article/pii/S034181622200830X (cit. on p. 8).

[13] Ian C. Dodd, Andrew D. Hirons, and Jaime Puértolas. «Plant-water relations». In: *Encyclopedia of Soils in the Environment (Second Edition)*. Ed. by Michael J. Goss and Margaret Oliver. Second Edition. Oxford: Academic Press, 2023, pp. 516–526. ISBN: 978-0-323-95133-3. DOI: https://doi.org/10.1016/B978-0-12-822974-3.00253-6. URL: https://www.sciencedirect.com/science/article/pii/B9780128229743002536 (cit. on p. 8).

[14] Umberto Garlando, Stefano Calvo, Mattia Barezzi, Alessandro Sanginario, Paolo Motto Ros, and Danilo Demarchi. «Ask the plants directly: Understanding plant needs using electrical impedance measurements». In: *Computers and Electronics in Agriculture* 193 (2022), p. 106707. ISSN: 0168-1699. DOI: https://doi.org/10.1016/j.compag.2022.106707. URL: https://www.sciencedirect.com/science/article/pii/S0168169922000242 (cit. on p. 9).

[15] STMicroelectronics. *STM32WL55JC Datasheet*. Accessed: 2024-07-06. Dec. 2022. URL: https://www.st.com/resource/en/datasheet/stm32wl55jc.pdf (cit. on p. 12).

[16] STMicroelectronics. *STM32WL55JC Reference Manual*. Accessed: 2024-07-06. Jan. 2023. URL: https://www.st.com/resource/en/reference_manual/rm0453-stm32wl5x-advanced-armbased-32bit-mcus-with-subghz-radio-solution-stmicroelectronics.pdf (cit. on pp. 13, 69).

[17] STMicroelectronics. *X-CUBE-AI: Expand Your STM32 Project with Artificial Intelligence*. Accessed: 2024-06-23. Sept. 2020. URL: https://www.st.com/en/embedded-software/x-cube-ai.html (cit. on pp. 15–17).

[18] STMicroelectronics. «Bringing AI to the Edge with STM32: The X-CUBE-AI Expansion Package». In: (June 2020). Accessed: 2024-06-23. URL: https://www.st.com/content/dam/public/xtnumber/AI/X-CUBE-AI.pdf (cit. on p. 16).

[19] STMicroelectronics. *X-CUBE-AI User Manual*. Accessed: 2024-06-23. Sept. 2020. URL: https://www.st.com/resource/en/user_manual/um2526-xcubeai-ai-expansion-pack-for-stm32cube-stmicroelectronics.pdf (cit. on pp. 16, 17).

[20] Luis Guerra and Tom Drummond. «Automatic Pruning for Quantized Neural Networks». In: *2021 Digital Image Computing: Techniques and Applications (DICTA)*. 2021, pp. 01–08. DOI: 10.1109/DICTA52665.2021.9647074 (cit. on p. 16).

[21] STMicroelectronics. *STM32CubeMX: Initialization Code Generator for STM32 MCUs*. Accessed: 2024-06-23. June 2020. URL: https://www.st.com/en/development-tools/stm32cubemx.html (cit. on pp. 16, 17).

[22] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. *Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation*. 2020. arXiv: 2004.09602 [cs.LG]. URL: https://arxiv.org/abs/2004.09602 (cit. on p. 17).

[23] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. «What is the State of Neural Network Pruning?» In: *Proceedings of Machine Learning and Systems*. Ed. by I. Dhillon, D. Papailiopoulos, and V. Sze. Vol. 2. 2020, pp. 129–146. URL: https://proceedings.mlsys.org/paper_files/paper/2020/

file/6c44dc73014d66ba49b28d483a8f8b0d-Paper.pdf (cit. on p. 17).

[24] Yegin Sornin. «LoRaWAN specification». In: *LoRa Alliance* 1.1 (2017), pp. 1–101 (cit. on p. 19).

[25] Ferran Adelantado, Xavier Vilajosana, Pere Tuset-Peiro, Borja Martinez, Joan Melia-Segui, and Thomas Watteyne. «Understanding the Limits of LoRaWAN». In: *IEEE Communications Magazine* 55.9 (2017), pp. 34–40. DOI: 10.1109/MCOM.2017.1600613 (cit. on p. 20).

[26] Lorenzo Vangelista, Andrea Zanella, and Michele Zorzi. «Long-Range IoT Technologies: The Dawn of LoRa™». In: *Future Access Enablers for Ubiquitous and Intelligent Infrastructures*. Ed. by Vladimir Atanasovski and Alberto Leon-Garcia. Cham: Springer International Publishing, 2015, pp. 51–58. ISBN: 978-3-319-27072-2 (cit. on p. 21).

[27] Alessandro Lovesio. «Design of a Neural Network development framework for plant monitoring applications». Master's Thesis. Politecnico di Torino, 2021. URL: https://webthesis.biblio.polito.it/21030/ (cit. on p. 50).

[28] Federico Cum. «A Neural network application for impedance-based plant monitoring: from a development framework towards edge computing». Master's Thesis. Politecnico di Torino, 2023. URL: https://webthesis.biblio.polito.it/24471/ (cit. on p. 50).

[29] Facebook and Microsoft. *Logo of the Open Neural Network Exchange*. https://www.apache.org/licenses/LICENSE-2.0. Logo of the Open Neural Network Exchange, a neural network data exchange format. 2017 (cit. on p. 56).

[30] Apache Software Foundation. *Apache License, Version 2.0*. https://www.apache.org/licenses/LICENSE-2.0. Accessed: 2024-06-23. 2004 (cit. on p. 56).

[31] STMicroelectronics. *LoRaWAN® firmware update over the air with STM32CubeWL*. Accessed: 2024-07-06. Jan. 2023. URL: https://www.st.com/resource/en/application_note/an5554-lorawan-firmware-update-over-the-air-with-stm32cubewl-stmicroelectronics.pdf (cit. on p. 70).