TRABAJOS FIN DE GRADO

Universidad de Jaén

Escuela Politécnica Superior de Jaén

Sustainable AI Kernel for Education on ARM-Based Low-Cost Devices

Autor: Fedi Nabli

Grado: Ingeniería en Informática

Directores: Francisco José Quesada Real y Álvaro Labella Romero
Departamento del director: Informática

Fecha: 10/6/2025

Acknowledgements

I would like to express my heartfelt gratitude to everyone who contributed in one way or another to the completion of the project. To those who supported me, believed in me, and accompanied me throughout this journey, I offer my sincerest thanks.

I would especially like to thank my academic supervisors, Dr. Francisco José Quesada Real and Dr. Álvaro Labella Romero, for their invaluable guidance, advice and insightful feedback throughout this endeavour.

I am also deeply grateful to Dr. Ines Baccouche, my academic supervisor at my home university, Pristini School of AI, for her support and insightful feedback during this project.

I would like to deeply thank my parents Hazem and Imene for their support along this journey and for all my friends and family members who supported all the way during this endeavour.

Finally, I extend my appreciation to the jury members for taking the time to review my project.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

ABI: Application Binary Interface
AI: Artificial Intelligence
ARM: Advanced RISC Machine
ASCII: American Standard Code for Information Interchange
AST: Abstract Syntax Tree

BIOS: Basic Input/Output System

CLI: Command Line Interface
CPU: Central Processing Unit
CSV: Comma Separated Values

DL: Deep Learning
DRAM: Dynamic RAM
DSL: Domain Specific Language

EL: Exception Level
EOF: End Of File

FFI: Foreign Function Interface
FIQ: Fast Interrupt reQuest

GIC: Generic Interrupt Controller
GICC: CPU Interface
GICD: GIC Distributor
GPIO: General-Purpose Input/Output
GPU: Graphics Processing Unit
GUI: Graphical User Interface

HDD: Hard Disk Drive

IoT: Internet of Things
IR: Intermediary Representation
IRQ: Interrupt ReQuest
ISR: Interrupt Service Routine

JSON: JavaScript Object Notation

LR: Linear Regression

MB: Megabyte
ML: Machine Learning
MMU: Memory Management Unit
MPU: Memory Protection Unit
MSE: Mean Squared Error

NPU: Neural Processing Units
NVMe: Non-Volatile Memory Express

OER: Open Educational Resource
OLS: Ordinary Least Squares
OS: Operative System

RAM: Random Access Memory
RISC: Reduced Instruction Set
ROM: Read Only Memory

SDGs: Sustainable Development Goals
SGD: Stochastic Gradient Descent
SIMD: Single Instruction Multiple Data
SoC: System on Chip
SRAM: Static RAM
SSD: Solid State Drive

UART: Universal Asynchronous Receiver/Transmitter
UI: User Interface
USB: Universal Serial Bus

VFS: Virtual File System

YAML: Yet Another Markup Language

Chapter 1

INTRODUCTION

1.1 Project Introduction

Artificial Intelligence (AI) has emerged as one of the most transformative forces of the 21st century, revolutionizing diverse sectors such as healthcare, education, transportation, finance, and environmental monitoring [3]. Despite its vast potential and benefits, the growing dependency on powerful computational infrastructures has introduced several limitations [4, 5]. AI algorithms, especially those used in Deep Learning (DL) and large-scale data processing, are highly demanding in terms of hardware. Their effective execution generally requires high-performance Graphics Processing Units (GPUs), substantial Random Access Memory (RAM), and fast storage systems. These requirements translate into high energy consumption and financial costs [6].

This situation raises two major concerns. First, the environmental impact of AI's energy usage is increasingly under scrutiny, with the need for sustainable computing models becoming more urgent. Second, the access to AI technologies and education is highly uneven across the globe. Institutions in economically disadvantaged or geographically isolated regions often lack the hardware and infrastructure needed to engage meaningfully with AI, perpetuating a digital divide in technological literacy and innovation capacity [7].

To address these challenges, this project proposes the development of a Sustainable AI Kernel specifically optimized for ARM-based low-cost devices. ARM (Advanced RISC Machine) processors [8] are widely used in embedded systems and single-board computers like Raspberry Pi [9] and BeagleBone [10] due to their cost-effectiveness and energy efficiency. These devices, though limited in processing power, are globally accessible and affordable, making them ideal platforms for introducing AI in low-resource environments.

The proposed Sustainable AI Kernel is a lightweight software layer designed to manage and execute AI algorithms efficiently on ARM-based hardware. Through techniques such as model compression, quantization, pruning, and edge AI processing, the kernel will allow AI tasks to run on devices with limited resources. This not only en-

hances energy efficiency but also extends the usability of older or repurposed devices, encouraging responsible consumption and production practices.

The motivation behind this proposal is grounded in its potential to democratize AI education, reduce environmental impact, and promote sustainable innovation. By enabling AI execution on low-cost hardware, we make AI accessible in educational institutions with limited infrastructure, opening new opportunities for students and educators worldwide. Furthermore, the approach has the potential to create smart environments in sectors like agriculture, health, and education, using recycled or donated devices.

Undertaking projects with a tangible social impact is essential in today's global context. To maximize their relevance and effectiveness, it is crucial that such initiatives are aligned with the 2030 Agenda [11], which outlines the United Nations' Sustainable Development Goals (SDGs) (see Figure 1.1).

Figure 1.1: 2030 Agenda — Sustainable Development Goals. Source: [1])

This project, in particular, contributes directly to the achievement of the following SDGs:

- *SDG 4: Quality Education*. Fostering inclusive and equitable access to quality education by integrating AI tools into classrooms regardless of economic background.
- *SDG 9: Industry, Innovation, and Infrastructure*. Promoting resilient and sustainable infrastructure for innovation using energy-efficient, low-cost technologies.
- *SDG 10: Reduced Inequalities*. Reducing disparities in access to digital resources and AI education across regions.
- *SDG 12: Responsible Consumption and Production*. Encouraging hardware reuse and energy-efficient computing.

- *SDG 13: Climate Action*. Minimizing the carbon footprint of AI computation through optimized software on sustainable hardware.

Beyond its educational relevance, the Sustainable AI Kernel can be instrumental in deploying intelligent low-cost applications such as smart classrooms, environmental monitoring, and telemedicine systems in rural or underserved regions. It can also be released as an open educational resource (OER), offering students the chance to explore, customize, and contribute to the kernel itself, deepening their understanding of AI and embedded systems.

To address this challenge, this project aims to bridge the gap between AI's computational demands and the need for equitable and sustainable access. By developing a Sustainable AI Kernel tailored for ARM-based low-cost devices, we create a pathway toward inclusive, efficient, and environmentally responsible AI education. This initiative has the potential to empower communities, foster innovation, and drive global AI literacy through accessible and sustainable technologies. To achieve this, the project will pursue a set of specific objectives, outlined in Section 1.2, which will guide the design and implementation of the proposed solution.

1.2 Objectives

- Develop a microkernel/hybrid kernel to run the AI engine.
- Create an AI engine optimized for ARM-based robotics kernels.
- Design and implement a CLI for model parameter definition and management.
- Develop JSON/CSV parsers for data and model loading.
- Evaluate system performance in real-world robotics applications.
- Document the entire process, including installation and user manuals

1.3 Planning

This project was developed following a module development approach [12]. Given that the different project tasks are not equal, a non-standard method was used during the planning and development of this project.

Unlike agile SCRUM [13] or waterfall [14] methodologies, the module method does not set a regular time for each component. However, according to the importance of the task, its relation to the other components and significance, each was given a time-frame to be designed and implemented correctly.

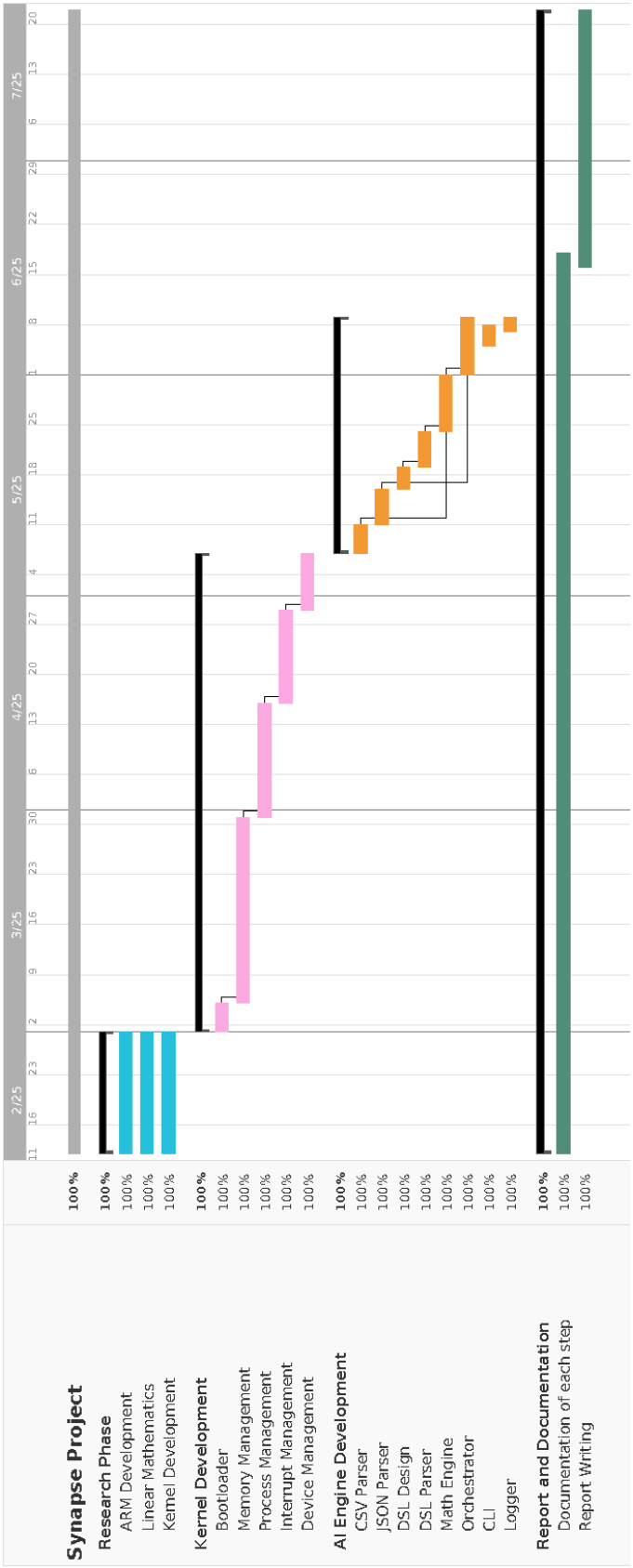

Figure 1.2: Project Timeline. Source: Own creation

The Gantt chart (Figure 1.2) outlines the planned and final schedule, organized into the different components and their timelines. The project was split into four phases:

the research, the kernel development, the engine development, and the project documentation and reporting.

1.4 Budget

Although the planing and development of the current version of the `Synapse Robotics` project did not involve direct financing, it is important to quantify the direct and indirect costs of developing the same product. This section will detail both the personnel budgeting and services and facilities budgeting to recreate the same product.

1.4.1 Personnel Budget

Each product needs personnel to create and manage it. Table 1.1 summarizes the personnel budget spendings. It lists the roles, their functions in the project, their salaries, the duration of their work, and the final cost. The C++ developer five months' engagement corresponds to his work, but also overviewing the whole project.

Role	Function	Annual Cost	Monthly Cost	Duration	Final Cost
Compiler Developer	Responsible for designing and implementing the different parsers for the AI Engine.	€60,000	€5,000	2 Months	€10,000
Rust Developer	Responsible for designing and developing the Math Engine in Rust	€ 60,000	€5,000	3 Months	€15,000
C++ Developer	Responsible for the Orchestrator and overviewing the AI Engine development	€65,000	€5,400	5 Months	€27,000
Kernel Developer	Responsible for developing the core components of the kernel	€80,000	€6,600	3 Months	€19,800
Embedded Developer	Responsible for developing the hardware specific kernel modules	€77,000	€6,500	3 Months	€19,500
User Program Developer	Responsible for developing kernel services and user space programs such as the terminal and developer libraries.	€75,000	€6,250	2 Months	€12,500
				Total	€103,800

Table 1.1: Personnel Budget

1.4.2 Services and Facilities Budget

For the project development, the personnel would need services and facilities to finish the product. These indirect costs contribute to the success of the project by providing necessary services such as electricity, internet, and development hardware. Table 1.2 lists the resources, their function, the cost, and the final cost of these services in order to successfully create the finished product.

Resource	Function	Monthly Cost	Duration	Final Cost
Internet	Used for research, downloading datasheets and opening documentations	€50	5 Months	€250
Electricity	Power required to operate the development machines and testing hardware during the development of the project	€25	5 Months	€125
Hardware Use	Estimated depreciation or usage cost of the personal computer used for development (MacBook Pro M4 Pro 48 GB RAM, suitable for embedded development and machine learning)	€75 / Month	5 Months	€375
			Total	€750

Table 1.2: Services and Facilities Budget

1.4.3 Total Cost

After calculating the direct and indirect costs for developing the final product, the final cost was calculated at: `€ 104,550`.

1.5 Structure of the Document

This document is structured into several chapters and appendices linked together to guide the reader into the full understanding of the system, the choices behind it, defining the concepts, and evaluation of it. The document is structured into the following chapters:

Chapter 2 provides the theoretical and technical knowledge needed to understand the project deeply. This chapter talks about three major concepts: AI, compilers and OS development. First, the chapter starts by introducing AI and Machine Learning (ML) and their types. The chapter then transitions into the math behind ML algorithms and talks about Linear Algebra (and its different types) to then transition to the implemented model in this project, which is Linear Regression. It goes deep into explaining the model internally and the math behind it. Second, the chapter explains the concepts of compilers and parsers and their different types. Finally, the chapter introduces OS concepts and theory, detailing each component that will be implemented.

Chapter 3 details the implementation and decision behind each component of Synapse Engine and Synapse Kernel projects. It begins with a comprehensive description of the project and sets the requirements needed for it, and with those the project will be measured. It then talks about the AI Engine (Synapse Engine) development starting with the design choices, the architecture before going into each component separately. Finally, the chapter discusses the design choices and implementation of the Synapse Kernel subproject highlighting each component and giving details about their insides.

Chapter 4 is the evaluation chapter, in it, I have run experiments about the final AI engine and compare them with the industry standard ways. This chapter defines metrics to test the system and discusses each measure in details for different scenarios.

Both performance and accuracy benchmarks are presented and analyse highlighting the project's strength and limitations.

Chapter 5 is the conclusion chapter. This chapter summarizes the main finding and the goal of the project, it discusses the purpose and how it was achieved by the implementation. It then discusses the evaluation findings, what limitations and challenges I was presented with, and finally introduces future work on the project that leads to a better performance and more impact.

Appendix A is the installation manual. In this appendix, I walk through the steps to install the Engine and test it on your local machine.

Appendix B is the user manual. In this appendix, I talk about the different commands that the engine exposes and how to use it to train the first model using this tool.

Chapter 2

BACKGROUND

2.1 Introduction

During this chapter, we will have an in-depth investigation of the theoretical foundations and current state-of-art advancements in the fields of AI, ML, Operating Systems (OS), and Embedded Robotics. We will explore core components of each and provide a comprehensive overview of AI Algorithms, Parser, OS concepts, functionalities, and technologies.

In addition, we will review different methodologies and technological approaches, highlighting the importance of optimized performance for Embedded Robotics especially for AI. We will also review end-to-end solutions used in robotics. This conceptual study aims to provide a solid understanding of these technologies and their current applications.

2.2 Artificial Intelligence

AI enables computers to simulate human intelligence and problem-solving capabilities. Whether be used alone or in combination with other technologies such as robotics, sensors, or embedded systems, AI can perform tasks and jobs that usually require human intervention or human intelligence. [15, 16]. Current examples of AI in our daily lives include digital assistants, generative AI tools such as OpenAI's ChatGPT [1], autonomous vehicles, and embedded robots.

As a field of Computer Science, AI encompasses ML and deep learning, which are often discussed together (Figure 2.1) [15]. These disciplines involve developing AI algorithms inspired by the human brain's decision-making processes, capable of "learning" from data to provide increasingly accurate classifications or predictions over time. AI has experienced several cycles of popularity; put the release of ChatGPT

[1] https://chatgpt.com/

seems to mark a turning point, even for septics. The last major breakthrough was in Generative AI Agents, but today Embedded AI in robotics had made significant strides.

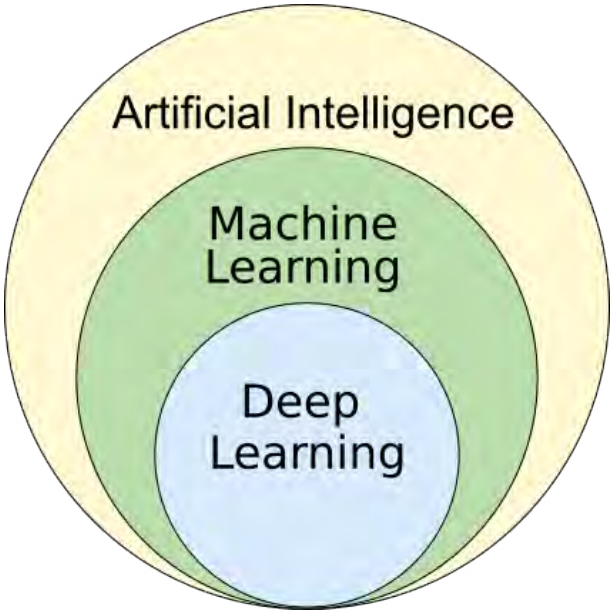

Figure 2.1: AI main subfields. Source: Wikimedia Commons

2.3 Machine Learning

2.3.1 Definition

ML [17, 18] is a subfield of AI that uses a combination of data and mathematical algorithms to create models capable of learning and making predictions based on the data, without the need to explicitly code the features. The learning can be done through three main types, depending on the nature of the data and the task. These types are:

1. **Supervised Learning**
    - **General Concept**: Supervised learning algorithms are trained on pairs (data, label), where each data entry has a label representing the prediction target. The goal is to generalize this knowledge to predict the target for unseen data.
    - **Relation to Embedded AI**: In Embedded AI, supervised learning can be trained on sensors data to predict how the system should behave in the feature.
2. **Unsupervised Learning**
    - **General Concept**: In Unsupervised learning, the data provided is not labeled, and the model objective is to identify patterns within the dataset on its own.

- **Relation to Embedded AI**: Unsupervised Learning can be used in Embedded AI and robotics for multiple cases, such as anomaly detection to predict failures and self-maintenance, sensor fusion for cleaner input for navigation/-control, and environment clustering to adapt to new space.

3. **Reinforcement Learning**

   - **General Concept**: This type of ML employs a trial-and-error strategy by implementing penalties and rewards to train the model. The model interacts with its environment and learns from its past actions.
   - **Relation to Embedded AI**: Reinforcement Learning can be used to optimize resources for power-constrained embedded systems, where it learns when to activate sensors, sleep mode or compresses data, or for navigation and obstacle avoidance, where it learns to explore unknown maps with limited energy.

2.3.2 ML Algorithms

ML algorithms enable a computer to learn from patterns from data and thus, these machines are now able to perform tasks such us prediction, classification, or clustering. Depending on the nature of the task and the data, these algorithms fall broadly into two major categories: *Regression* and *Classification*.

Regression Algorithms [19] predict continuous numerical outcome by modelling relationships between dependent variable (target) and independent variables (features). They aim to fit the best possible curve to the dataset. The most commonly utilized regression algorithms include:

- **Linear Regression** [20]: The simplest regression algorithm. It fits a straight line to observed data points by minimizing the squared differences between observed and predicted values. Mathematically, the model is expressed as:

$$y = \beta_0 + \beta_1 x \tag{2.1}$$

  where:

  - y: dependent variable (target)
  - x: independent variable (feature)
  - β_0: intercept (bias)
  - β_1: slope (weight)

  Linear Regression is widely used in embedded systems for simple predictive tasks such as sensor calibration, battery life estimation, or forecasting resource consumption due to its low computational complexity and minimal memory footprint. Figure 2.2 shows a linear regression example, fitting data points to a straight line.

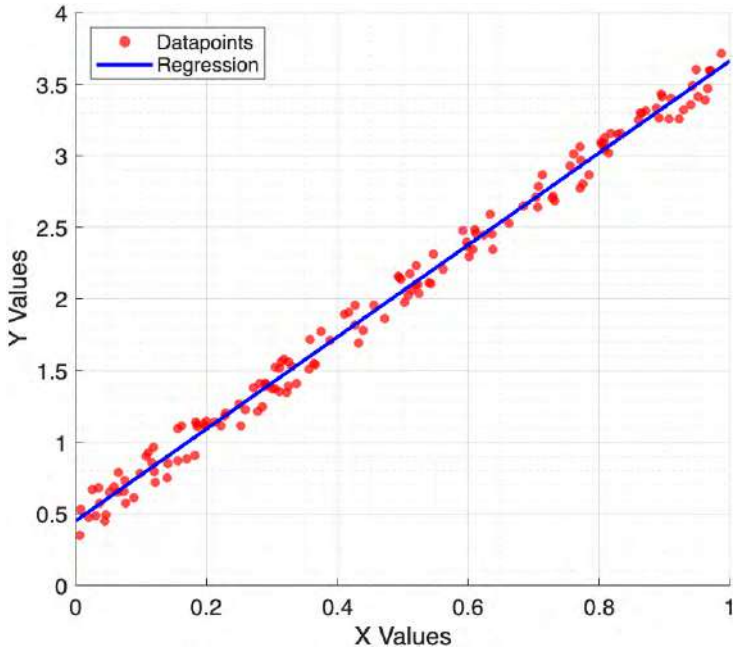

Figure 2.2: Linear Regression Source: Own creation

- **Polynomial Regression** [21]: It expands on linear regression by considering polynomial relationships among variables, allowing the modeling of more complex, non-linear data. The mathematical expression is:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_n x^n \tag{2.2}$$

  where:

  - n: degree of the polynomial
  - y: dependent variable (target)
  - x_i: independent variables (features)
  - Other parameters ($\beta_0, \beta_1, \ldots, \beta_n$): represent coefficients of polynomial terms.

  Polynomial regression is effective for embedded system applications that require modelling non-linear sensor readings, complex environmental data prediction, or anomaly detection. Figure 2.3 shows a polynomial regression example fitting data points on a non-linear curve.

2.3.3 Linear Algebra

Linear Algebra [22] is a fundamental area of mathematics that studies vector spaces, linear transformations, matrices, and systems of linear equations. Its methods are central to modelling and solving complex computational problems efficiently and reliably, making it an essential mathematical tool in numerous scientific and engineering fields, particularly in AI and ML.

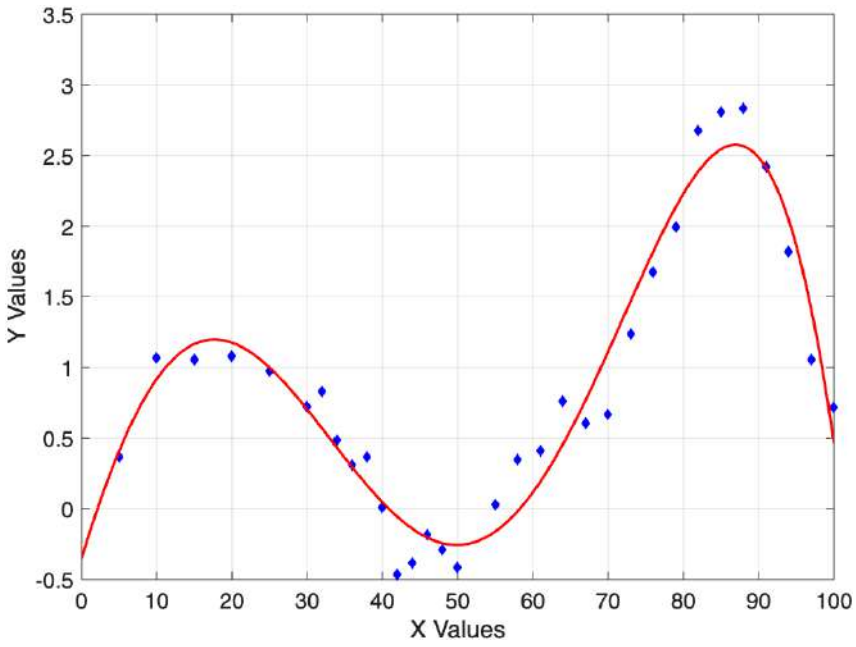

Figure 2.3: Polynomial Regression. Source: Own creation

Linear algebra provides the mathematical backbone required by many AI algorithms. Most ML algorithms-particularly in supervised and unsupervised learning-heavily depend on linear algebraic concepts, including vectors, matrices, and linear transformations. Operations such as matrix multiplication, and solving linear systems underpin core ML processes like optimization, dimensionality reduction, and data transformations.

Linear Algebra offers:

- **Computational Efficiency:** Enables efficient handling and manipulation of large datasets through matrix operations.
- **Optimization Capability:** Provides mathematical structures essential for optimization algorithms, facilitating model training and convergence analysis.

Key mathematical elements of linear algebra are essential for understanding subsequent sections, particularly Linear Regression, include:

- **Vectors and Vector Spaces:** A vector is an ordered list of numbers, typically representing features or data points in an ML context. Vector space defines the mathematical framework within which vectors operate. For instance, a vector $\mathbf{v}$ in $\mathbb{R}^n$ is represented as:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \tag{2.3}$$

- **Matrices:** A Matrix is a two-dimensional array of numbers used extensively to represent datasets, transformations, and relationships between variables. A matrix $\mathbf{A}$ with m rows and n columns is defined as:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \tag{2.4}$$

- **Linear Transformations:** Linear transformations map vectors from one vector space to another while preserving vector addition and scalar multiplication. They form the basis for understanding linear models.

  Mathematically, a linear transformation T can be represented as:

$$T(\mathbf{x}) = \mathbf{A}\mathbf{x} \tag{2.5}$$

  where $\mathbf{A}$ is a transformation matrix.

- **Systems of Linear Equations:** Systems of linear equations represent relationships among variables and are central in solving for model parameters. They are typically expressed in matrix form:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{2.6}$$

  where:

  - $\mathbf{A}$: coefficient matrix.
  - $\mathbf{x}$: vector of unknown variables.
  - $\mathbf{b}$: vector of known outcomes.

Understanding these fundamental elements of linear algebra lays the necessary foundation for comprehending advanced ML algorithms, especially linear regression methods, which leverage these concepts to model and solve real-world problems effectively.

2.3.4 Linear Regression

Linear Regression [20] is a fundamental supervised learning algorithm extensively used for predicting continuous numerical outcomes by modelling linear relationships between a dependent variable and one or more independent variables. Due to its simplicity and interpretability, it is widely applied in various scientific and engineering domains.

Simple linear regression models the relationship between a single independent variable $\mathbf{x}$ and a dependent variable $\mathbf{y}$. A linear equation models this relationship.

$$y = \beta_0 + \beta_1 x + \epsilon \tag{2.7}$$

where:

- $\mathbf{y}$: Dependent variable (target)
- $\mathbf{x}$: Independent variable (feature)
- β_0: Intercept (bias)
- β_1: Slope (weight)
- ϵ: Error term (residual), representing unexplained variance

The parameters β_0 and β_1 are estimated by minimizing the mean squared error (MSE) between the predicted and actual values:

$$MSE = \frac{1}{n}\sum_{i_1}^{n}(y_i - (\beta_0 + \beta_1 x_i))^2 \tag{2.8}$$

Linear regression fits a straight line to data points by minimizing the distance (errors) between actual and predicted points (Figure 2.4)

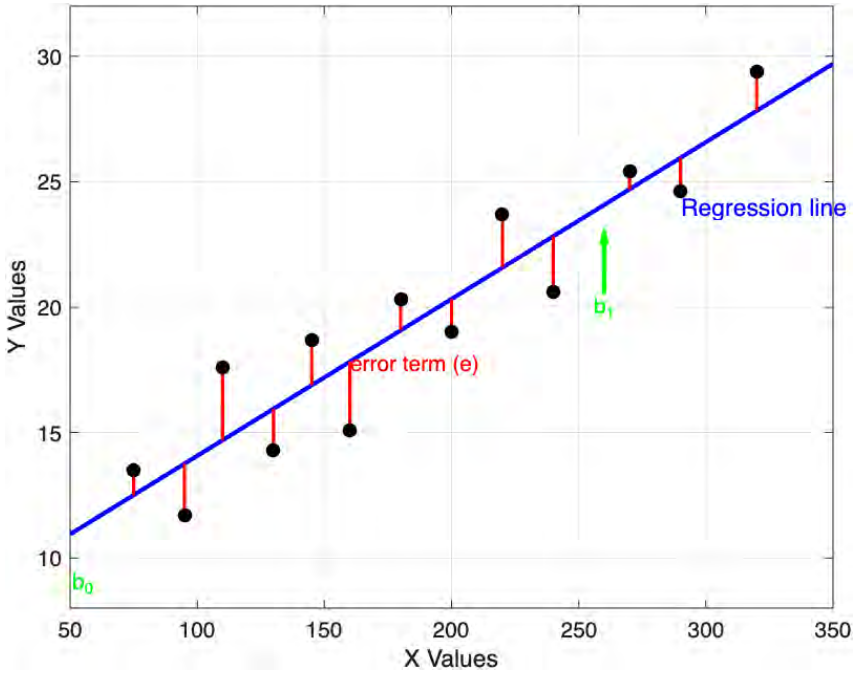

Figure 2.4: Simple Linear Regression. Source: Own creation

On the other hand, *Multiple Linear Regression* [23] extends simple linear regression to accommodate multiple independent variables, allowing the prediction of a dependent variable based on a linear combination of multiple features. The general form is expressed as:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n + \epsilon \tag{2.9}$$

In matrix form, it is represented concisely as:

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon} \tag{2.10}$$

where:

- $\mathbf{y}$: vector of observed values
- $\mathbf{X}$: feature matrix (include bias term as first column of ones)
- $\boldsymbol{\beta}$: vector of coefficients ($\beta_0, \beta_1, \ldots, \beta_n$)
- ϵ: vector of residual error

The optimal coefficients $\boldsymbol{\beta}$ are typically calculated using the Ordinary Least Squares (OLS) method:

$$\boldsymbol{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{2.11}$$

While analytical solutions such as OLS exist, they can become computationally infeasible for very large datasets. Therefore, optimization algorithms like *Gradient Descent* are commonly used. Gradient Descent iteratively adjusts model parameters ($\boldsymbol{\beta}$) to minimize the cost function MSE:

$$\boldsymbol{\beta}^{(new)} = \boldsymbol{\beta}^{(old)} - \alpha \frac{\partial}{\partial \boldsymbol{\beta}} J(\boldsymbol{\beta}) \tag{2.12}$$

where α is the learning rate controlling step size and $J(\boldsymbol{\beta})$ is the cost function.

There are different approaches based on the Gradient Descent. For example, *Batch Gradient Descent* computes the gradient using the entire dataset at each iteration. This provides stable but computationally intensive updates, defined mathematically as:

$$\frac{\partial}{\partial \boldsymbol{\beta}} J(\boldsymbol{\beta}) = -\frac{2}{n}\mathbf{X}^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) \tag{2.13}$$

Pros: Stable convergence.

Cons: High computational cost per iteration for large datasets.

Another example is Mini-Batch Stochastic Gradient Descent (SGD), which offers a compromise by computing gradients using a subset (“mini-batch”) of data points at

each iteration. This reduces computational cost while maintaining stability. The gradient calculation at iteration t is:

$$\frac{\partial}{\partial \boldsymbol{\beta}} J(\boldsymbol{\beta}) = -\frac{2}{m} \mathbf{X}_t^T (\mathbf{y}_t - \mathbf{X}_t \boldsymbol{\beta}) \tag{2.14}$$

where:

- m: Size of mini-batch (subset of data points)
- $\mathbf{X}_t, \mathbf{y}_t$: Feature matrix and target vector for the mini-batch at iteration t

Mini-Batch SGD iteratively selects random subsets of data until convergence, striking a balance between computational efficiency and convergence reliability.

Pros: Efficient computation, suitable for large datasets, better generalization.

Cons: Moderate convergence reliability.

Figure 2.5 illustrates how mini-batch gradient descent progresses towards convergence through iterative parameter updates.

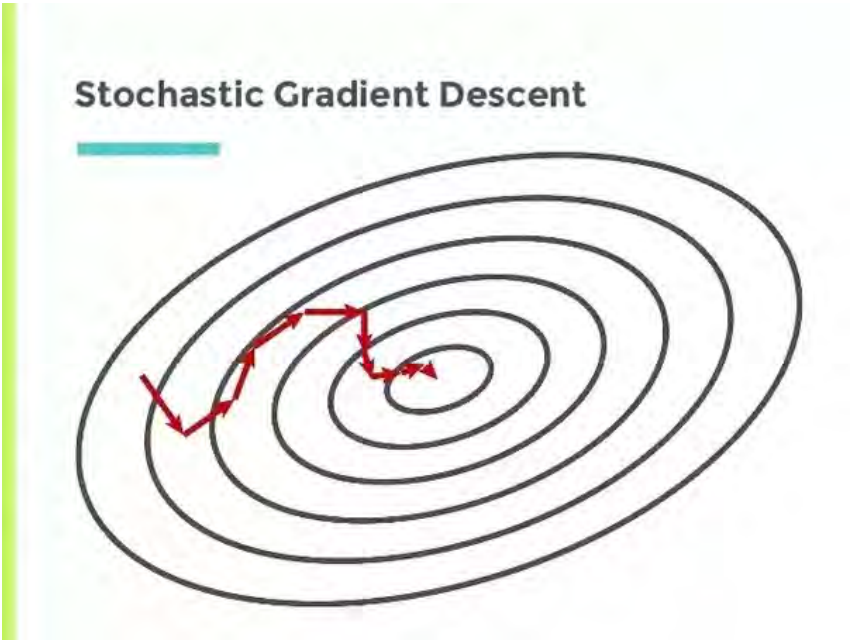

Figure 2.5: Mini-Batch SGD Convergence. Source: [2]

Linear Regression and its variants (especially optimized with mini-batch SGD) are ideal for embedded AI applications due to their simplicity and efficiency. Common use cases include real-time sensor calibration, continuous environmental monitoring, predictive maintenance, and energy consumption forecasting.

2.4 Compilers General Concept

Parsers and compilers [24] play a critical role in software systems that involve complex data processing, programming languages, or domain-specific languages (DSLs). These programs are responsible for interpreting input raw data and transforming it into a structured format, understandable by humans and computer programs. Parsers are fundamental for much software including compilers and interpreters (e.g., programming languages), configuration files, and many other components.

The main responsibility for a parser is to check the validity of the input data based on a predefined grammar rules. Like human languages, computer languages such as programming languages, configuration file (e.g., YAML, JSON, etc.), and DSLs also have a predefined syntax and grammar rules. The parser uses these to validate, and check for correctness of the input file.

To achieve the wanted results, parsers typically go through many stages. There are two parser classes, One-Pass and Two-Pass parsers. The name refers to how many times the parser reads the raw input to achieve the same results. One-Pass parsers only goes through the input once, and the two-pass goes through two passes. In the first, they create an Intermediary representation (IR) that the second pass interprets and validates.

One-Pass parsers are simpler, smaller, and usually tailored for a subset of a language or highly structured data, whilst Two-Pass parsers are tailored for more sophisticated languages or those who do not have a highly structured data such as programming languages. In the next sections, we will be looking more into each, starting with the One-Pass parsers.

2.4.1 One-Pass Parsers

One-Pass parsers (see Figure 2.6) interpret the input data in a single pass through the input. These parsers combine the three main stages into one. These stages consist of lexical analysis, syntax analysis, and semantic analysis in some cases. This approach has several advantages for simpler data formats, including:

Figure 2.6: One-Pass parsers. Source: Own creation

- **Simplicity**: One-Pass parsers implementation and logic are straightforward because of the highly structured input data, which facilitates debugging, maintenance, and have smaller size.

- **Efficiency**: Since One-Pass parsers performs all processing in a single pass, they are more efficient in limited resources and structured data since they do not need to retain any data, resulting in less memory footprint and faster processing.

These types of parsers are very simple and straightforward as discussed earlier, thus they are useful for highly structured input data and configuration files such as Comma Separated Values (CSV), JavaScript Object Notation (JSON) subsets, and Yet Another Markup Language (YAML) files. One-Pass parsers are used therefore mainly where immediate processing and low overhead are critical.

The phase of the parser algorithm differentiate from one implementation to another depending on the parser data. They can parse by line, by statement, or using other approaches. But, they all share similarities in their functionality. They usually do some lexical or syntactic analysis (e.g., if the data type is supported), and basic validation (e.g., a JSON file should start with `{` and end with `}`.

Let us take an example of a CSV file. A CSV file has two main parts, the header, and the data. A one-pass parser can then split the file line by line, knowing the first line is always a header it registers it as the column names and all the next lines, make a row. The simplicity of this type of file also lets the validation happen at the same time. Once we have a line, we can split the fields by `,` or a similar delimiter, which gives us a list, thus we can check that the length of the rows and columns number are exact and even handle some null values and replacing them.

This shows the efficiency and practicality of One-Pass parser, but what happens when the grammar and syntax get more complicated? Usually, one-pass is not enough and there is a need to separate each component by its own, which gives us Two-Pass parsers.

2.4.2 Two-Pass Parsers

Two-Pass parsers, as the name suggests, processes the input data into two distinct phases, separating the lexical and semantic analysis. The first phase usually outputs an IR that gets passed to the second phase, which reads the IR and does semantic analysis. Whilst Two-Pass parsers might need more resources, this technique enables parsers and compilers to process complex grammatical syntax, making it a better fit for programming languages, modular configuration files, and DSLs.

The Two-Pass parsers stages (or passes) communicate through an Intermediate Representation, usually an Abstract-Syntax Tree (AST). The first pass, responsible for lexical analysis, consists of three stages, a lexer, a tokenizer, and a parser. The last stage generates an AST and gets passed to the second pass as its input. The second pass, responsible for semantic analysis, usually consists of one major component which is the validator that traverses the Intermediate Representation and finally outputs the data into a structured output as Figure 2.7 shows.

To understand each component better, let us take a more detailed look at each:

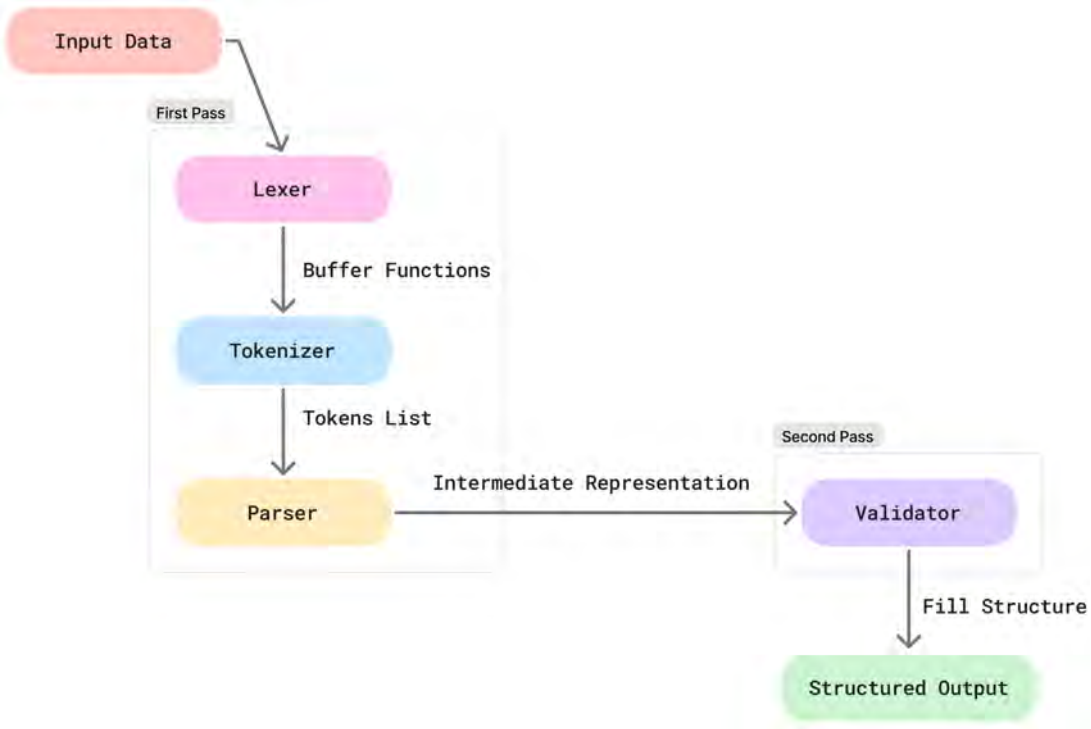

Figure 2.7: Two-Pass parsers. Source: Own creation

- **Lexer**: The lexer does not do any processing on the data, but provides helper functions to the tokenizer to traverse the text and extract only important details.

- **Tokenizer**: The tokenizer is the first text processing component. Its role is to split the raw text input into a token structure. A token is a structure that contains information about a subset of a text.

- **Parser**: The parser is where the core logic of parsing is implemented. It takes the token list and creates a tree that represents the program into a computer friendly format that the validator can use. In this component, some tokens are discarded since their only purpose is basic grammar validation used only by the parser. This component also does basic grammar validation (e.g., verifying an assignment has a keyword token, an equal sign token =, a value token, and in some cases an end statement token such as ;. This performs the basic validation while creating the AST node by node.

- **Validator**: The validator is the component of the second stage, it takes the now more computer friendly structure from the parser component in the first stage, it does final semantic validation and fills the final structure to be used by another program or goes through further processing.

With the basic concepts known, we must understand further how each component work, which we will explore in the following subsections, starting with the lexer.

2.4.2.1 Lexer

The lexical analyser, commonly known as the lexer, is the entry point of a parser. In this component, no processing happens, but its purpose is to provide helper functions to the next stage in the pipeline, the tokenizer, to process the text.

The lexer scans the raw input text character-by-character and puts it into a structure that contains the buffer and the helper functions. A buffer is a simple linear array containing all the characters in a sequential order. This enables indexing, and since the array has a fixed size, this helps more and gives more security.

The raw text input is not pure, it contains some text not needed to understand the intent. For example, programming languages has comments, these comments are intended for the programmers to understand the piece of code, but they do not serve any purpose for the computer to understand the program. Another example is the end-of-line character(s). The lexical analyser provides a common function to the tokenizer when it is traversing this input to ignore these unnecessary parts or for accessing a part of the input.

The lexer structure exposes some helper functions alongside some variables to track the positioning. Some functions include:

- **Peek**: Allows the tokenizer to “look” at an upcoming character without advancing the current position.
- **Advance**: The function “consumes” the current character, returns it, and advances the position cursor to the next character, therefore consuming it.
- **Position Tracking**: The lexer structure maintains pointers to the read and next characters and provides functions to get a subset of the input.
- **Skipping Functions**: These functions skip over sections of the input such as comments, skip characters, etc...

Whilst the lexer does generate any tokens directly, it lays the foundation for the next stage, which is the tokenizer, by providing helper functions and data regarding the input.

2.4.2.2 Tokenizer

The tokenizer is the second stage in the pipeline, it leverages the lexer’s functions to split the input text into a series of tokens that are used for further processing of the input.

A token is a structure that contains information about a specific subset of the text. Usually, a token contains the lexeme (the raw text as a string), the position, the token type, and the value formatted into the correct type (e.g., convert numbers).

To understand better the process of tokenization, let us take an example of the following input:

```
int res = 10 + 2;
```

Figure 2.8 shows how a tokenizer would split the input.

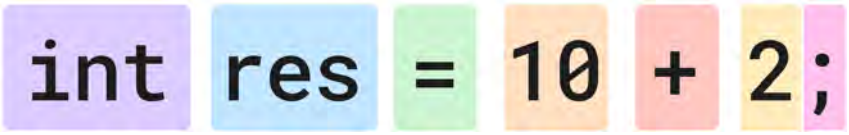

Figure 2.8: Tokens Splitting. Source: Own creation

We can see that the tokenizer split the sentence into seven separate tokens, ignoring whitespace between each token. Each token in the figure is represented by a different coloured box. The tokens consist of:

- **Keyword**: A keyword represents a reserved identifier of the language. it is used to give more context such as conditioning, branching, etc.... In the example above, the word `int` represents a keyword that defines the data type.
- **Identifier**: An identifier is a name that represents parts of the language such as a variable name or a function name, it is a simply a user defined item serving as a label. In the example above, the word `res` in an identifier that defines a variable which holds the result of the equation.
- **Operators**: An operator is used either in assignment or for an equation. In the example above, we can see two operators that serve two different purposes. First, the `=` (equals) operator used to assign the result value to the `res` identifier. Second, the `+` (plus) operator, which is the same mathematical plus to add the two number tokens.
- **Number**: A number token is just a token holding the formatted number value. The text is a series of characters, thus in order to process it correctly, the tokenizer, when it hits a number, it converts it into a whole or decimal number and stores it in the token value with the correct data type.
- **Semicolons & other special characters**: Special characters depends on from a language to another, but few of the most common ones are semicolons (`;`), commas (`,`), and colons (`:`). In the example above, we can see the semicolon that represents the end of the statement.

The tokenizer process use four main jobs:

- **Lexeme Identification**: Using the lexer's helper functions, the tokenizer detects lexemes (a subset of the input text), does some basic validation and creates a token. A lexeme detection is mostly done by using a delimiter, usually a comma (`,`) or whitespace(s).

- **Token Creation**: After detecting the lexeme, the tokenizer detects the data type, uses the lexer's function to get a slice position, puts everything together into a structure, and converts the string into the correct type (e.g., string to number)
- **Error Handling**: After identifying the lexeme, the tokenizer proceeds to validate the lexeme (e.g., a decimal number should have number(s), a point, preceded by number(s), or if a keyword is written correctly).
- **Skipping**: The tokenizer ignores comments, white spaces, and other non-significant characters using the lexer's helper functions.

The tokenizer goes through the whole input creating token by token until it reaches the End Of File (EOF) outputting a tokens list that gets passed to the final component of the first stage: the parser.

2.4.2.3 Parser

The parser is the final stage of the first phase. It takes the token stream from the previous stage and proceeds to creating an Intermediary Representation whilst doing lexical analysis.

Let us take the same example of earlier: `int res = 10 + 2;`. Figure 2.9 shows the final output of the first phase, the AST. But we can see that some tokens are missing. Let us explore how the parser takes the tokens, creates the AST from nodes, and validates the statement.

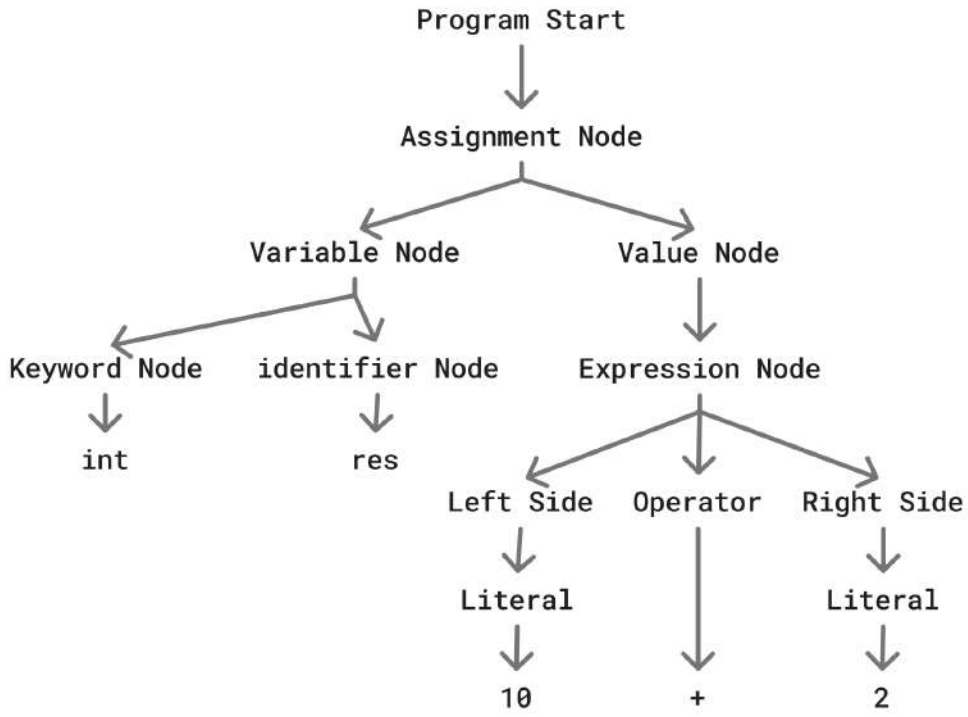

Figure 2.9: AST Example. Source: Own creation

The grammar of the language can be written as follows:
`keyword identifier equals literal operator literal semicolon.`

The equals sign is used to signal an assignment operation: the identifier holds the computed value of the value node, and the semicolon signals the termination of the statement. These two tokens does not serve any purpose other than helping the creation of the AST.

Let us now walk through the detailed creation process. First, the parser starts reading the token stream, it encounters the `int` which is a keyword token representing a data type and from it, it creates a keyword node. The parser knows that the next token should be an identifier token based on the grammar rules. Thus, first it checks the type of the next token. If it matches the expected one, it created an identifier node and merges the two nodes (keyword node + identifier node) to form a variable node. If the peeked token does not match the expected type, the parser throws an error.

Now, we have the right side of the equation. Next, the parser sees the equals' assignment token; It knows then the rest of the expression is the value node. The expression ends when it finds the end of statement token, in this case it is the semicolon token. The right hand-side is an expression in itself. The parser sees a literal token, it then creates a literal node, it checks the next token, and it sees it is a binary operator (plus sign +). It now expects either a literal or an expression. In this case, it finds another literal. It then creates an expression node containing the three nodes (two literal nodes and the operator node).

As of now, the parser peeks at the next token, since there is no other expression, the parser expects a semicolon token. In case the token is not found, the parser will return an error to the user, and if the token exists, it validates the grammar of the expression, finishes the construction of the AST by creating the assignment node with contains both the right and left sides before existing.

Now, we have a computer-friendly structure that can be traversed easily and validated by the second phase. Now the AST acting as the Intermediary Representation is passed to the validator.

2.4.2.4 Validator

The validator enforces comprehensive semantic rules, significantly enhancing the robustness and correctness of the parsed structure and preparing it for further compilation stages or execution.

The validator, also known as the semantic analyser, is the last component in the parser puzzle. it is the component of the second phase.

The validator gets its input data from the parser, it gets the IR. It traverses through the AST and does semantic analysis. Semantic analysis is the process of verifying the meaning and logical consistency of the language. Traversing the AST is the second pass, since the validator needs to read the program again, only this time instead of a

raw text input, it reads a computer friendly structure.

Semantic analysis consists of four main stages, with our previous example of `int res = 10 + 2;`:

- **Type Checking**: In most cases, especially in programming languages, the validator is responsible for checking and validating the types of variable, function, etc... In the example, we declared that the `res` variable is of integer type, let us suggest that the input had a mistake and used a decimal number in the expression. The validator here throws an error, suggesting converting the expression to a whole integer number or the `res` variable to a float type instead.
- **Scope & Declaration Checking**: The validator verifies that each identifier used declared before usage. If the variable was not defined, the validator will throw an error to the user.
- **Constraint Validation**: Languages have constraint rules, the validator is responsible for verifying and enforcing them. For example, an array must have a specific length, when accessing it, a user might use an invalid index, here the validator will throw an error suggesting the invalid indexing.
- **Logical Consistency**: Validators are responsible for finding logical inconsistencies in the program. For example, in a statically typed languages, the variable cannot change types. In the example of the following input:

  ```
  int res = 12;
  res = "hello";
  ```

  the variable `res` was reassigned to a string data type. In the suggested language, we know that the variable is of an integer type when it was declared. When the validator stumbles upon the second statement, it detects the invalid data type and throws an error to the user.

The validator traverses the whole AST top-to-bottom, starting at the Root Node or Program Start Node, until it reaches the last Node. At each step, it runs the four functionalities, looking for inconsistencies. If the program is validated, the validator fills a final structure to be used by other programs later.

2.5 Operating System Components

OS [25, 26] are one of the most important piece of software today. They are responsible for managing computer hardware resources by abstracting complex hardware system interactions, and they usually provide a visual interface for users and applications.

An OS responsibilities mainly tackle the critical system's hardware components including memory management, process scheduling, device interactions, exception handling, and file system management. In an embedded environment and robotics, the

operating system's efficiency and robustness is key to performance and responsiveness.

Creating an OS from scratch requires in-depth understanding of both hardware and software components and layers, and typically required fine-grained control over each component separately and how it interacts with the other components. Thus, choosing the right technologies and languages is key here. Usually, low-level languages are used for this. In this project, the OS part was created using Assembly [27], C [28] and C++ [29]. Assembly language provides the necessary low-level control over hardware for early stages. C, being a procedural programming language, is used extensively alongside Assembly for the core kernel development. Finally, C++, provides Object-Oriented programming [30], and is used for higher level kernel services, drivers, and components [31].

Understanding the need and the key components of an OS is fundamental for the rest of the project, thus in the next sections we will delve into the world of operating systems more, and we will start by the what and why of an OS.

2.5.1 What are & Why the need for OS?

Operating systems play a critical role in managing effectively the hardware resources of the system, ensuring security, reliability, and management of constrained resources. Operating systems also offer abstraction to the user applications and developers to ensure the focus is on the functionality rather than primitive direct hardware manipulation.

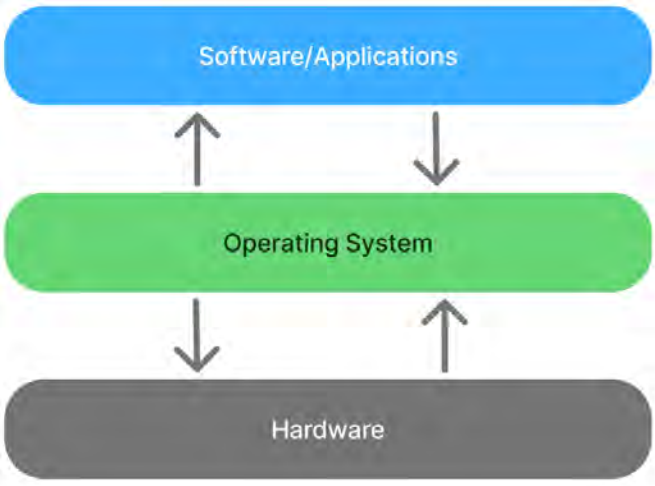

Figure 2.10: OS Position. Source: Own creation

Historically, operating systems evolved from simple control programs used on mainframe computers early on to a sophisticated system capable of handling multiple processes effectively. Is embedded systems, operating systems also provide real-time capabilities and abstraction ensuring minimal latency and predictable responses to external events, those are used mainly in robotics, automotive, and Internet of Things (IoT) devices. Figure 2.10 shows the position of an OS in a typical setup.

As can be seen in the figure, the OS is a middle man between software applications and actual device hardware to ensure the efficient use and management of these resources. But an Operating System isn't one piece of software, it is a combination of many components, each responsible for handling part of the hardware efficiently and intelligently. In the next section, we will explore the key components of the OS (which compose the kernel) and then each component in details.

2.5.2 Kernel and core OS components

Historically, OS and kernel are the same. Early Operating Systems were a kernel, some drivers and a simple text based interface such as a shell, sometimes called a CLI. Nowadays, due to much more powerful hardware, a modern OS usually features a Graphical User Interface (GUI).

The kernel, then, is the core building block of any operating system and is responsible for managing the system resources, ensuring and enforcing security rules, and providing necessary services to applications and processes. As mentioned earlier, kernels have a higher privilege than user applications which gives them unrestricted access to hardware, and as a result, kernel development needs precise understanding of low-level hardware components and knowledge in order to handle system resources precisely. Figure 2.11 shows some of the most important kernel and OS components.

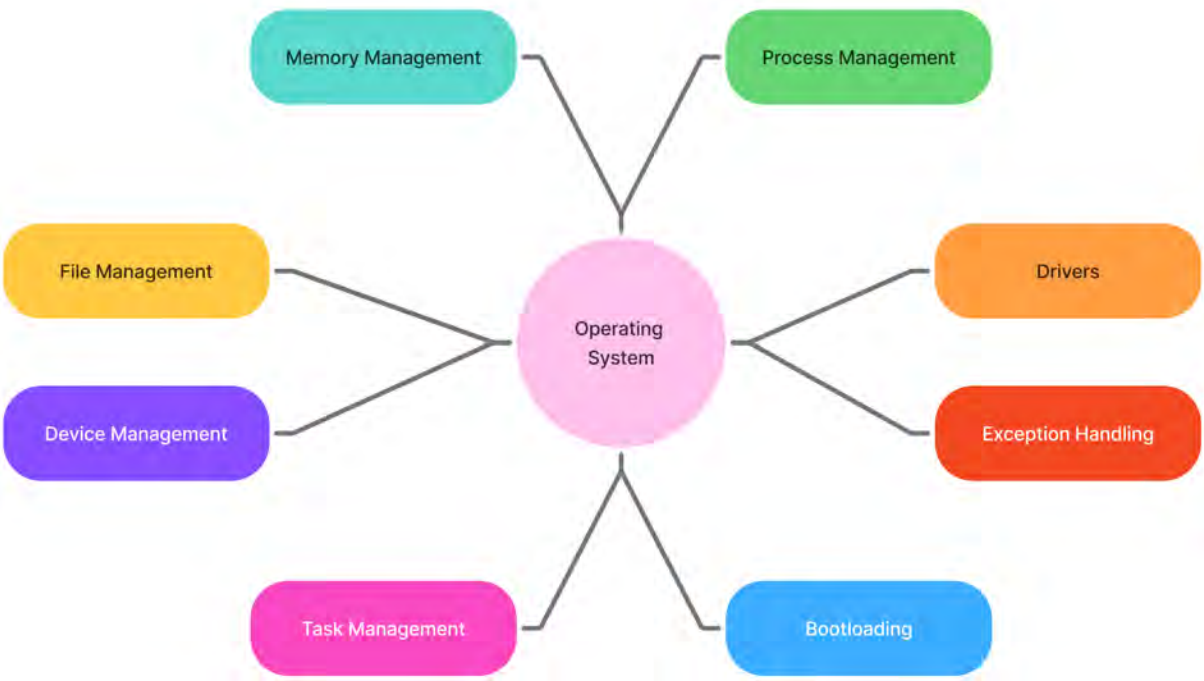

Figure 2.11: OS Core Components. Source: Own creation

Kernels might differ from one OS to another, and from one job to another, but they all share some features, as shown in the figure above:

- **Bootloading**: This is the entry point of any operating system, it usually resides at

a specific memory address by the vendor firmware. Its role is to set up the system in a known configuration and set up the runtime environment, then load the full kernel. Because it needs fine-grained control over hardware that is initialized, it is usually written in Assembly language and sets up the runtime environment for higher-level languages such as C.

- **Exception Handling**: Like any system with many interactions, faults or exceptions can happen at any point of time during the OS usage, either at software or hardware level. If not handled correctly, this can cause confusion and data loss, thus, kernels usually have exception handling mechanisms to revert to a known stable state and taking the corresponding action needed.
- **Drivers**: A kernel by itself offers most basic functionalities needed to operate, but it needs more features and modularity to work with diverse hardware components. Drivers are a piece of software that uses the kernel functions to extend the OS functionalities. An example can be a GPU driver, whilst an OS can run without a GPU, it might be necessary to use it in specific cases such as AI training or running graphics heavy software. To summarize, a driver extends the functionality of an OS kernel, but the OS can function without it.
- **Process Management**: A program, or user application, is just a process. A process, or a program in execution, is a sequence of instructions executed in a predefined order. A process is also a self-contained structure that keeps track of much important information regarding the process such as process state, memory allocations, files being used, I/O buffers, etc.
- **Task Management**: A process is a set of tasks. A task is the fundamental unit of work in an OS. As mentioned in the previous point, a process is a sequence of instructions. These instructions can be opening a file, running an algorithm, showing results to the user, etc. Each small set of instructions such as reading a file is a task. To understand let us look at a small process, its purpose is to read a file, read numbers from it, and displays the sum of them, this can be divided into 3 fundamental tasks: read a file is a task, extracting only numbers is another task, calculating the sum as an algorithm is the third task, and the final task is showing the results to the user. Each task is an independent unit, and together they form a process. A task can also be scheduled independently by the kernel, and this is how we achieve multitasking in operating systems. This is very useful, especially in multitasking and in multithreaded programs.
- **Memory Management**: One of the most critical parts of any kernel is the memory, especially the RAM. The RAM size can range from one system to another, it can be a few Megabytes to some Gigabytes. This memory contains the operating system, program data, processes data, etc.... Some peripherals also use mapped memory regions to access their I/O. Managing the memory resources is critical for smooth performance and avoiding hardware problems.
- **Device Management**: A system typically contains multiple devices that serve different purposes. Such devices can be a storage device such as a hard disk, a Solid State Drive (SSD) or an SD Card, can be devices that add functionality to the system such as an Ethernet port, a Network interface, or a Bluetooth interface. External devices are also managed by this system such as Universal Serial

Bus (USB) connections for input/output such as a screen, a mouse, a keyboard, or a camera. Handling these devices is critical to the functionality of the system, and the kernel is therefore responsible for it. Device management ensures clear communication, secure data handling and giving a clean high-level interface to the device to be used by other kernel components or user applications.

- **File Management**: Building on storage devices mentioned in the previous point, files are the main way data is stored and organized. Thus, kernels are responsible for handling the files and organizing the data flow. File management typically involves file operations such as creating, reading, writing and deleting files while ensuring data security and integrity. File management is also responsible for data access and file permissions.

After understanding the basic and core components of an operating system, we will discuss each part in details, starting with the bootloader.

2.5.3 Bootloader

When a system, or a computer, powers on, it looks for a small program called the bootloader, which is usually a few bytes or a few Kilobytes. The bootloader is the entry point of any operating system; its task is to set up the initial state of the Central Processing Unit (CPU) and configure the system to a known state before loading the main kernel of the OS. The bootloader is often written in Assembly as we do not have any runtime environment yet, and it gives us fine-grained control over the setup and initialization process.

The bootloader can vary in size and in functionality in different architectures, and we categorize them into two architectures:

- **x86 & x86_64**: The bootloader in the x86 architecture [32] has a fixed size of 512 bytes and must end with a hexadecimal value stored in the last 2 bytes. The hex value is: `0x55AA`. This fixed size is due mainly to the unified firmware called the Basic Input/Output System (BIOS) that is first loaded and looks for the hexadecimal signature to load the bootloader which sets up the system before loading the kernel from a storage device (e.g., hard disk, SSDs, SD-Card, etc. . .).
- **ARM & AArch64**: ARM architecture [33] is the fastest growing architecture in computing, especially in the last 5 years. Unlike x86, ARM is a standard and is implemented differently by various vendors. As a result, ARM systems do not rely on a unified firmware, which in turn caused bootloaders to not have either a fixed size or basic I/O functions like the ones provided by the BIOS for the x86 machine. Thus, in ARM systems, bootloaders need to have more functionalities written by the developer, which leads these small programs to range from a few Kilobytes to a few megabytes.

In robotics and embedded systems, battery usage and low power are one of the most important aspects of the system. In this area, ARM excels due to its reduced

instruction set following the reduced instruction set computer (RISC) principle. This is the main reason it was chosen for this project, and thus we will now focus on it, explaining in detail the boot process of ARM machines, specifically those that implement the ARMv8–ARM version 8–instruction set.

2.5.3.1 ARM bootloading

Figure 2.12 shows the general steps of an ARM bootloader. It starts when the device gets powered on, then it reads Read Only Memory (ROM) code from the vendor, typically written and packaged with the System on Chip (SoC) which is programmed to look into a specific address to load the bootloader and start the setup. The address is not known or general for ARM devices, it varies from vendor and SoC, the kernel developer gets it from the reference manual of that specific system.

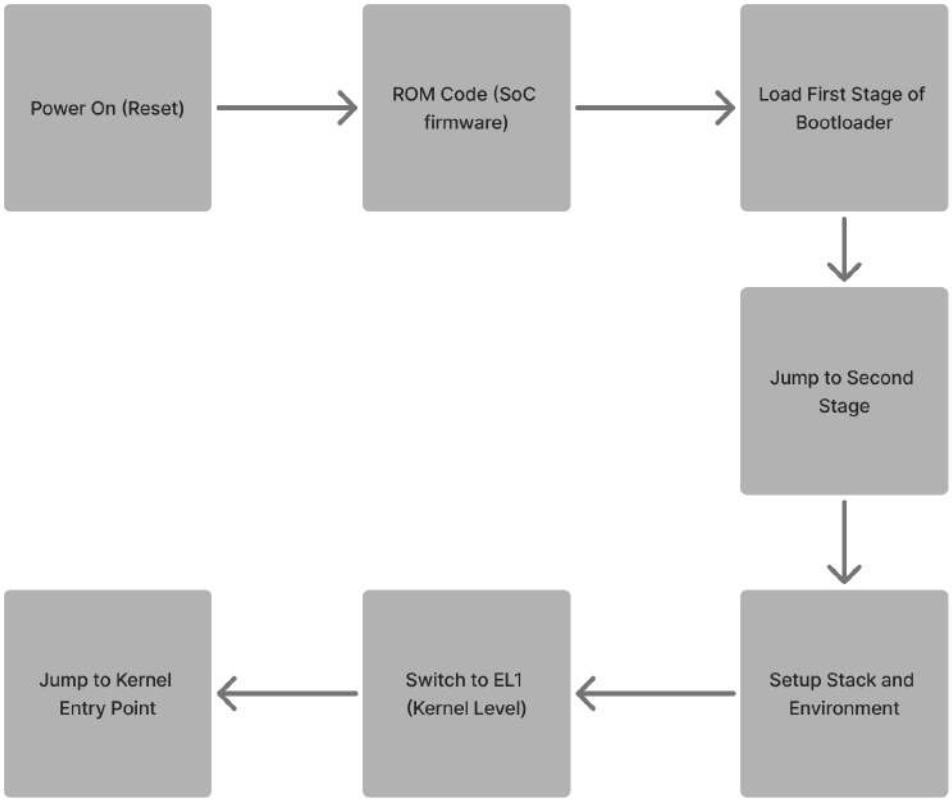

Figure 2.12: OS Boot Sequence in ARMv8. Source: Own creation

2.5.3.2 First-stage bootloader

Let us start by taking a look at the first stage. it is the entry point which is loaded by the initial firmware. Its purpose is to perform initial checks, set up early debugging via UART, initialize hardware and set the execution mode into the correct `EL` which stands for Exception Level. An Exception Level (EL) is used to define the privilege mode or

level of code execution. There exist four exception levels: EL0, EL1, EL2, and EL3, with higher numbers signifying higher privilege (see Figure 2.13).

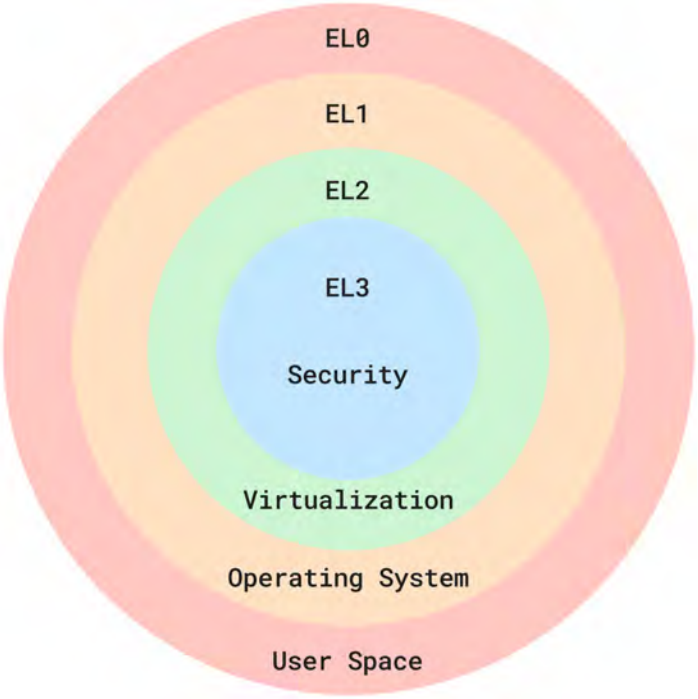

Figure 2.13: ARM Exception Levels. Source: Own creation

By the ARM standard pulled from the official documentation, the main components of software which are the Operating System and User application run on EL1 and EL0 respectively. EL2 is used for virtualization and EL3 is used for secure monitoring.

Knowing which level the OS and kernel should run on, we can commence the first-stage setup. First we need to check on which CPU core we are running on, the setup must be running on the primary core and halt the secondary ones. We can achieve it by using the `mpidr_el1` instruction, which gives us the core ID and we proceed accordingly.

Once we are running on the primary core, we disable all interrupts by masking them. This is a crucial step because in this critical phase, the process must not be interrupted under any circumstances. Now that we are sure that we will not be interrupted, we move into the stack initialization. This step sets a pointer to the stack memory in a predefined address and clears it, and then initializes the Universal Asynchronous Receiver/Transmitter (UART) interface for early kernel debugging and logging.

Next comes the Exception Level setup. As mentioned earlier, the kernel should run in EL1 according to the official documentation. Thus, we first need to check on which level we are by using the `CurrentEL` register. Most systems here are either on EL2 or EL1 by default depending on the vendor, but it is important to check for all levels and if we are in EL3 or EL2 and if the condition is true, we jump level by level until we are in EL1. This sets the privilege level for the kernel. The final step in setting up the hardware is setting the exception vector table pointer. This is where we tell the CPU where the different exception handler functions reside in the memory by setting

the `vbar_el1` register to point at a memory section which is exactly 2 Kilobytes in size. Next, we clear and disable any memory protection or memory management unit until we further set up in the kernel.

The final step in first-stage bootloaders is clearing a section for BSS data. BSS data section is a reserved space for uninitialized variables that we will later on use.

After all this setup, the first-stage is done. It had set up the hardware into a known state, it put us into the right privilege level, and we can now jump to the second-stage.

2.5.3.3 Second-stage bootloader

The second stage finishes some setup and sets the environment for the kernel. Its purpose is to set up a runtime, environment, set up system registers and boot information, and finally cleans the system registers and jumps to the kernel main function.

First, this stage ensures that Memory protection and memory management is off, and sets the vector table (exception function) register again to ensure it was not overwritten. After that, the minimal runtime environment is setup. The runtime environment is meant to be able to run higher-level code, such as C code.

Next, the second stage usually gets some system configuration information such as available RAM and architecture of the processor and puts them into a boot structure that can be passed to the kernel. Most kernels also feature a “Magic Code”, which is a hexadecimal string used to check that the kernel was indeed called from the bootloader and not being bypassed, it serves as a simple security measure.

Finally, the second stage clears the system registers by setting them to zero to avoid leftover values and then jumps to the main kernel entry point and passes the boot structure as an argument. The main kernel function is mostly written in C since now we have a runtime is now usable.

2.5.4 Universal Asynchronous Receiver/Transmitter

The UART is a key communication interface used mainly for debugging, sharing data, and interacting with other systems. In embedded systems, especially, UART provides a simple serial communication method that facilitates sending and receiving data between two devices.

In kernel development, UART is commonly used for early debugging and message transmission. UART is asynchronous, meaning it is not happening at a specific time interval.

Figure 2.14 shows an example of a two-way UART communication between a host machine and an embedded device.

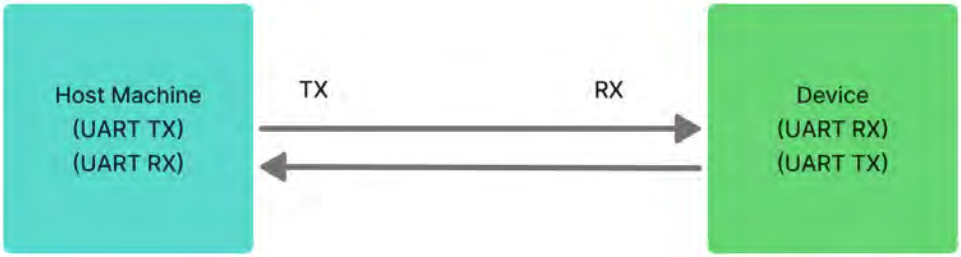

Figure 2.14: UART Communication. Source: Own creation

UART requires two connections, each side has a TX and an RX port. TX stands for the Transmitter and RX stands for the Receiver port. For communication, each RX on one machine connects to the TX of the other machine, thus achieving an efficient, reliable two-way communication.

2.5.5 Exceptions

Exceptions in Operating Systems represent an event that disrupts the normal execution flow of instructions, causing the CPU to change the flow to handle the exception event. These exceptions can be triggered by different conditions including undefined instructions, memory faults (accessing illegal memory space), or software system calls. Figure 2.15 demonstrates an example of a software exception triggered by a user program.

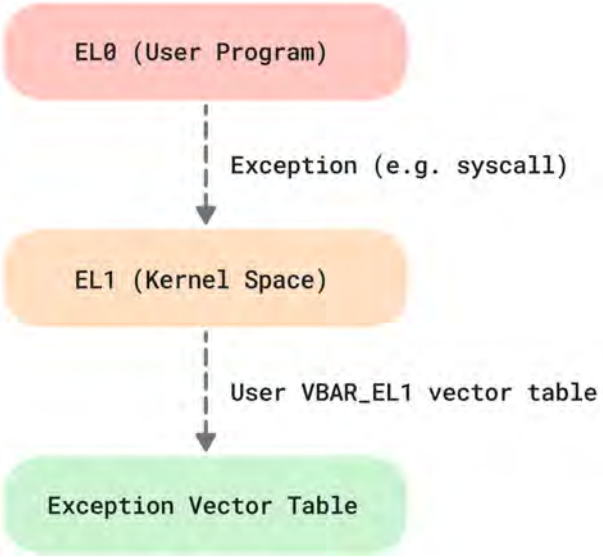

Figure 2.15: Exception Flow in ARM. Source: Own creation

In the bootloader chapter, we mentioned that the first stage sets up the `vbar_el1` register with the exception handlers table pointer. When an exception triggers, the kernel looks at the vector table for the correct handler.

Let us analyse the example provided by the figure in detail. We can see the three stages of an exception. First the exception is triggered from the user space (EL0) by

a system call (syscall) such as opening a file, the call goes to the kernel where it has privilege and uses the vector table to look up the correct function, the processor then is interrupted and handles the condition before returning to the normal execution flow.

2.5.6 Interrupts

Interrupts are signals that tell the processor to temporarily halt its current task and attend to a specific event or request. The signals can be triggered from software or hardware, but are concerned with devices attached to the host machine. Examples of devices that trigger interrupts are mice, keyboards, cameras, etc. . . .

In ARM-based systems, the kernel manages the interrupts via an interrupt controller, named the Generic Interrupt Controller (GIC). Figure 2.16 shows a normal flow of a hardware interrupt.

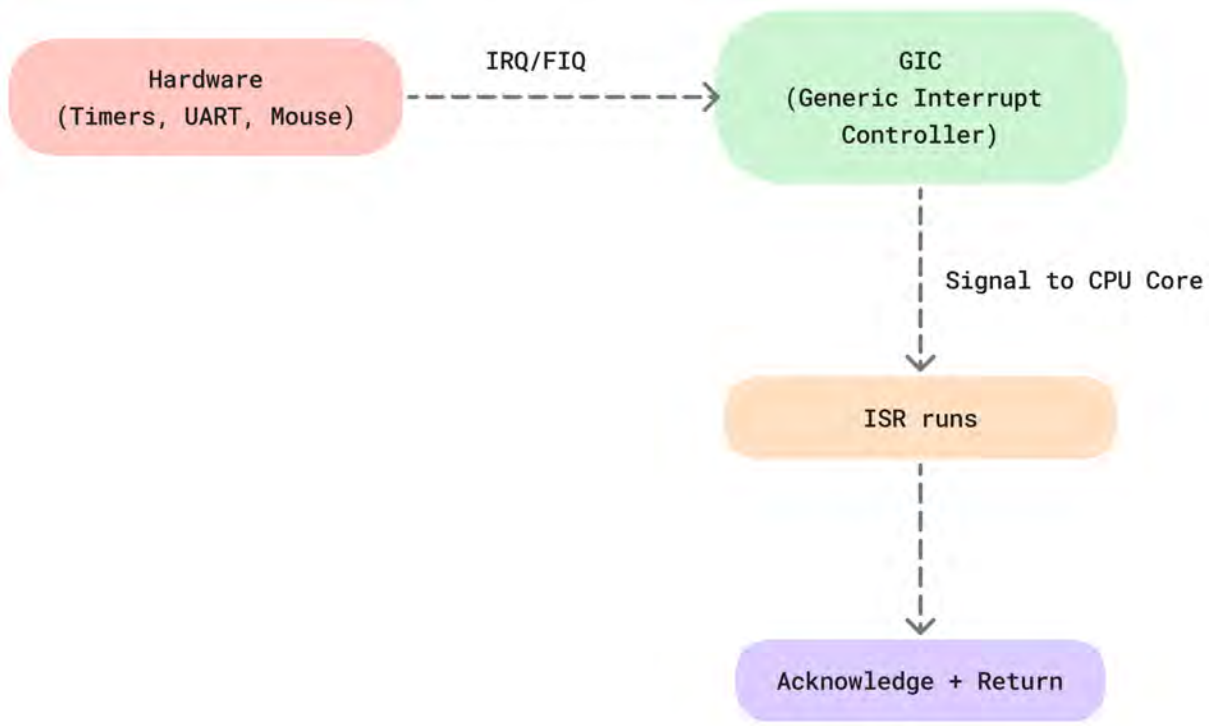

Figure 2.16: OS Interrupts. Source: Own creation

In a system, there exist multiple interrupt sources, they can be internal such as timers or external such as mouse movement or keyboard input. These systems trigger either an Interrupt ReQuest (IRQ) or a Fast Interrupt reQuest (FIQ) depending on the priority of the interrupt and its priority in the system. The signal goes to the GIC which is responsible for managing the registered interrupts and dispatching them to the processor cores. The GIC calls the correct Interrupt Service Routine (ISR) which contains the instructions to execute when a specific interrupt is requested. After running, the ISR acknowledges it run and returns. Figure 2.17 shows an execution timeline for an interrupt.

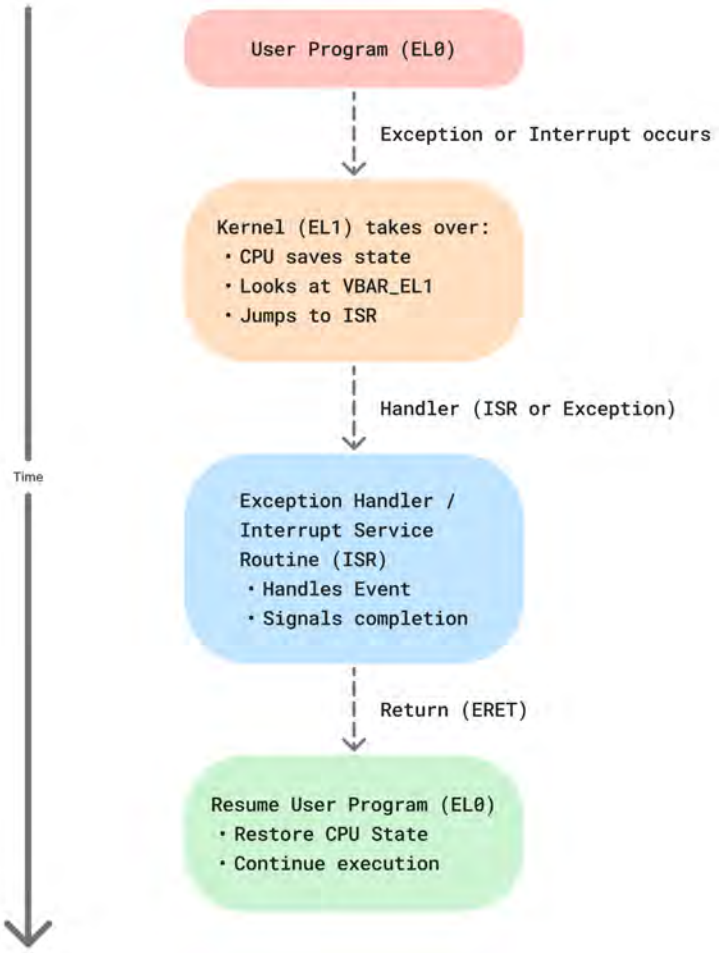

Figure 2.17: Interrupt Execution Flow. Source: Own creation

Let us take the example of a user program wanting to read from the keyboard input and walk through it. First, a user program, running in EL0, causes an interrupt via the kernel interrupt system such as a keyboard input interrupt. The request is forwarded to the kernel in EL1 with privileges. It first saves the current execution state so it can restore it later and continue execution from where the user program left, then it looks up the ISR handler in the vector table using the `vbar_el1` register and calls the handler. The handler does its execution to handle the event, such as reading the character from the keyboard and converting it to ASCII, and then it signals its completion to the caller. Finally, after returning using the `eret` command, the processor restores its previous saved state and continues normal execution.

2.5.7 Timers

Timers are one of the most important components of the operating system. It is it used for scheduling tasks and processes, handling timeouts, and generating periodic interrupts. ARM-based systems rely on a physical hardware timer to ensure precise timing and synchronization. The timer is crucial for real-time responsiveness and deterministic behaviour. Figure 2.18 shows how a timer works inside a kernel.

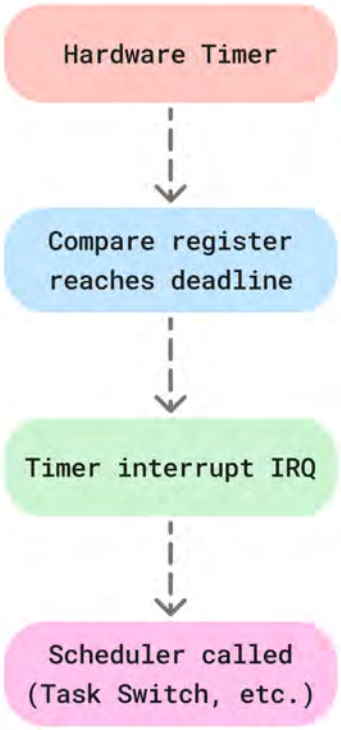

Figure 2.18: OS Timers. Source: Own creation

First is the hardware (physical) timer, which is responsible for generating signals at a predefined interval. Then, a register inside the system is responsible for intervals. This is a programmable register to set a predefined interrupt frequency wanted by the OS component. The register serves as a countdown of the signals gotten from the hardware timer. When the register reaches 0 (the deadline), the timer interrupt function is invoked and each function listening to the interrupt will run in turn inside the scheduler. The scheduler is usually responsible for managing tasks to allow multitasking inside the OS but can also call other functions depending on the needs such as periodically checking memory etc. . . .

2.5.8 Memory management

In OS, memory management is a critical part of hardware resources management. This process involves managing the system's memory resources, allocating memory, freeing it, and handling I/O mapping.

The memory management system is a complex system that handles different memory types handling from the user and provides a unified interface for interacting with these subsystems. In embedded systems, the memory resources can be of these types: Static RAM (SRAM), Dynamic RAM (DRAM), and other devices. The memory manager unifies them into a single contiguous memory address space.

Figure 2.19 shows an example of a typical memory layout in an operating system. Let us explore it in details to understand how the memory is managed.

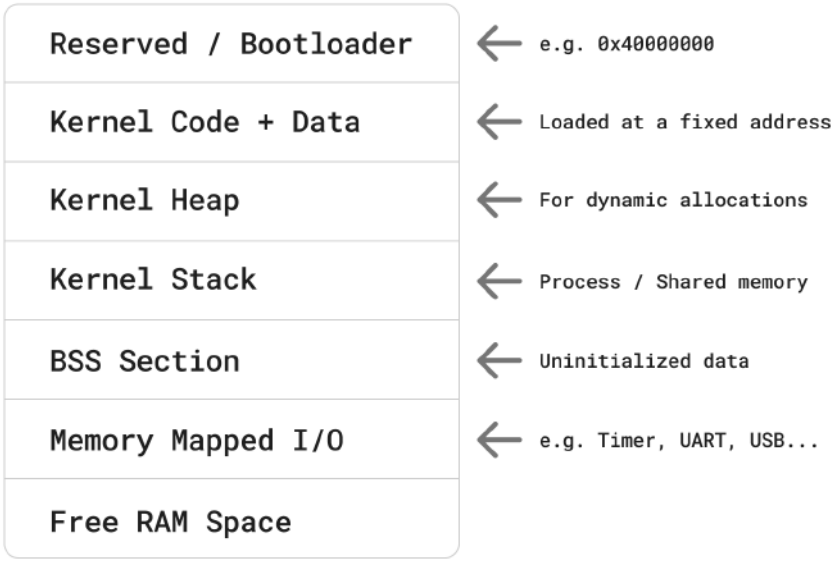

Figure 2.19: OS Memory Layout. Source: Own creation

We can see that the system divides the resources into seven sections:

- **Reserved / Bootloader Space**: This is usually a small section in the memory region used to start the boot process or other reserved code space. It has a specific address dependent on the machine hardware, since the computers looks at that address for the bootloader.
- **Kernel Code + Data Space**: After the bootloader finishes the initial hardware setup, it loads the full kernel code, since the size is already known, a section of the memory is reserved for the kernel code to load at a fixed address.
- **Kernel Heap Space**: The OS leaves a space for the Heap memory. This memory usually gets 25 percent of the available RAM resources. The heap is used for dynamic allocations using the standard library functions such as `malloc`, `free`, and `realloc`. This memory is allocated during program runtime and managed by the programmer, usually.
- **Kernel Stack Space**: The kernel stack is used for the processes' memory such as local variables, return addresses, and function call arguments. Each process has its own private kernel stack during the execution.
- **BSS Space**: This memory region stores global and static variables that are either declared but not initialized or those who are explicitly initialized to zero.
- **Memory Mapped I/O Space**: In a system, many devices needs to communicate and transfer data. These are usually memory mapped I/O, meaning that in order to access the data ports on these devices, the OS gives them a specific address space. Example of such devices include UART, USB, and hardware Timers.
- **Free RAM Space**: This memory space is flexible, meaning it can shrink or grow as needed. This is usually left unused for dynamic kernel services that can be loaded or unloaded as needed.

2.5.9 Process & Task management

Process and Task management are responsible for creating, managing, and scheduling processes inside the operating system. This system must be efficient to ensure optimal resource usage, enable multitasking, and provide isolation between applications. This is necessary to ensure system stability and responsiveness.

A process encapsulates executable code, private data, and operating system structure to keep track of information related to this process such as the process ID, the tasks list related to this process, open files, and signal handler.

Figure 2.20 shows the life cycle of a process from when it is created until it is terminated.

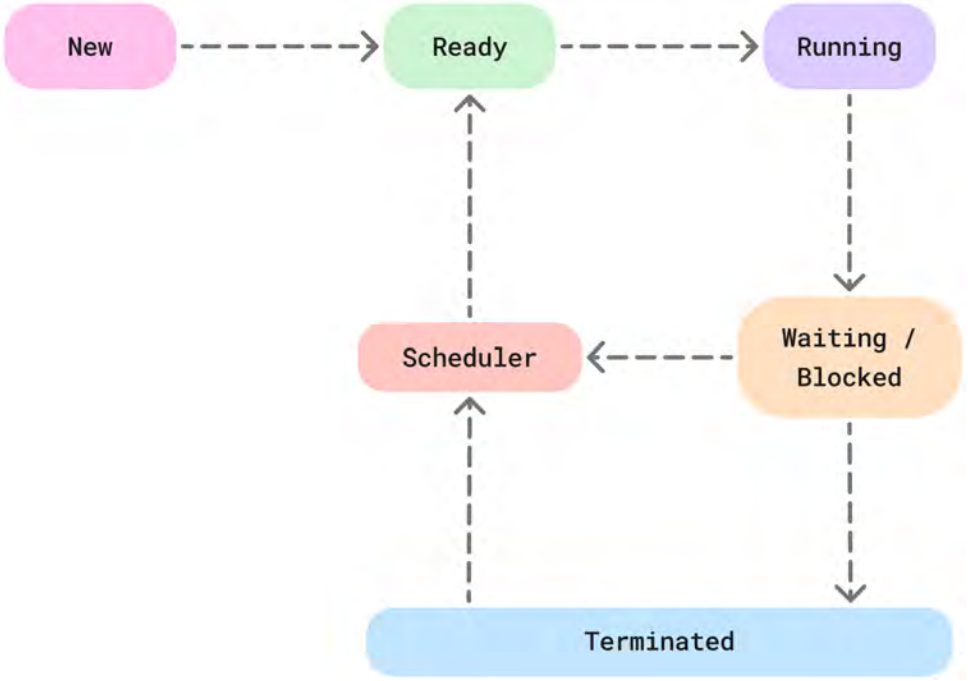

Figure 2.20: OS Process Life-cycle. Source: Own creation

When a new process is created and initialized, its state is transitioned into the `Ready` state waiting to be run. When the scheduler gets to this process, it transitions it into the `Running` state, now the process is executing. At any moment, a process is either interrupted by the scheduler or an interrupt. In that case, it transitions into the `Waiting / Blocked` state. From that state, it tells the scheduler that this process is no longer running, and the scheduler looks for the next process to put into the `Ready` or `Running` state. When the process is unblocked or finished waiting, it either returns to the `Ready` state or `Terminated` state. In the case of termination, the scheduler is notified and start looking for the next process.

A task is the fundamental unit of any process. A process is one or more tasks, each having their own small task. In a case of a parser where it needs to read file, create new files and running the algorithms, it can be divided into three tasks. The first task is responsible for opening the file, the second task is responsible for creating a new file,

the final task would be the parser algorithm.

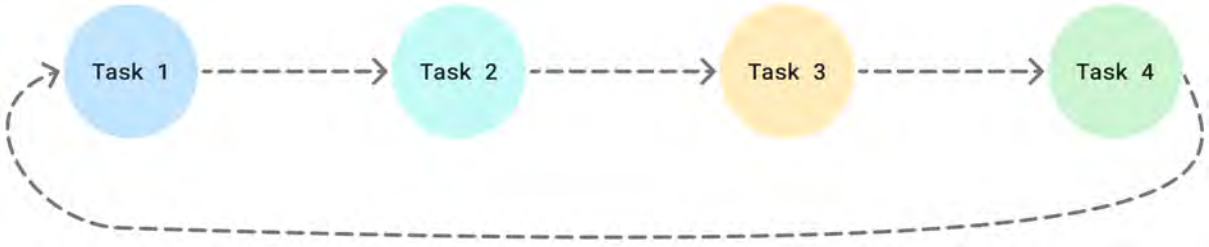

Figure 2.21: Process Task List. Source: Own creation

Figure 2.21 shows a task list of a simple process. Tasks are usually linked lists, with the first task registered as the main task. The process structure holds a pointer to the initial task and can traverse them as needed, returning to the first if needed.

2.5.10 Disk Management

In an operating system, storage is a key component. RAM and different memories cannot hold long term data, they lose all the data when the power shuts down. Thus, a permanent storage device is needed to store the data, including the Operating System itself.

There exists different persistent storage devices that serve different purposes ranging from small fast memory that holds the boot code to a different hard disk such as Hard Disk Drives (HDDs), SSDs, and Non-Volatile Memory Express (NVMe). Since a system can, and usually, have more than one storage device, a disk management system is necessary to ensure reliable and secure data storage by providing a clean interface.

A Disk management system have many responsibilities including data storage and retrieval, partitioning, and securing data. The system can interface with the different serial interfaces of the storage device hardware and implements these function for each type and internally manages the specific implementation of these functions. Such function are `open`, `close`, `delete`, `create`.

These functions create a stable, reliable foundation for the File Systems, which we will explore next.

2.5.11 File Systems & Virtual File Systems

File systems organize and manage data on storage devices, providing File Systems organize and manage data on permanent storage devices by providing a structured method for managing, storing, and retrieving files and directories. The File System is a standard, and a device can have only one file system at once. This is important as

each system have different standards, and it needs to handle critical operations such as file creation, file deletion, and data integrity.

When a disk is partitioned, the operating system registers each partition as a separate storage device, thus each can have its own file system.

A system that has more than one virtual storage device needs to handle multiple file systems at one, this is the role of the Virtual File System (VFS), which provides a unified interface functions that each file system implements on it its own and gets assigned to a storage device which hides the specificities for the user.

Figure 2.22 shows how a virtual file system handles multiple devices at one.

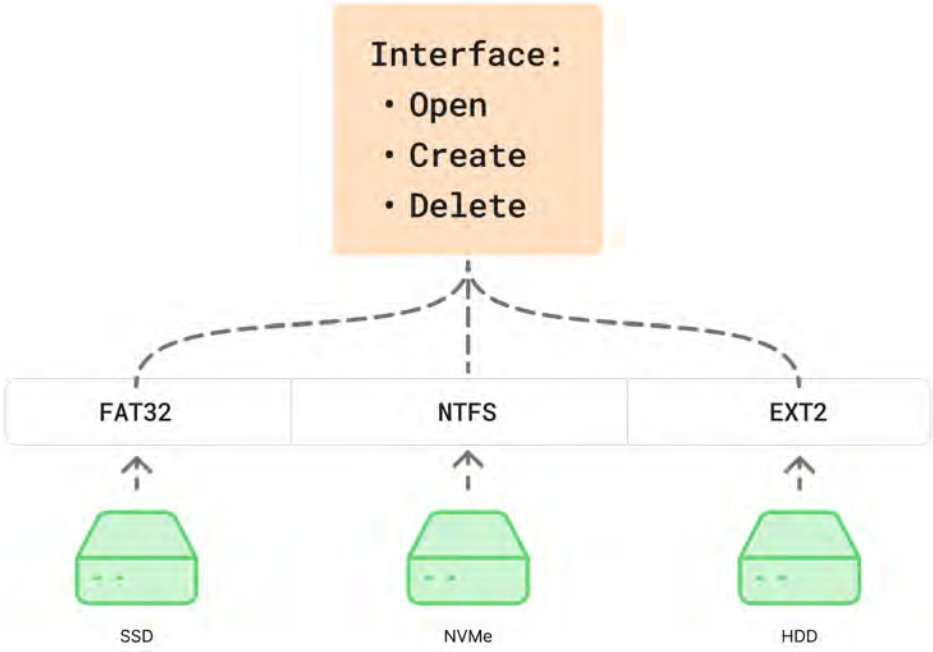

Figure 2.22: Virtual File System. Source: Own creation

When a new device is added, it gets assigned an ID and a slot in the virtual file system list. The operating system assigns a slot to a virtual disk. A virtual disk can be the full physical disk or a partition from a physical disk. In the example above, we see the user has three disks attached physically to his physical hardware, with the help of the Virtual File System, each device has one partition initially as is assigned a slot. The slot contains the device File System and is managed by the VFS when it is first added. This enables the operating system to handle multiple File Systems simultaneously and enables the user to swap disk devices between multiple systems of operating systems.

The VFS exports an interface containing the important functions to operate on the storage device. Internally, each File System implements the same interface functions to work with their internal structures.

By now, the operating system implements most of the core components, and we have a fully functional base kernel. But, it is still missing the last component in order to be usable. This component is the user space programs. By implementing those, we

will have a base operational, usable OS.

2.5.12 User Space, Programs & CLI

User space and user programs are application and services running on the lowest privilege mode. In the ARM exception levels, referenced in Figure 2.13, EL0 has the lowest privilege and this is where the user space lives. This lower privilege ensures security and ease of control of the user space. In order to interact with the hardware (e.g., get keyboard input, allocate memory, etc...), the kernel exposes the `stdlib` (standard library) to use with languages and runtime (e.g., C, C++, etc...) programmatically. The user space can also interact with the kernel services using system calls (interrupt).

Kernels and early OS did not include a full GUI like the latest ones. Instead, they featured a simple User Interface (UI) called the CLI or Shell. The Command Line Interface (CLI) enabled users to interact directly with the operating system using textual commands. The CLI internally executes user space programs, these programs have many roles including controlling ad configuring the system, monitoring the system health, file manipulation, and program execution.

The CLI, then, is a crucial part of operating systems especially in embedded systems which have very limited resources and thus are unable to run full GUIs, and links the operating system kernel and the standard libraries together enabling user interaction with the system.

2.6 Conclusion

This chapter has provided a conceptual study of three main domains. First is the state-of-art inside the ML filed where we took a look about ML, linear algebra and how an ML model works internally. We focused on regression and specifically on Linear Regression, which lets us predict continuous values. Second, we took a look about compilers and parsers, how they work and why do we need them. Finally, we took a look at operating systems, their components and how they function. These components, as far as they might seem from one another, are interconnected to form a complete system.

The operating system provides a CLI that the user interacts with, the CLI in turn uses the OS kernel to manage and interact with the hardware. From the CLI we can call an AI Engine that enables the user to train and run AI models that uses the kernel services to operate (e.g., allocate memory, creating and reading files, output data to user, etc...). The AI engine itself makes use of parser and compiler theory to process input files such as CSV data file, configuration files and output files to create a full system.

In the next chapter, we will move into the implementation of the system, and there we will see how the components interact with each other in detail.

Chapter 3

PROJECT DEVELOPMENT

3.1 Introduction

With the rise of AI in the last years and the continuous improvements in the field, a new trend emerged: the use of AI in many fields including automation, security, and robotics. Now, we have machines more intelligent, such as autonomous robots used in warehouses, self-driving vehicles, and much more.

Training and running AI models requires a huge amount of resources and time, mainly due to the development environment and tools. Tools such as `Tensorflow` and `Pytorch` are made to run on servers and desktop machines, which have great resources that embedded systems often lack. These tools also usually rely on higher level languages, need external tools and libraries, and require complex setup. Whilst these tools are tested and industry standard, they usually perform poorly on embedded systems or require a secondary setup to run the AI and ML models on the target hardware.

The Synapse Robotics project emerges from these problems and the clear need for a performant, modular, portable, and optimized AI workflows for embedded and resource-constrained hardware. In this chapter, we will take a look at the development phase of the Synapse Robotics project with its different components, highlighting the importance and the role the project plays.

3.2 Project Description

The `Synapse Robotics` project is a new initiative aimed to ease and lower the entry barrier for hobbyists, embedded developers, and students. The project is divided into two separate components: the *Synapse Engine* and the *Synapse Kernel*. The two-component project aims to address most of the users. By having this separation, the project achieves modularity and portability. The users are able to choose between

using the Synapse Engine by itself or with the Synapse Kernel to have the full performance enhancement.

Synapse Engine is a high-performance, lightweight, optimized AI and ML training and inference framework. The framework was designed specially for embedded and resource-constrained systems. The engine provides the whole end-to-end pipeline which includes data processing, model definitions through a custom developed DSL, training, and prediction capabilities. The engine was developed using embedded languages mainly `Zig` [34], `C++` [29], and `Rust` [35] to ensure high efficiency and cross-platform compatibility. The engine uses optimization for ARM-based systems, a common architecture in embedded systems.

`Synapse Kernel` is a custom lightweight operating system kernel developed to complement the Synapse Engine by providing a minimal yet efficient runtime environment tailored for embedded robotics applications. It manages low-level system resources such as task scheduling, memory management, and hardware abstraction, enabling real-time execution of AI workloads with low latency. The kernel is designed to be modular and configurable to support various embedded platforms and to facilitate seamless integration of the AI engine. By combining the kernel with the engine, users can deploy a full-stack AI solution optimized for performance, reliability, and real-time control on embedded robotic systems.

Synapse Kernel is a custom operating system kernel developed from scratch to complement Synapse Engine by providing OS level AI optimizations. The kernel manages low-level system resources such as process-management, memory management, and hardware abstraction enabling real-time execution which is needed for real-time embedded applications. The kernel is designed to be modular and configurable in order to support various platforms. By combining the kernel with the AI engine, users have a reliable AI workflow optimized for resource constrained-resources systems, providing optimized performance, reliability, and real-time control for robotics systems.

Together, both Synapse Engine and Synapse Kernel provide a flexible, reliable, and powerful framework for AI workflows tailored for embedded systems. The combination enables developers to build, train, and deploy ML models efficiently in a constrained environment.

3.3 Analysis

This section provides a detailed analysis of the Synapse Robotics project by focusing on the key functional aspects and interactions between users and the system components. The analysis emphasizes how users engage with the system through various workflows involving training and prediction processes, as well as the overall engine interface. Understanding these interactions is crucial for ensuring the design aligns with user requirements and system capabilities, and it informs subsequent implementation phases.

3.3.1 Requirement Specification

To ensure the Synapse Robotics project met its goal, a set list of functional and non-functional requirements was defined during the design, implementation, and reiteration phases. These requirements will serve as the validation points for the final implementation of the project.

3.3.1.1 Functional Requirements

Functional requirements describe the essential actions, tasks, or behaviours that the system must perform. The compiled set of requirements consist of:

- CLI for user interaction.
- Robust Data parsing for configuration and data pipelines (CSV, JSON, and SYNJ).
- DSL for flexible and reproducible model configuration.
- ML algorithms implementation (Linear and Multi-Linear Regression).
- Custom ARM-based kernel optimized for AI.

3.3.1.2 Non-Functional Requirements

Non-Functional requirements define the desired quality attributes and constraints of the system. These requirements define how a system should behave. The following set of non-functional requirements have been chosen for this project:

- Maintainability and clear documentation.
- Portability across diverse systems and hardware architectures.
- Ease of use of the CLI tool.
- Custom Logging for better CLI
- High-performance with minimal memory footprint.
- Ease of setup and usability.

3.3.2 Use Cases

Use case diagrams illustrate the interactions between the end user, referred to as the "User", and the Synapse Engine. These diagrams highlight the main workflows supported by the system, including the use of the command line interface, training ML models, and running inference on them. Each use case diagram portrays a distinct aspect of system operation by providing a clear view of the system's core functionality.

Figure 3.1 shows the general use case diagram for the Synapse Engine. The User interacts with the system primarily through the CLI, which serves as the entry point for various activities. From the CLI, the User can extend the functionality to train models, run inference on models, configure model parameters, and dump model data and parameters. This diagram emphasizes the modularity of the system's design and the flexibility offered to the user for different AI workflows.

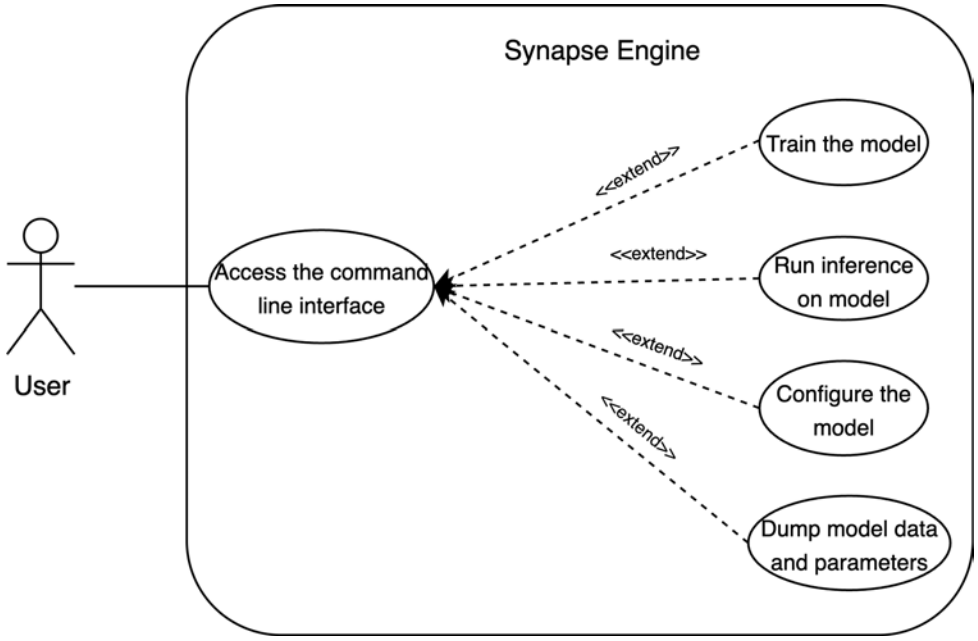

Figure 3.1: General Use Case. Source: Own creation

Use Cases and Use Case Narratives

The following use cases describe the specific scenarios and requirements from the perspective of the primary user interacting with the Synapse Engine. Each use case is accompanied by its corresponding use case diagram for clarity.

Use Case 1: Accessing the CLI and General Operations **ID:** UC-01

Actor(s):

- Primary: User
- Secondary: Synapse Engine CLI

Description/Goal: Enable the User to access the CLI of the Synapse Engine, which provides the foundation for executing core workflows such as training models, running inference, configuring models, and dumping model data.

Use Case 2: Training Process

Figure 3.2 illustrates the training process, which allows the User to provide training data and configuration files to the Synapse Engine via the CLI. This workflow enables the initiation of model training, validation of input, and saving of the trained model for later use.

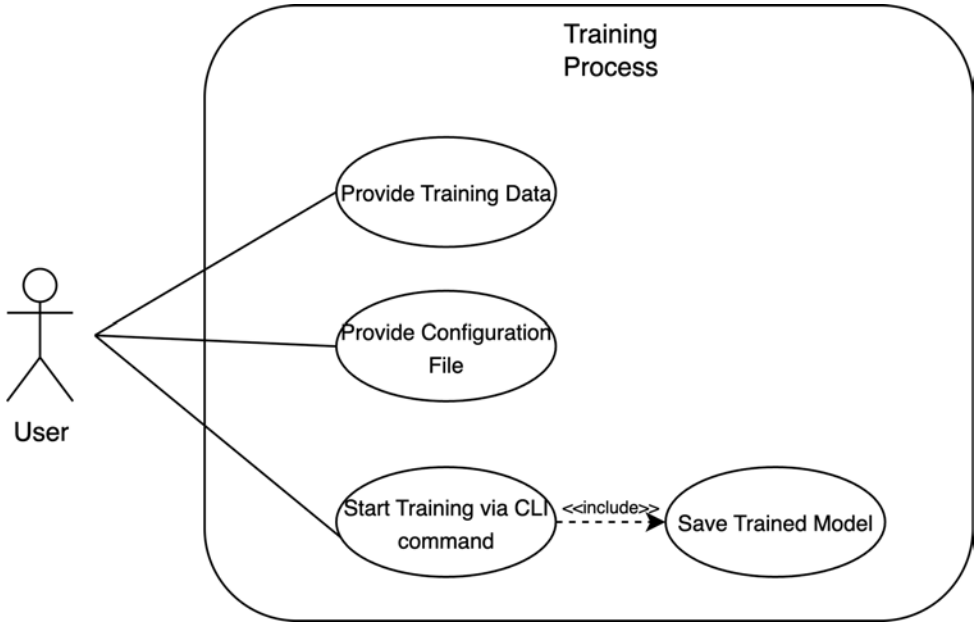

Figure 3.2: Training Use Case. Source: Own creation

ID: UC-02

Actor(s):

- Primary: User
- Secondary: Synapse Engine Training Module

Description/Goal: Enable the User to provide training data and configuration files, then initiate model training via the CLI, saving the resulting trained model for future use.

Preconditions:

- Training data is available and formatted correctly.
- Configuration file specifying model parameters is prepared.
- CLI training command is available and functional.

Postconditions/Success Guarantee:

- Model training completes successfully.
- Trained model is saved to the specified location.
- User is notified of training completion and model availability.

Main Flow:

1. User provides training data to the system.
2. User provides a configuration file specifying training parameters.
3. User issues the training start command via the CLI.
4. System validates inputs and configuration.
5. System executes the training process on the provided data.
6. Upon completion, the system saves the trained model.
7. System informs User of successful training and model storage.

Alternative Flows:

- AF1 - Training data invalid or inaccessible: System reports error and requests corrected data.
- AF2 - Configuration file invalid: System reports error and requests corrected configuration.
- AF3 - Training process interrupted or fails: System reports failure and logs error details.

Use Case 3: Prediction Process

Figure 3.3 illustrates the prediction process, which permits the User to run inference on input data using a previously trained AI model. The User interacts via the CLI to load the model, run the prediction, and receive output results.

ID: UC-03

Actor(s):

- Primary: User
- Secondary: Synapse Engine Prediction Module

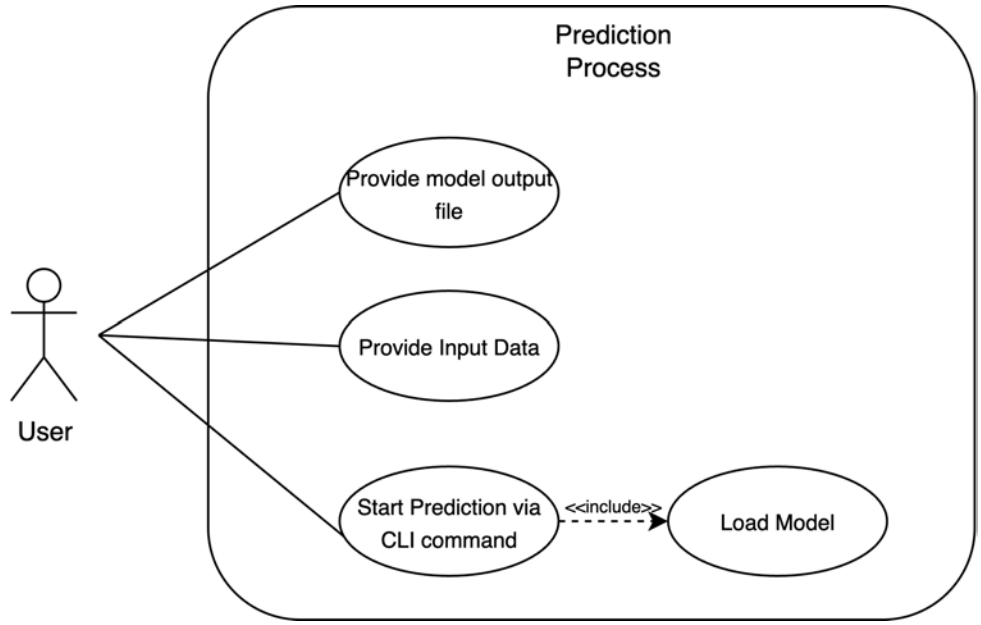

Figure 3.3: Prediction Use Case. Source: Own creation

Description/Goal: Allow the User to provide input data and model output files to run inference (prediction) using a trained AI model via the CLI.

Preconditions:

- Trained AI model is available and accessible.
- Input data is correctly formatted.
- CLI prediction command is available and functional.

Postconditions/Success Guarantee:

- Model loads successfully.
- Inference runs on the input data.
- Prediction results are output to the specified file.

Main Flow:

1. User provides the input data and model files to the system.
2. User issues the prediction start command via the CLI.
3. System loads the trained AI model.
4. System runs inference on the input data.
5. System writes prediction results to the CLI.

6. System informs User of successful completion.

Alternative Flows:

- AF1 - Model file not found or corrupted: System reports error and aborts prediction.
- AF2 - Input data format invalid: System reports error and requests corrected input.

3.4 Design

In this section, we will take a look at the architecture and design decisions that went into the project. We will look at the overall architecture of both the `Synapse Engine` and `Synapse Kernel` and understand the basic components.

3.4.1 System Overview

The `Synapse Engine` and `Synapse Kernel` projects are individual yet complementary projects that form the `Synapse Robotics` project. This design allow flexibility, modularity, and optimal performance in the development environment. This design choice enables the users to either use the optimized engine by itself on major platforms such as Windows, Linux, and macOS, or bundled with the kernel designed to run directly on resource-constrained systems suited best for embedded projects or educational setups.

We will first see how both systems behave separately or bundled together, then we will see the design choices for each in details.

3.4.2 Global Architecture

Figure 3.4 demonstrates the flexible setup. With the *Synapse Engine* being an independent user program, it gives the user the choice of using any supported operating system and architecture. The figure shows the engine being able to run on either a standard operating system (e.g., macOS, Linux, Windows, etc. . .) or on top of the custom *Synapse Kernel*. The kernel was developed from scratch for ARM-based systems.

3.4.3 AI Engine Design

Figure 3.5 illustrates the high-level internal architecture of Synapse Engine. The architecture emphasizes modularity, flexibility, and easy of maintenance.

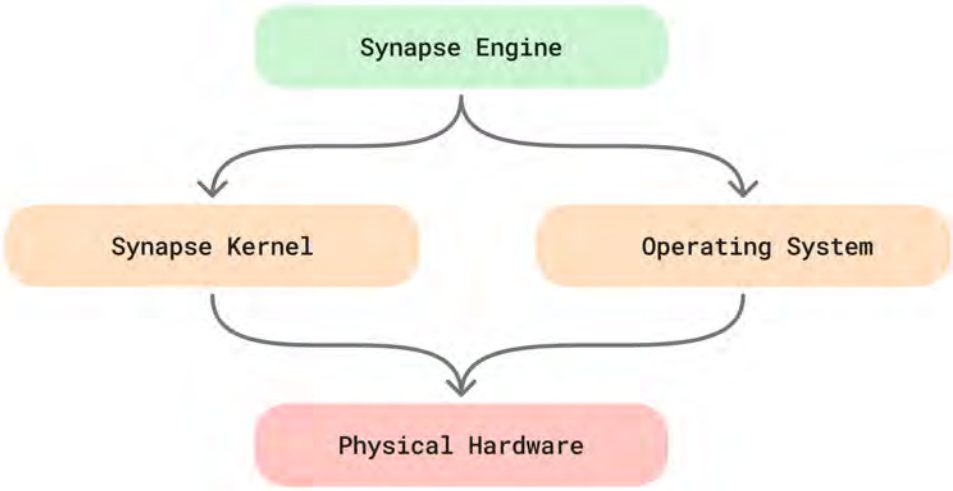

Figure 3.4: Global System Architecture. Source: Own creation

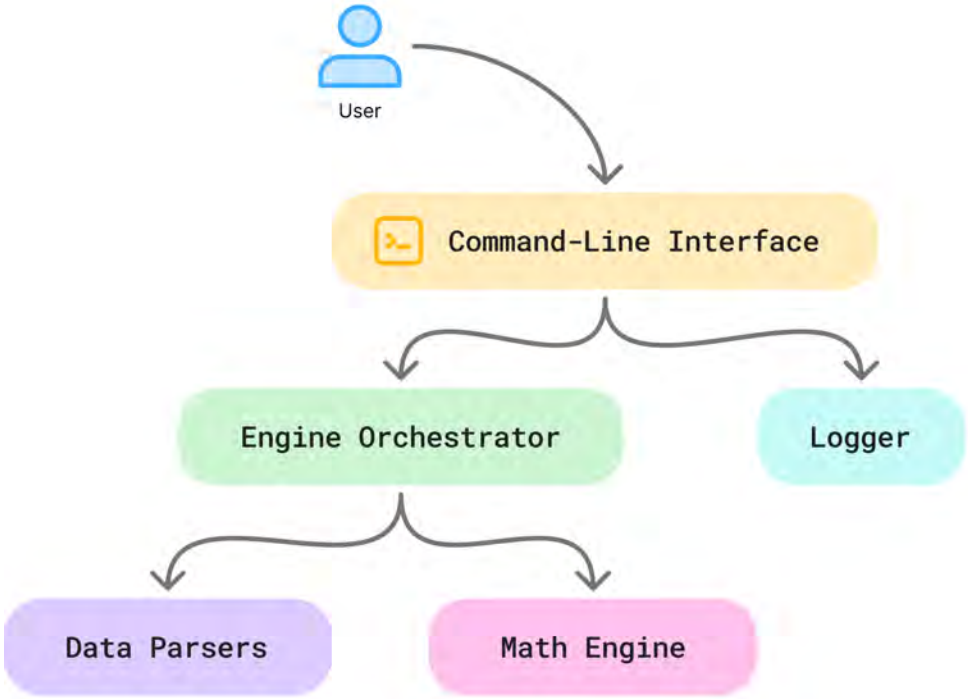

Figure 3.5: AI Engine Architecture. Source: Own creation

The engine is composed of five different layers:

- **CLI**: This layer presents the user interaction point, offering intuitive commands for managing ML models.
- **Engine Orchestrator**: The orchestrator is the linking component. It servers as the central hub by using the commands, it manages the overall workflow and calling the correct modules to finish a task.
- **Logger**: The logger provides structured, coloured output for the CLI to enhance user experience and simplify monitoring.

- **Data Parsers**: The data parsers are a set of 3 parsers for the CSV input data, the JSON data, and the DSL input. The parsers are implemented to deliver speed and safety. These output a structured format given to the orchestrator to handle them according to their data.

- **Math Engine**: The Math Engine is the most critical part of the AI Engine, it handles the math behind the models and uses hardware level optimization for faster processing. The implementation of this module focuses on memory safety and high-performance execution required by AI and ML algorithms.

Each component within the system holds a purpose designed for a specialized task. Together, they ensure an efficient, secure, and user-friendly system.

3.4.4 Kernel Design

Figure 3.6 portrays the different high-level layers of the Synapse kernel.

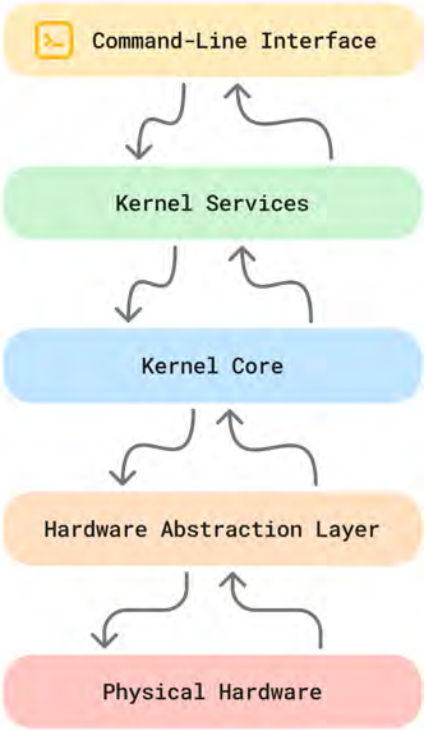

Figure 3.6: ARM Kernel Architecture. Source: Own creation

The kernel system is divided into five key layers, each serves a different purpose and interacts with the other layers. These layers are:

- **Command-Line Interface**: This is the highest-level component. It is a user program and provides the primary point for the user interaction.

- **Kernel Services**: The services are high)level OS components that provide abstractions from the core kernel components. These components include File Systems, Disk management abstraction and many more.
- **Kernel Core**: The core is the central component for the kernel. It implements low-level–architecture independent–mechanisms that provide the main subsystems. These subsystems are responsible for the critical behaviour of the system. Such subsystems include memory management, scheduling handling, interrupt handling, etc. . . .
- **Hardware Abstraction Layer**: The Hardware Abstraction Layer (HAL) provides universal access to physical hardware systems, its task is to isolate hardware-specific details such as General-Purpose Input/Output (GPIO), UART, timers and exposes a standard Application Binary Interface (ABI) to the kernel higher levels.
- **Physical Hardware**: The actual hardware the system runs on. For this project, an ARM-based SoC was chosen. The physical hardware also includes the RAM, peripherals, etc. . . .

In the Figure above, each component have a bidirectional arrow. These arrow indicates the communication flows between the different layers. Each service request services from the layer below and it usually returns data to the layer above. This ensures clear communication and components integrations while keeping modularity.

By using this architecture, the kernel achieves modularity, performance, and easy of extensibility, making it suitable for embedded systems and robotics platforms.

3.5 Implementation of a DSL & Configuration Processing Phase

3.5.1 Principles

Most AI and ML frameworks historically required users to learn programming languages alongside the framework. But in recent years, many efforts are made to lower this barrier that requires the user to become familiar with different technologies. They achieved this by creating new ways, such as creating UIs (e.g., Google's teachable Machine) or using configuration files such as YAML. But there was also the development of DSLs.

To achieve the goal of the *Synapse Engine* which ensures the ease of use and reproducibility, a DSL was developed to serve the purpose. The DSL was named `SYNJ`. The focus of this language is clarity, ease of use, and reproducibility. Thus, the language significantly lowers the barrier for newcomers, students, and researchers. It also provides a simple way to reproduce the same model between different systems and a fast way for researchers to create AI and ML models for experiments.

To ensure the DSL achieves its purpose, the following set of guidelines have been compiled in the design and implementation:

- **Minimalism**: Simple syntax with no complex data input, and only essential parameters are required. This reduces complexity and gives the user flexibility.
- **Flexibility**: The language must be easy but flexible to allow it to grow if needed for more complex models.
- **Reproducibility**: The configuration file should be self-contained to ensure consistent results across multiple runs.

3.5.2 Used Technologies

The DSL is a critical component, thus the technology used to implement the DSL parser needs to be safe, fast, and reliable. Thus, the `Zig programming language` was chosen to implement the parser because of its simplicity, high performance, ease of interoperability with C, and explicit error handling.

3.5.3 DSL Design

SYNJ syntax is easy and was inspired from JSON, YAML, and simple programming languages. The syntax has the following grammar:

```
keyword = value;
```

A simple configuration file for a linear regression model might look like Listing 3.1:

```
model_name = "Iris Flower";
algorithm = LinearRegression;
csv_path = "tests/test.csv";
train_test_split = [80, 20];
target = "species";
features = [
"sepal_length",
"sepal_width",
"petal_length",
"petal_width"
];
epochs = 100;
learning_rate = 0.001;
batch_size = NULL;
early_stop = { "patience": 10 };
output_path = "model/model.json";
```

Listing 3.1: Configuration for linear regression model

From the example, we can see the language supports the important configuration for a ML model such as the data path, train-test splitting, the target, the features, learning rate and number of epochs. In addition, the DSL gives optional parameters such as the batch size, early stop and the output path.

While the language is simple, it needs to be able to parse efficiently all the different data types and handle validation. Thus, a two-pass parser was chosen to implement it. This allows to efficiently handle the different data types and quirky and better error handling and reporting, while also leaving the opportunity to grow the syntax and improve it.

3.5.4 DSL Syntax Details

As this stage, the DSL is designed to mainly work with ML algorithms, thus the following parameters are required to create a model:

- **model_name**: The name given to the model, this acts as an identifier but serves no real purpose in the actual model.
- **algorithm**: This parameter is responsible for choosing the right model type, the algorithms are predefined to work with implemented algorithms. If this parameter is not recognized or not implemented, it will fail and will continue.
- **csv_path**: This parameter is a string value that holds either a relative or full path to the CSV data file to be opened and parsed.
- **train_test_split**: This parameter defines how the orchestrator will split the input CSV data into the training and testing set. It is an array that has 2 value which sum must be equal to a 100.
- **target**: This specifies the name of the target column in the CSV data.
- **features**: An array containing at least one column name. These are later on matched to verify that the names are valid and exist in the CSV data.
- **epochs**: The number of iterations to run. This number should be greater or equal to one.

While the option above give a good fundamentals to create an ML model, further parameter tuning is necessary, thus the DSL includes the following set of optional parameters exists:

- **learning_rate**: This parameter controls the rate change when updating the model, it can be tuned to get improved results.
- **batch_size**: This parameter is used to make the training use batches of a certain size, instead of the whole train set all at once. This can improve the performance of the model, especially in large data.

- **early_stop**: This parameter will stop the model training if no improvements have been made after a number of iterations specified by the user.
- **output_path**: This path is used to output the final model data, including the trained parameters and weights.

The optional parameters can either non exist in the given file or have `NULL` values, some will be given a default value if not specified.

The DSL might seem simple enough, but the different data types and the constant need for layers of validation made me use a two-pass parser.

3.5.5 Implementation Details

The two-pass parser consists of two main phases, in each the code is traversed to validate the input and create the final structure.

The first phase contains three components:

1. The lexer which takes the raw input and provides helper function.
2. The tokenizer, which uses the lexer to create a token stream to be processed by the parser.
3. The parser which takes the token stream, does basic grammar validation and constructs an AST along the way to be given to the second phase.

The first stage uses the AST an Intermediary Representation and passed it to the second stage for a second traversal and final validation. The second stage only consists of one component:

1. The validator which takes the AST, traverses it and does final validation and error reporting.

Let us explore each phase a bit more.

1. **Lexer**: The lexer takes the raw input as a string buffer and provides a structure to keep track of some information such as the position and helper functions to traverse the input buffer.
2. **Tokenizer**: With the help of the lexer structure and function, the tokenizer takes the buffer and produces a token stream. A token represents a substring with some meaningful data, such as keyword tokens, literal tokens, or numerical tokens. In this stage unnecessary token are skipped (e.g., white spaces, tabs, etc. . .). Together they compose the Lexical analyser in which it performs basic validation such as recognizing valid keyword and literals.

3. **Parser**: The parser or syntax analyser is the final step in the first phase. It takes the token stream as an input and generates an AST acting as an IR to be given to the second and final stage. During this stage, alongside creating the AST node by node, the parser checks for syntax errors and ensures correct grammar. This is syntax validation and in the DSL the grammar checking consists of ensuring that the `keyword = value;` is adhered for each statement. Mainly, it checks that the equal sign exists between the key and value, and that the statement must end with a semicolon (`;`). It also checks for correct data types and correct formatting (e.g. an array must be closed, a string must be closed, etc. . .).
4. **Validator**: The validator, the only component in the second stage, makes the semantic analyser. This stage works with the AST unlike the first phase, thus at this stage, the syntax is dealt with and the validator's purpose is to enforce the semantic rules such as ensuring mandatory fields are present, the correct type of the keyword, and any specific rule. For example, the `train_test_split` field must be an array with two items and their sum must equal to a 100, also the patience in the early stop should be greater than 0 but less than `epochs - 1`. All of these final checks are made in this stage, at the same time, the final internal structure is being filled with the correct data and then passed to the orchestrator for the next steps.

Upon successful parsing, the DSL internal structure is passed to the orchestrator to be used in the later pipeline stages. The next stage is the data processing phase, which we will explore in the next section.

3.6 Implementation of Data Processing Phase

3.6.1 Principles

After parsing the configuration file successfully, the orchestrator moves into the data processing pipeline. This phase is crucial to ensure reliable and robust data parsing for a successful training or inference process. There exist two data processing stages for different modes of the AI engine. The first stage is for the input data. In the current implementation of only ML models, only CSV files are supported. The input CSV contain the features and target fields and the related data. The second stage is the data output file. The chosen structure for this version is the JSON format. The JSON file is generated at the end of training and can be used to recreate the same model in a different environment or used to run inference on the model.

Considering the importance of reliable and efficient data parsing of these files, the following guidelines have been compiled:

- **Efficiency**: The parsers must be fast, reliable, and memory efficient especially for parsing large datasets or big models especially on a resource constrained hardware.

- **Data Integrity**: The parsers should be consistent across multiple hardware devices and multiple runs.
- **Reliability**: The parsers should output a clearly formatted and validated structure to be used by the other components on the AI engine.

3.6.2 Used Technologies

To guarantee optimal performance and reliability of these parsers, keeping in mind the need for portability and interoperability with other languages, the `Zig programming language` was chosen to implement both the CSV and JSON parser due to its simplicity, explicit memory management, and great interoperability with the C programming language.

3.6.3 Input Data

The Synapse Engine only supports ML algorithms as of now, since the implemented ML algorithms are simple and work with numerical data, I have decided to only support numerical data values for the input data. The decision made the implementation of the parser faster, clearer, and reliable. This also meant that a one-pass parser is enough to handle the CSV data while also keeping the room for improvements.

The CSV parser is divided into three steps:

- **Header Parsing**: The first entry line in the CSV file is considered as the header. This row defines the columns names and row length. The column names are alphanumerical strings and are parsed as strings.
- **Row parsing**: Each subsequent line is considered a data row. These rows either contain a number (either an integer or a floating point number) or a null value. Any non-numerical entry results in a failure in the parsing and is directly reported to the user. In this stage, trailing commas and null values are handled and replaced with zeroes.
- **Validation**: After parsing the rows into an internal temporary data structure, each row is validated. Each row must have the same length as the number of column.

An example of supported CSV data input file might look like:

```
Celsius,Farenheit
-50,-58
-40,-40
-30,-22
-20,-4
```

```
-10,14
0,32
10,50
```

3.6.3.1 CSV Parser Algorithm

To understand better how the CSV parsing algorithm works, let us take a look at Figure 3.7 in details.

The algorithm works as follows:

1. Split file into separate lines using the \n separator and put them into a list.
2. Initialize a counter at 0 to track line number.
3. Check the line number variable if it equals to zero
4. If the line number ins zero, it means we are reading the header so we parse the header and extract column names.
5. If the line number is greater than zero, the line is a data row and is parsed accordingly. At this step, we also do line validation and null values replacement.
6. the counter in incremented by one.
7. Check if we are at the last line.
8. If it is not the last line, we repeat step 5 to step 7.
9. If we are at the last line, we finish parsing, and output the final structure.

The use of one-pass parser with immediate validation ensures early on error detection with efficient handling, robust parsing, and fast parsing which is a key component in resource-constrained systems.

3.6.4 Model Parameters

After a successful model training, the orchestrator exports the final model data into a JSON structure. This file contains metadata about the model, the training details, and the model parameters. This format was chosen due to its simplicity and human readability. Unlike many formats, JSON provides a clear syntax that can be read by humans and thus let the end user see the final results. This output file also serves as a way to allow reproducibility and ease of deployment.

In order to achieve the wanted output, the following data must be present to ensure reproducibility and data integrity:

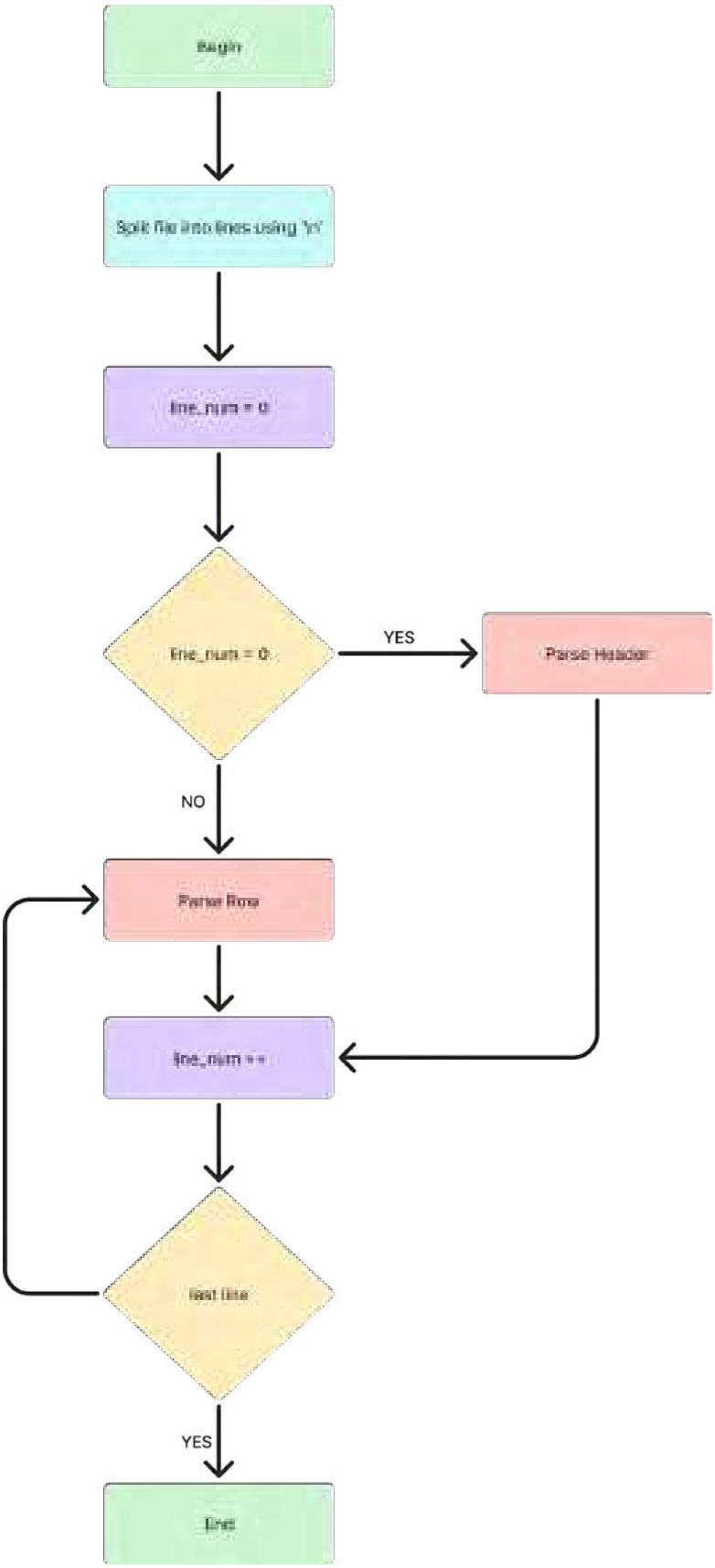

Figure 3.7: CSV Parser Algorithm. Source: Own creation

- **Metadata**: The first part of the file contains additional information about the trained model such as the schema version, the run ID, the model name, the model type, and the target name. This data makes it easier for future improvements or changes in the engine to be backwards compatible.
- **Training final details**: The next section in the final output file contains data that comes from the math engine about the training phase. Mainly, it contains the number of trained epochs and the final loss value. The number of epochs can be less than the number provided in the configuration if an early stop was triggered.
- **Model Parameters**: The final section of the file contains the two fields of an ML model. The ML model implemented (Linear Regression) contains the weights and bias parameters. The weights in an array of the same length as the number of features and the bias in a single value.

An example JSON output for a Linear Regression model looks like Listing 3.2:

```
{
    "schema_version": "0.1.0",
    "run_id": "6623_776",
    "model_name": "Celsius To Farenheit",
    "model_type": "LinearRegression",
    "target": "Farenheit",
    "epochs_trained": 1600,
    "final_loss": 6.6045463568398287e-09,
    "weights": [1.7999972407705038],
    "bias": 32.00000117429645
}
```

Listing 3.2: JSON output for linear regression model

As the example shows, the JSON data is simple and not complicated, thus, I have chosen to implement the parser as a one-pass parser since it achieves the necessary functionalities with minimal memory footprint and reliable output.

3.6.4.1 JSON Parser Algorithm

The JSON is only parsed in the inference case. The AI engine uses the JSON file to recreate the same model with the final parameters so it can infer on new data. Figure 3.8 illustrates the JSON parsing algorithm.

The algorithm works as follows:

1. Validate that the file starts with the opening brace ({).
2. Check if the next character is a closing brace (})
3. If the character is a closing, the JSON is invalid, report error and abort.
4. If the next character is not a closing tag, then we check if the last character is a closing brace (}).

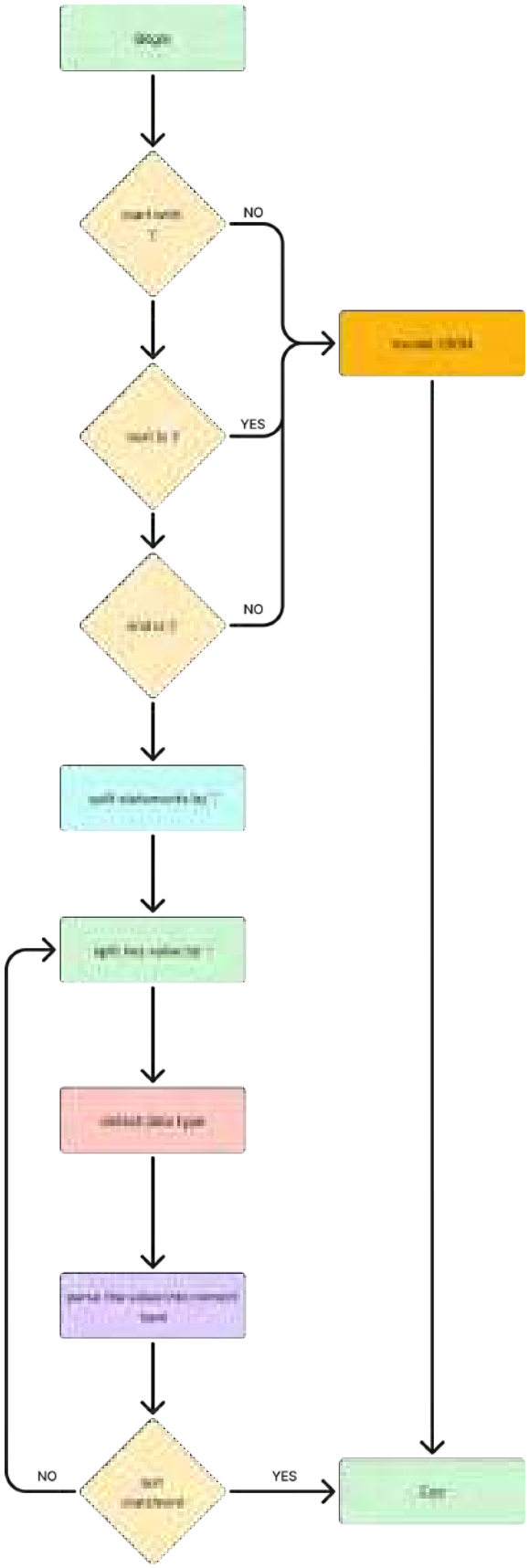

Figure 3.8: JSON Parser Algorithm. Source: Own creation

5. If the end character is not a closing brace, then the JSON is invalid, report error and abort.
6. Otherwise, we have a valid basic JSON format, so we split the file into statement by the comma (`,`) and store it into an array.
7. For each statement, we split it by the colon (`:`) into the key value pair.
8. Then, detect the correct value data type and parse it accordingly.
9. Check if we are at the last statement.
10. If we are not at the last statement, repeat step 8 to step 11.
11. If we are at the last statement, we have finished parsing, and output the final structure.

Even though the schema contains a different number of data types (including strings, integers, floating points, and arrays), the simplicity of the syntax and predictable format, makes the one-pass parser the ideal choice. The parser also validates the expected data type for each field while it is parsing the field and checks that all fields are present as required.

Any violation of the schema syntax, missing field, or an incorrect type will result in a descriptive error message that will indicate where the problem is (field or line) to ensure clear error reporting to the end user

3.7 Implementation of Math Engine

3.7.1 Concepts & Principles

The Math Engine is the core component of the Synapse Engine project. It is responsible for the core implementation of the ML algorithms and the core training loop. Thus, the engine must be highly accurate, efficient, and robust, especially in the computations modules necessary for ML algorithms. The current implementation only implements a linear regression model.

To achieve the desired output and ensure maximum efficiency, the following set of guidelines have been compiled:

- **Modularity**: The engine must be modular in its implementation, necessary to ensure future extensibility, and the use of only needed modules.
- **Safety**: The engine must be memory safe, especially in handling large datasets.
- **Accuracy**: The math engine must be reliable and accurate in the calculations. This is important, especially in an iterative process.

- **Efficiency**: The engine performance must take advantage of the maximum hardware resources to improve the training process.

3.7.2 Used Technologies

The Math Engine needs to be efficient, fast, and memory safe above all. It also needs to be able to interact with different hardware components to take maximum advantage of the available acceleration. Thus, the `Rust programming language` was chosen to implement it.

Rust provides memory safety, and with its unconventional memory management it allows for concurrency without race condition. It also provides the Foreign Function Interface (FFI) that facilitates the integration with C and C-compatible languages. This means that the math component is safe, can be expended easily later on, and can communicate with different components through its FFI layer.

3.7.3 Implementation

The Math Engine at this stage only focuses on ML algorithms. Currently, the engine implements Linear Regression and Multi-Linear Regression algorithms. In this section, we will focus on the details of the design and implementation of the math engine to see how it achieves this.

3.7.3.1 Linear Regression Training Algorithm

First, let us take a look at how the training algorithm works. The algorithm in the Math Engine uses the SGD optimizer using batch processing and epochs, described as follows (see Algorithm 3.1):

Algorithm 3.1: Gradient descent algorithm

Result: Optimized parameters
Initialize weights and bias;
for *each epoch* **do**
  Compute predictions $yHat$;
  Compute gradients $(dMSE/dw, dMSE/db)$;
  Update parameters: $weights = weights - learningRate * dMSE/dw$;
  $bias = bias - learningRate * dMSE/db$;
  Check for convergence or early stopping
end

Figure 3.9: Math Engine Architecture. Source: Own creation

3.7.3.2 Architecture Overview

The math engine is structured into seven layers to achieve its functionality. Figure 3.9 illustrates the layers of the Math Engine. These layers are:

1. **FFI Layer**: The FFI layer exposes C-compatible structures for input and output to interact with the AI engine externally. The layer only exposes C header files with certain structures in order to get the data and exposes structures for the final model data.
2. **LinearRegression Layer**: This layer exposes the main structure for the linear regression model, it uses the solver loop to train and update the parameters before filling the C structure and exposing it to the FFI layer.
3. **Solver Loop**: The solver loop uses the linear algebra functions to train a model. this solver abstracts the training loop from any model. It uses the other layers to also calculate the loss.
4. **Linear Algebra Layer**: The Linear Algebra layer uses the layer below it to create the core algorithms of the solver such as loss functions, the gradient algorithm, and the activation functions.
5. **Statistics Layer**: The statistics layer offers stats functions such as mean calculation, median calculation, and other stats functions used later by the solver and some linear algebra functions for the training.

6. **Math Layer**: This layer declares the basic mathematical structures needed for linear algebra or any ML algorithm. Mainly, it exposes three data types.
    - Scalar, which is a floating point data type with different floating function.
    - Vector, which is a one dimensional structure that holds data of a certain type.
    - Matrix: A matrix is a two-dimensional array that usually holds the data from the parsed CSV file.
7. **Random Layer**: The random layer generates random numbers. Its sole purpose is for these numbers to be used as initial parameters for the model.

The layered architecture of the Math Engine enables easy maintenance of the engine, keeping everything clearly separated for future updates or changing one module for another. This enhances reliability and performance. This also enables the for future updates and extensibility easily without compromise.

3.8 Implementation of the Orchestrator & CLI

3.8.1 Principles

The orchestrator is the main component of the AI Engine, it is the glue that holds all the other components together to form the final system. The orchestrator contains three major components which are the core logic that links and verifies the subcomponents, the CLI system, and the logging module.

Due to the criticality of this component, it must be reliable, efficient, and performant. Thus, a comprehensive guideline has been compiled:

- **Modularity**: The orchestrator must be modular to allow for ease of maintenance, upgradability, and extension.
- **Robustness**: The orchestrator must be reliable and predictable in all cases.
- **Performance**: The orchestrator must be implemented efficiently and have minimal latency during user interactions.

3.8.2 Used Technologies

Due to the importance of the component, more than one technology has been used to develop the component. It was mainly developed with C and C++ to create the three components and tie them together. `C++` has been used to implement the core and the CLI tool, whilst `C` has been used to implement the logger.

The logger was developed in C is made to the simplicity of the wanted result and the interportability with other system languages such as Zig or Rust.

3.8.3 Orchestrator

The Orchestrator component is the brain and the control unit that coordinates the tasks between the various AI Engine components including parsing (CSV, SYNJ, and JSON), handling the CLI commands, logging, and running the training and inference processes. The implementation details include:

- **Centralized Entry Point**: The `main()` function acts as the entry point managing command handling.
- **Dynamic Command Dispatching**: Commands are dispatched using a hashed command lookup, optimized for rapid retrieval and execution without compromising performance.
- **Model Lifecycle Management**: Coordinating parsing, validation, training, serialization, and inference steps with clear checkpoints and validations as each step.

3.8.3.1 Orchestrator Algorithm

The orchestrator does not have any complex logic behind it, its purpose is to validate the CLI commands, parse them and call the necessary modules to execute the task. The algorithm is explained below (see Algorithm 3.2):

Algorithm 3.2: Algorithm Orchestrator Main

```
Parse command-line arguments;
if command is recognized then
    Dispatch to relevant command handler;
    Execute handler workflow (train, predict, etc.);
else
    Display error and help message;
    Log all interactions and errors;
end
```

3.8.3.2 Orchestrator Architecture

Figure 3.10 illustrates the four layers that compose the orchestrator.

1. **CLI**: The CLI is the highest level of the orchestrator and the AI engine. The user interact with the CLI tool to run commands and invoke the AI engine.

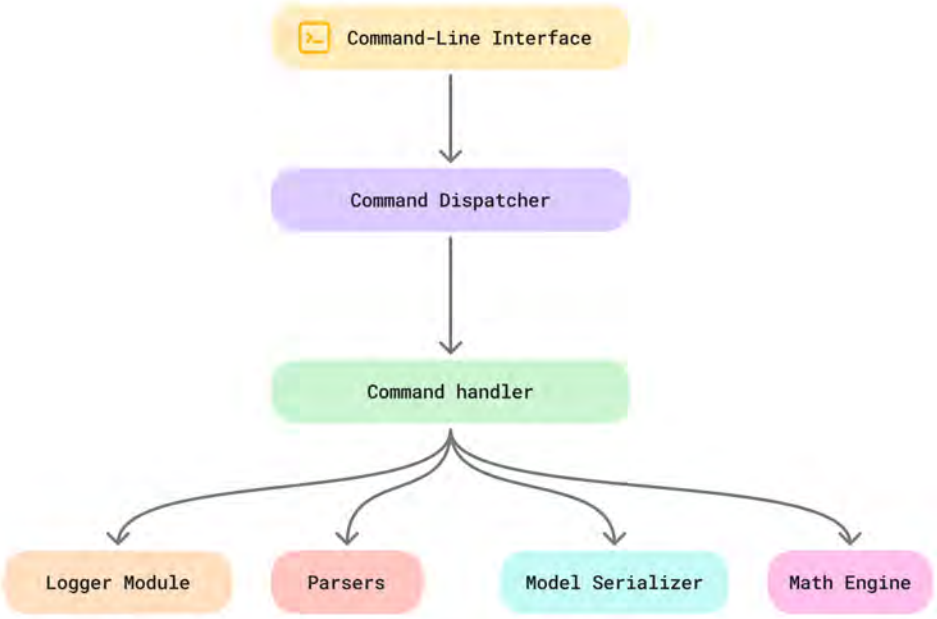

Figure 3.10: Orchestrator Architecture Source: Own creation

2. **Command Dispatcher**: After typing a command in the CLI, the AI engine in invoked, and it first calls the command dispatcher which checks that the typed command exists and written correctly. In case of failure, the dispatcher throws an error and shows a help message to show the correct usage.

3. **Command Handler**: If the command exists and is correct, the command handler parses the final flags, construct the necessary structures and uses the other components (logger, parsers, serializer, and the math engine) to complete the task.

3.8.4 CLI

The CLI offers simple, intuitive commands with extensive feedback and validation to offer the best user experience possible. In the implementation of the AI Engine, I have decided to use compile-time hashes to represent the different commands to ensure fast command lookup and minimal overhead to not compromise the performance of the whole system.

3.8.4.1 CLI Command Parsing

The CLI uses the FNV-1A hashing algorithm at compile time to optimize command recognition. This drastically improves lookup speeds and reduces runtime computation. Flag parsing is implemented through a custom parser handling boolean, integer, and string flag types efficiently.

3.8.4.2 CLI Available Commands

The CLI provides several user-friendly defined commands, each tailored for specific tasks:

- `train`: Initiates training using configuration files.
- `predict`: Runs model inference based on input data.
- `param`: Facilitates the creation or editing of configuration files.
- `dump`: Displays final model details and statistics.
- `help`: Provides guidance and usage information.
- `version`: Displays current software version details.

3.8.4.3 CLI Execution Algorithm

Algorithm 3.3 describes the workflow for CLI execution:

Algorithm 3.3: Algorithm CLI Execution:

```
Parse initial user arguments;
Hash the command input;
if hash matches known command then
    Dispatch command to handler;
    Parse additional flags and arguments;
    Execute command-specific logic;
else
    Display unknown command message;
    Provide detailed logging at each step;
end
```

3.8.5 Logger Module

The Logger module enhances the user experience by providing visually distinct and informative feedback. Utilizing ANSI colour codes and formatting, it makes it easier to spot different types of messages (error, logs, success, etc. . .). The implementation of the logger is very kept simple and configurable for low overhead and easy of modularity later on as needed.

3.8.5.1 Logger Features

The key features of the Logger module include:

- Multiple log levels (TRACE, INFO, SUCCESS, WARN, DANGER, FATAL).
- ANSI colour-coded outputs for better clarity.
- Macro-based ABI simplifying integration with other components.

3.8.5.2 Logger Algorithm

The logging algorithm is described below (see Algorithm 3.4):

Algorithm 3.4: Logger Algorithm

```
Determine log level;
Select appropriate ANSI color/style;
Format message with caller context;
Output formatted message to terminal;
```

3.8.5.3 Logger Architecture

The logger is simple in architecture, and only has four layers, as illustrated in Figure 3.11 below.

The logger exposes C macros and a general function to be used with any C compatible languages such as C++, Zig or Rust. When a macro is called, it translates the actual function. The function, after determining the log level, uses the ANSI colours string formatters to create a styles log before outputting it to the terminal.

Through rigorous design and structured implementation, the Orchestrator, CLI, and Logger modules provide a powerful, user-friendly interface ensuring efficient, reliable, and informative interactions with the Synapse Engine while not sacrificing any performance.

3.9 Implementation of the ARM kernel

We have looked at the implementation details for the AI engine in eh earlier sections, we have covered the design choice, implementation details, and how the AI engine works using the optimizations. The second component in the project is the custom ARM-based kernel. This kernel provides more low-level functionalities optimized for ARM embedded systems and running AI models on them. In this section, we will detail the design and implementation of the custom kernel developed.

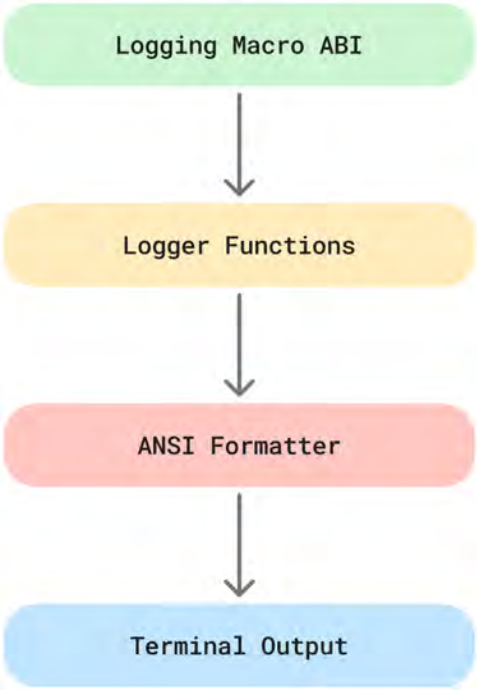

Figure 3.11: Logger Architecture. Source: Own creation

3.9.1 Principles

Creating an embedded-system ARM-based kernel from scratch requires good planning and meticulous development. An embedded kernel must be performed adhering to a specific rule set. Thus, the following guidelines have been compiled to ensure the clear and reliable kernel:

- **Determinism**: The kernel must be predictable, execution time and handling interrupts are critical to ensure the system works as intended without interference as it must be reliable in critical-performance applications.
- **Abstraction**: The kernel should abstract hardware specificities.
- **Modularity**: The kernel should be modular in order to achieve maximum compatibility with most ARM-based system.
- **Extensibility**: The kernel should be able to extend easily to support new features or new architectures.

To achieve the desired results, the kernel was developed using multiple technologies and was developed on a multi-layered approach. In the subsequent subsections, we will discuss further details.

3.9.2 Used Technologies

To achieve maximum efficiency and complying with the guidelines, the following the languages and technologies were used:

- **Assembly**: ARM assembly was used to create the bootloader, hardware specific control, and early hardware control.
- **C**: The core and services of the kernel were developed in C. C gives a higher-level control without hardware specifics.
- **Cross-Platform Toolchain**: A set of Makefiles alongside cross-platform compilers was used to build the different kernel parts.

3.9.3 Kernel Overview

The kernel is divided into 4 distinct layers that sit on top of the actual physical hardware. Each layer serves a purpose and relies on the underlying layers to achieve its functionalities. Figure 3.12 illustrates a comprehensive overview of the custom ARM kernel designed to be optimized for embedded AI.

The system is organized into five distinct layers, the layers are:

- **Physical Hardware**: The kernel was designed to be run on multiple ARM-based processors. The i.MX 8 Plus System on Chip (SoC). The i.MX 8 Plus is a quad-core Cortex-A processor, a Cortex-M M7 microprocessor with an integrated Graphical Processing Unit (GPU), and a Neural Processing Unit (NPU). This compact form that encapsulated all the hardware needed to run, or train an AI ML model made it a suitable choice for the project. The layer also includes the other components for a full system such as Random Access Memory (RAM), peripherals, etc...
- **Boot Process**: The boot process is the first layer in the kernel, it is written in assembly and its purpose is to initialise the hardware, set up the hardware into a known state and provides early hardware access for debugging such as UART. This stage also setup up the environment for higher-level languages and provides hardware abstractions.
- **Core Kernel**: The core kernel is the core of operating system, it is mainly implemented in C. This layer uses the boot layer and abstractions to provide all the necessary functionalities to the operating system. All the subsystems in the core layer are necessary to the functioning of the system. If any of these fail, the kernel aborts and does not finish the initialization process. The core kernel includes these main subsystems:
  - **Memory Subsystem**: The memory subsystem is responsible for managing the system's memory and provides abstraction functions to the higher-level layers to interact with the device memory.

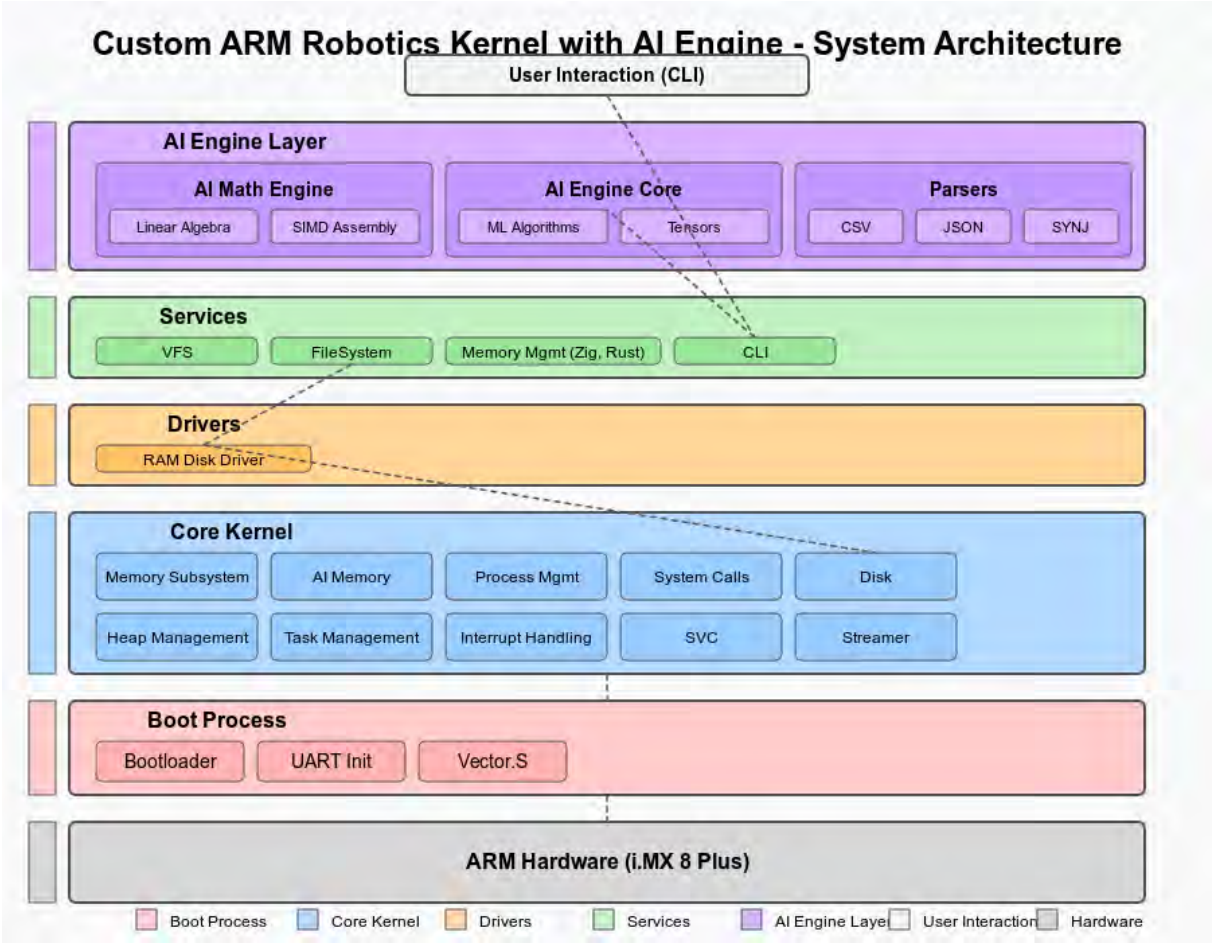

Figure 3.12: Custom ARM Robotics Kernel. Source: Own creation

  - **Process Management**: The core kernel is responsible for managing processes, multitasking, and scheduling tasks.
  - **Interrupts & System Calls**: The final primary task of the core is providing interrupts and systems calls handling. The calls include either interacting with system hardware (e.g., accessing a peripheral, sending data, etc. . .) or higher-level access such as keyboard input, printing to the screen etc. . .

- **Drivers**: The drivers layer is an extension to the core kernel components, this acts as a modular part of the core kernel. The drivers are high-level hardware specific functionalities such as a GPU driver for maximum compatibility or Disk drivers.

- **Kernel Services**: The kernel services sit above the kernel core components. This layer uses the core components to provide higher-level modules that can be loaded and unloaded as needed. This gives flexibility to the kernel to function on any given hardware. These services include the Virtual File system and some higher-level APIs to the user space to be used.

This layered, modular architecture ensures robust abstraction while also giving efficient communication between layers. It also provides flexibility and thus makes the

kernel able to function on a variety of hardware and gives the possibility of future extensibility for other hardware chips or other architectures as needed.

In the following subsections, we will dive into each of the components and layers in more details, seeing the design and implementation processes that went into them.

3.9.4 Bootloader

When the system is first started, a small program titled the bootloader. This program is the transition layer between the power-on hardware to a fully operational kernel ready to handle tasks. For the custom ARM-based kernel, a custom bootloader has been developed from scratch in order to be as minimal as possible while giving all the functionalities needed. This process is composed of a multi-stage sequence. Figure 3.13 illustrates the four phases of the boot process.

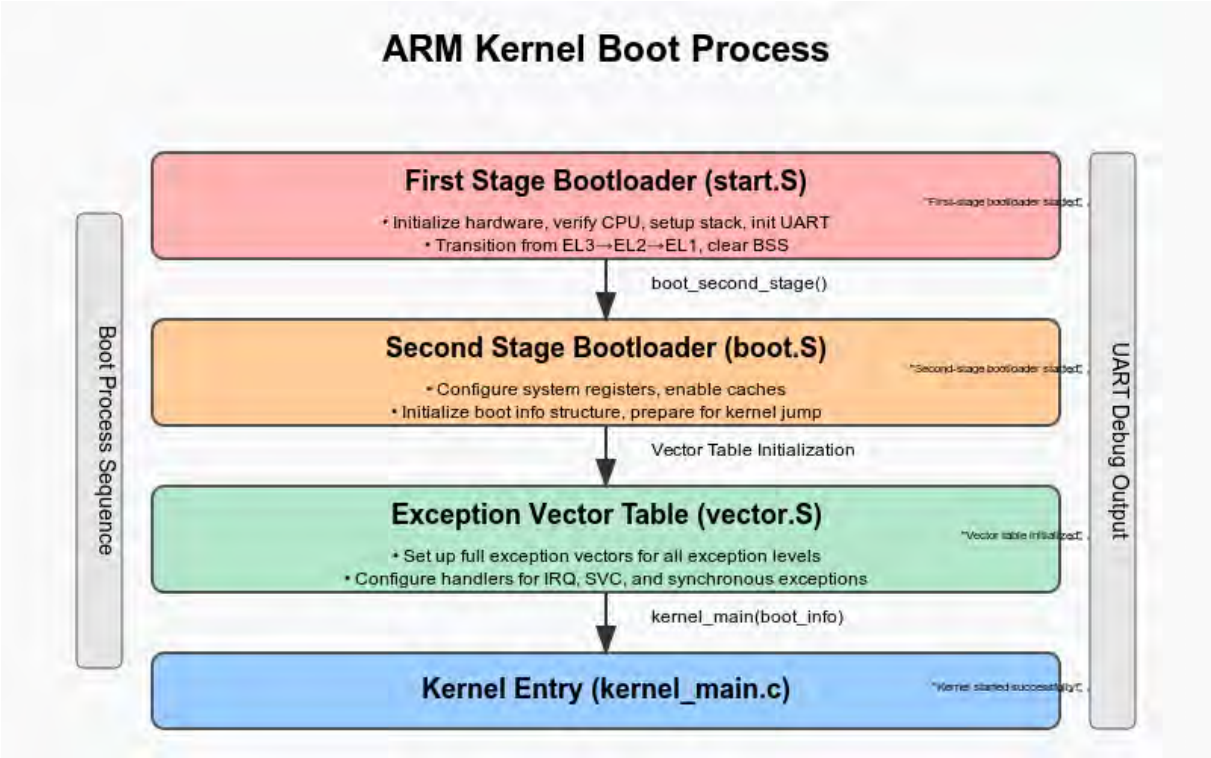

Figure 3.13: Kernel Boot Process. Source: Own creation

The boot sequence responsible for initializing the hardware and setting up hardware-specific components is composed of the following stages:

1. **First-Stage Bootloader**: This is the entry point of the OS Kernel. The hardware is instructed to look at a certain address in the memory which contains this stage and start the boot process. Due to the lack of environment support and the system being in an unknown state, this stage is developed using Assembly. This stage is mainly responsible for initializing the fundamental hardware, verifies the CPU mode and that it is running on the primary core in a multicore system. It

also initializes early UART for early debugging and early communication. Then, this stage sets the CPU in the correct privilege mode by putting it into the correct exception level (EL1). Finally, the first stage has done most of the important initialization and passes the control to the second stage.

2. **Second-Stage Bootloader**: This stage takes over the first one in order to do additional hardware system configuration. It mainly configures the system registers, it sets flags and configures certain hardware components such as caches, the Memory Management Unit (MMU) and any other component needed. It also uses the Exception Vector Table to initialize a vector table with the interrupts for early system. Then, it prepares the environment for the execution for high-level languages such as C and finalizes the runtime environment. Finally, it initializes an internal `boot_info` structure that contains details about the initialized hardware such as the amount of RAM, the architecture it is running on, and a magic code to verify the control was handed correctly to the core kernel.

3. **Exception Vector Table**: Before the execution begins, the ARM processors need to have the exception table setup. This table installs exception handlers for all the supported exception levels. It is critical to ensure correct address mapping, fast interrupt responses, etc... This guarantees a predictable control flow in normal execution mode and when an exception occurs.

4. **Kernel Entry**: After the two stages have finished setup up the low-level details of the hardware and did the necessary checks and configuration, it passes the control to the kernel core entry point. At this stage the bootloader is done, it loads the full kernel and passes the control to it. This constitutes the final stage in the boot sequence, and now everything is running properly in a known environment. This stage is usually written in a higher-level language, such as C in this case.

It is important that each stage executes flawlessly to ensure the reliability, stability, and security of the system. After the sequence, the kernel has a stable foundation that it can use to create the core components and later on the services.

3.9.5 Early Communication

Since the kernel is meant to be run on small systems and have a minimal footprint on the end device, a Graphical User Interface (GUI) was not developed for this version of the kernel. Thus, in order to achieve communication, early debugging, and having a minimal formatted output, the kernel uses UART provided by the hardware. It gets initialised in the boot process and provides abstracted functions to the higher-levels.

3.9.6 Exception & Interrupt Handling

Robust exception and interrupt handling in a kernel is fundamental for predictable and reliable system functioning, especially in an embedded environment where precise

event response is critical. Figure 3.14 portrays the architecture of interrupts and system calls implemented in this kernel.

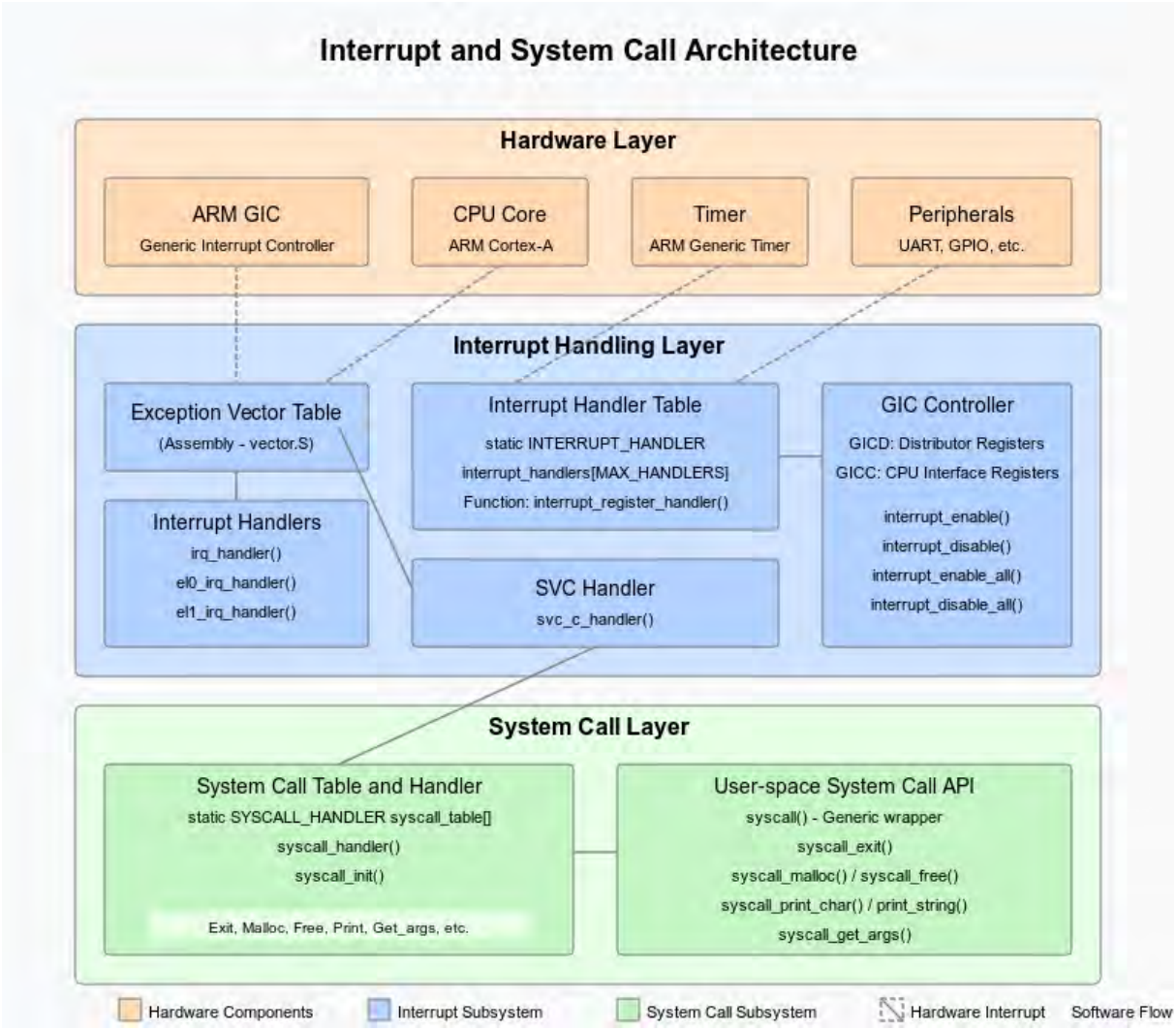

Figure 3.14: Interrupt and System Call Architecture. Source: Own creation

The architecture is organized into three principal layers:

1. **Hardware Layer**: This layer encompasses all physical interrupt sources and controllers:
   - **ARM GIC**: Aggregates, prioritizes, and routes hardware interrupts from peripherals and the CPU core.
   - **CPU Core**: Executes exception-level code and services interrupts at EL0, EL1, or higher.
   - **Timer (ARM Generic Timer)**: Periodically generates timer interrupts for scheduling and timekeeping.
   - **Peripherals**: Devices such as UART, GPIO, and others, each potentially generating hardware interrupts.

2. **Interrupt Handling Layer**: This layer translates hardware events into software actions:

    - **Interrupt Handlers**: Core C functions such as `irq_handler()`, `el0_irq_handler()`, and `el1_irq_handler()` are invoked based on the interrupt origin and type. These routines save the processor context, determine the cause of the interrupt, and dispatch to appropriate service routines.
    - **Interrupt Handler Table:** `interrupt_handlers[MAX_HANDLERS]`, is a static array of function pointers registered dynamically via the following function `interrupt_register_handler()`. This enables flexible, runtime binding of custom ISRs for each interrupt source.
    - **GIC Controller Access:** The kernel communicates directly with GIC distributor (GICD) and CPU interface (GICC) registers, using functions like `interrupt_enable()`, `interrupt_disable()`, and their "all" variants to globally mask or unmask interrupt lines.
    - **SVC Handler:** The `svc_c_handler()` function traps software interrupt requests (Supervisor Calls), enabling controlled entry into privileged kernel services from user space.

3. **System Call Layer**: This layer enables safe, abstracted access to kernel services from user-space applications:

    - **System Call Table and Handler:** The kernel maintains a system call dispatch table (`static SYSCALL_HANDLER syscall_table[]`), mapping syscall numbers to C handler functions. The generic `syscall_handler()` routine performs validation and argument marshalling before invoking the target syscall. Typical system calls include process exit, dynamic memory allocation (`malloc`, `free`), output functions, and environment queries.
    - **User-space System Call API:** Applications invoke system calls via a generic `syscall()` wrapper, which issues an SVC instruction to trigger a privilege transition. High-level ABIs such as `syscall_exit()`, `syscall_malloc()`, `syscall_free()`, `syscall_print_char()`, and `syscall_get_args()` simplify interaction with kernel services.

Flow of Events

- **Hardware Interrupts**: Peripherals or timers assert interrupt lines, which are routed by the GIC to the CPU. The CPU jumps to the relevant entry in the exception vector table, which then invokes the registered handler from the interrupt handler table. Upon completion, the original processor context is restored.
- **Software Interrupts (System Calls)**: User applications trigger system calls via the SVC instruction. The SVC handler captures this, validates the syscall number, and dispatches to the relevant kernel routine. Results are returned to user space, preserving security and isolation.

This architecture enforces clear separation between hardware events and privileged software services, allowing efficient, scalable interrupt processing and a safe, extensible system call interface—both essential for responsive, multitasking embedded systems.

3.9.7 Kernel Memory Management and AI Memory Subsystem

The memory management subsystem is the most critical point in the system after the boot process. This component is responsible for efficiently managing the hardware memory resources throughout the lifetime of the system. The memory in systems differs how it is managed between a kernel implementation and another. Since the focus of this project is optimization of AI on resource-constrained systems, the memory subsystem in this kernel exposes direct APIs for a memory that is optimized for AI structures. Figure 3.15 shows the memory layout implemented in this kernel.

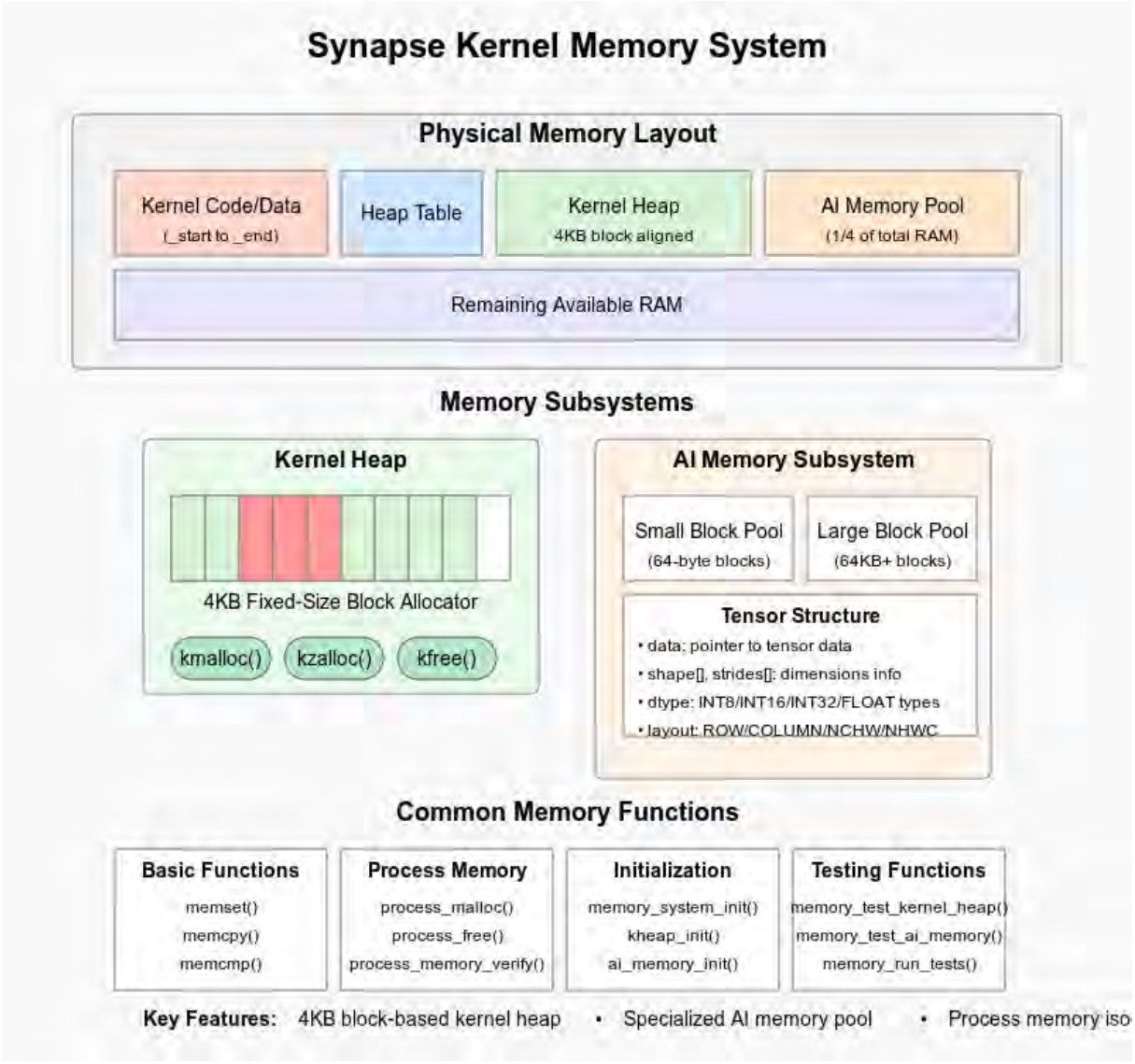

Figure 3.15: Synapse Kernel Memory System. Source: Own creation

3.9.7.1 Memory Layout

The Physical memory layout explains how the kernel partitions the physical system RAM into regions. These regions are:

- **Kernel Code/Data**: This is a reserved section of the RAM that contains the loaded executable kernel and its related static data.
- **Heap Table**: This section is responsible for storing block allocations metadata for the dynamic memory management. When an another subsystem or a user calls memory functions such as `malloc` and `free`, the heap table keeps track about the heap blocks.
- **Kernel Heap**: The heap is the dynamic memory in the system. It is a reserved empty section of the system RAM, typically a quarter of the available region, divided into blocks used for general-purpose allocations.
- **AI Memory Pool**: This is a dedicated memory region, typically a quarter, exclusively reserved for AI operations and tensor storage. It exposes its own ABI to be used and will not interfere with the heap.
- **Remaining Memory**: The remaining memory space is reserved for further expansion (e.g. loading more kernel services), add more memory for the AI pool, or can be used to map I/O peripherals so they can be addressed.

3.9.7.2 The Memory subsystems

As described above in the figure, the kernel has two major systems in the memory subsystem to manage the dynamic memory.

- **Kernel Heap**: The heap is the general-purpose dynamic memory existing in any of the operating systems. In this kernel, the heap is divided into 4KB fixed-size blocks for efficient allocation and deallocations.
- **AI Memory**: This is a specialized region of the RAM that provides rapid access for AI structures and operations. This region manages two pools. The first is the *Small Block Pool*, it allocates blocks of 64 bytes, ideal for small data, tensors, or scalar value. The second is the *Large Block Pool*: This handles allocations in blocks of 64KB supporting large tensors and big data for ML algorithms.

The AI memory pools are physically isolated to ensure deterministic allocations, high performance, and preventing data from overlapping with the other memory systems or kernel region. Having the AI memory pool inside the kernel ensures lower latency when implementing AI algorithms and provides a secure region for processing data.

Tensor Structure and Memory Layout The central feature of the AI pool is the support for high-performance tensor operations. All tensor memory is allocated from the AI memory pool. Figure 3.16 portrays the tensor structure and the memory layout for the AI region.

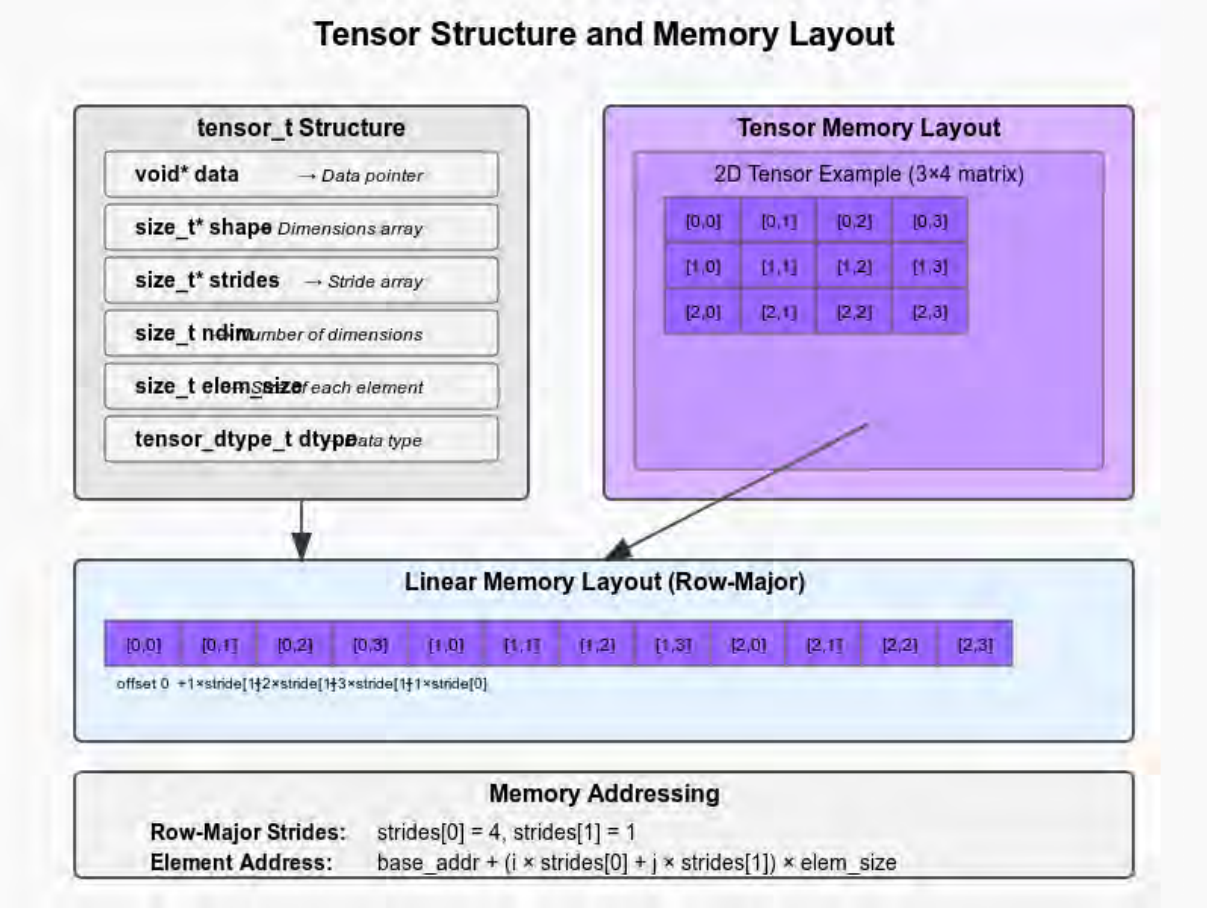

Figure 3.16: Tensor Structure and Memory Layout. Source: Own creation

- **tensor_t Structure:** Each tensor is described by a simple structure that keeps track of its data and shape. This includes:
  - A pointer to the actual data (such as numbers or measurements).
  - An array that stores the size of each dimension (for example, rows and columns for a matrix).
  - An array for “strides” that helps the system quickly find elements in memory.
  - Information about how many dimensions the tensor has.
  - The size of each element (e.g., 4 bytes for a float).
  - The type of data stored (such as 8-bit integer, 32-bit float, etc.).
- **Tensor Memory Layout**: Tensors are stored as one continuous block of memory, with elements arranged row by row. For example, a 3x4 matrix will have all its values placed one after another in memory, starting with the first row, then the second, and so on. This makes it quick and easy for the system to access any part of the tensor and process data efficiently.

- **How Elements Are Accessed**: The stride information is used behind the scenes to calculate where each value is stored. This allows the kernel to find and work with data quickly, even for tensors with many dimensions.

This design allows the kernel's AI modules to handle everything from simple vectors to large, multidimensional arrays efficiently—essential for real-time AI tasks on embedded hardware.

Common Memory Functions To support kernel and AI operations, the memory system exposes a robust API, including:

- **Basic Memory Utilities:** `memset()`, `memcpy()`, `memcmp()` for low-level operations.
- **Initialization and Testing:** Functions like `memory_system_init()`, `kheap_init()`, `ai_memory_init()`, as well as test routines to validate different allocators integrity.

In summary, the kernel memory subsystem is designed to balance the needs of classic memory management like any other OS, but also innovates in implementing the AI memory pool for rigorous demands of embedded AI. This architecture provides both flexibility and high performance.

3.9.8 Process & Task Management

Now that most of the core kernel is implemented, only process management is missing. Process management is fundamental to the functioning of any kernel. Process management also includes task management, and together they form the process management subsystem. This subsystem is responsible for efficient resource allocation, running tasks, scheduling, and enabling multitasking. Figure 3.17 provides an overview of the subsystem architecture and different components.

The system is organized around several components interacting together:

1. **Process Allocation Table**: The process allocation table is at the core of process management. it is responsible for tracking all memory allocations for any active process resources within the system. Each entry contains information about the allocation size and the allocation memory pointer, packed into the `process_allocation` structure containing the entry data. Each process has its own process allocation table indexed to ensure each process has safe resource tracking and released upon process termination.
2. **Process Structure**: To keep track of processes, the kernel represents each process by its internal data structures. The structure contains data about the identity and resources used by the process such as the process ID, the process name, the main task for this process, where the process exists in memory and its size,

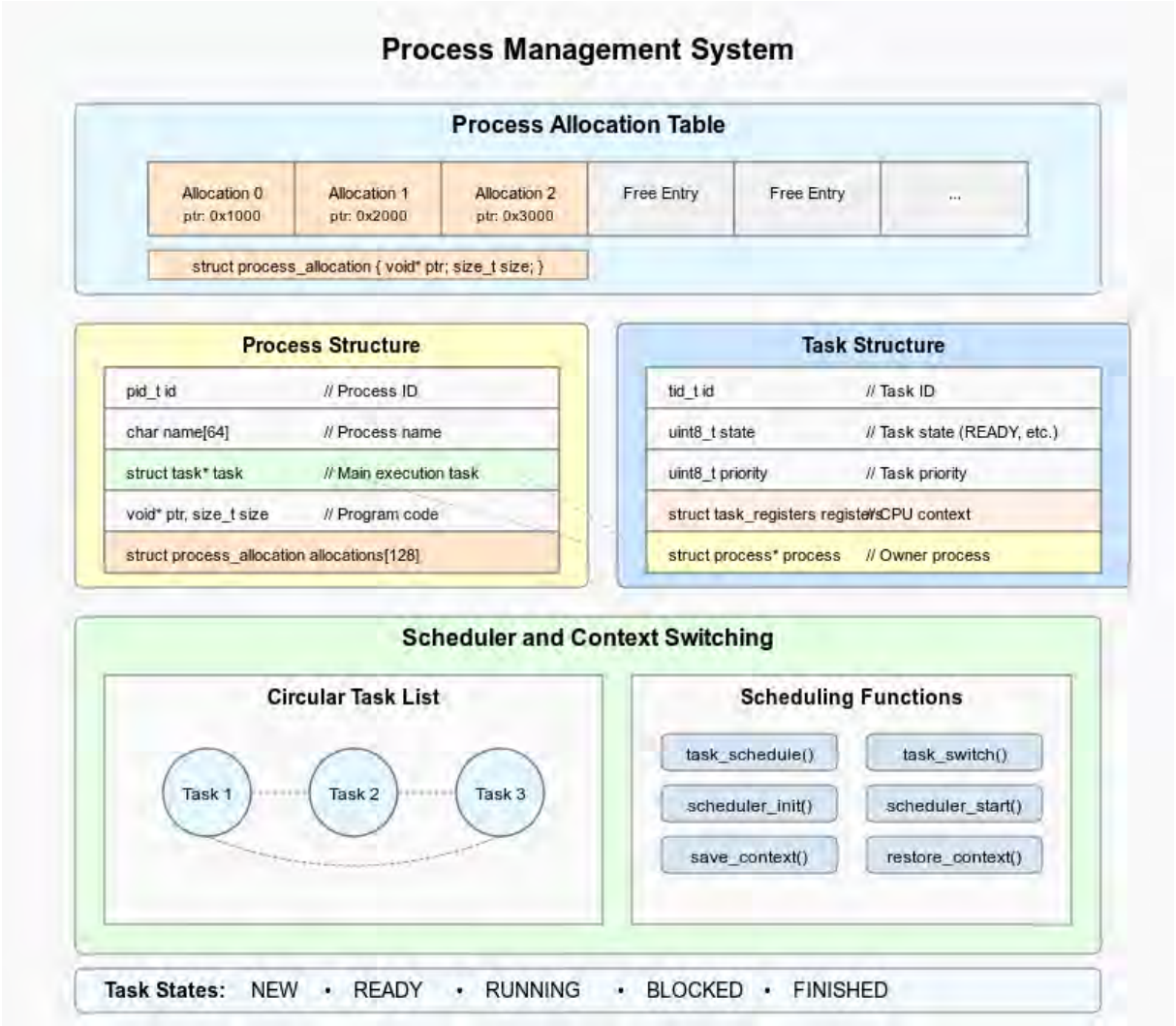

Figure 3.17: Process Management System. Source: Own creation

finally it contains the process allocation table for this process. Each process has a total of 128 allowed allocations.

3. **Task Structure**: A task is the fundamental component of a process. each process is formed from at least one task. To keep track of each task, the kernel uses an internal structure containing the task ID, the task state, the task priority, the registers, and the parent process. The state and priority of a task are used to figure which task is next. the registers state is saved for context switching, so when a task resumes, the processor state is restored, and saves before putting the task on hold.

4. **The Scheduler**: The purpose of this component is critical for multitasking and ensuring reliable and predictable task processing. The scheduler processes the task list as a circular linked list. Meaning the scheduler goes through the task in order, and when it reaches the last task, it goes back to the first one. The task state manages the task transitions. the states are: **New, Ready, Running, Blocked** and **Finished**. This design allows for deterministic control over the execution flow and enables the system to efficiently manages the resources and

ensure everything cleans up properly.

3.9.8.1 Execution Flow

- When a new process is created, the kernel creates the process structure and all its related tasks, at least the main task exists.
- The scheduler cycles through the circular task list, selecting the `Ready` tasks for executing next.
- When a task is blocked or yields, the task context is saved and the next ready task is running.
- When a task is terminated, it triggers a cleanup routine that frees the resources used.

The structure of the process and task management subsystem ensures predictable, modular, and extensible process management. This is a critical and necessary component for a real-time or embedded systems' kernel.

3.9.9 Disk Management

For the current kernel version, I have opted to use a similar design to the Linux `initramfs` service. This decision was taken to provide a fast and efficient storage suitable for small embedded systems, especially in early stages. The disk driver is a RAM-based disk implementation. This allows for ultra-fast I/O operations and is great for temporary data and early testing. Figure 3.18 illustrates how the disk subsystem works in the `Synapse Kernel`.

As the above figure demonstrates, the actual implementation is divided into three layers:

1. **RAM space**: This is a reserved portion from the system RAM to act a storage space.
2. **Driver**: The disk driver accesses the reserved RAM space and implements specific functions that emulate a disk (e.g., read, write, etc. . .).
3. **Disk Interface**: The disk interface, a universal ABI provided by the kernel to the services and higher-level layers.

the driver layer manages operations involving the disk-reserved memory region. The implementation files are meant to mimic an actual disk implementation, and internally they provide normal disk function and converts the logic into ram memory.

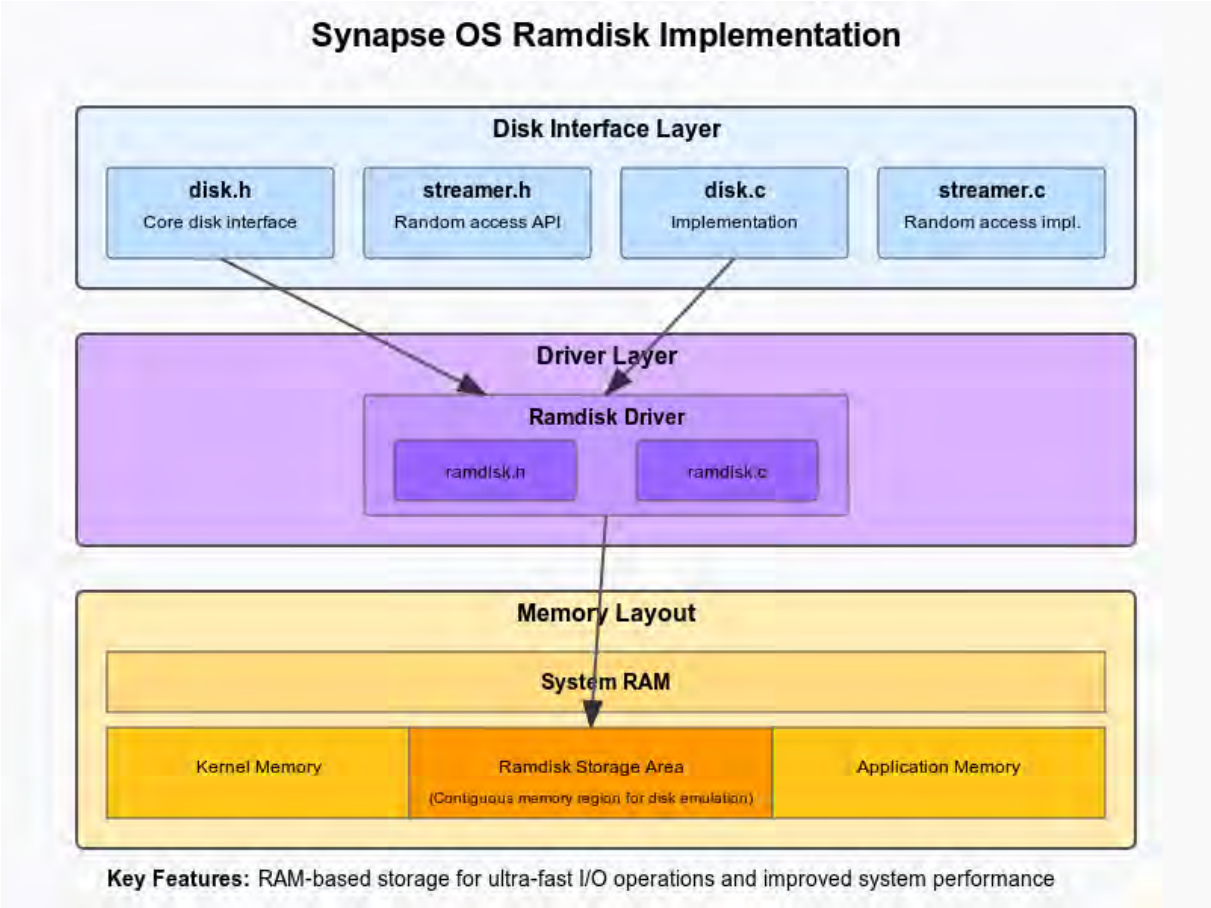

Figure 3.18: Synapse OS Ramdisk Implementation. Source: Own creation

The Disk Interface layer provides a uniform ABI for all disk like devices including emulated disks and any physical disk from different interfaces including SATA, PCIe, etc. . .

This architecture allows for more modular approach in the kernel development and allow it to adapt to different environments, especially in embedded systems that might have non-conventional permanent storage such as eMMC or SD Cards.

3.10 Conclusion

In this chapter, we have explored the development phases of `Synapse Project` in-depth, highlighting the importance of the project and the approach takes behind it to facilitate the creation and deployment of AI ML models with simple commands. The components of the project, `Synapse Engine` and `Synapse Kernel` were developed to focus on modularity, portability, and optimal performance especially in embedded and resource-constrained systems.

Chapter 4

EVALUATION

In this chapter, we will evaluate the system's performance and accuracy for `Synapse Engine` through experiments. The evaluation focuses on key metrics crucial to real-world applications, including runtime efficiency, minimal memory usage, and model accuracy. The experiments are made using real-world datasets used in education and in a real-world application.

4.1 Metrics

To objectively evaluate the project and see if it achieved its wanted purpose, a list of metrics has been compiled to provide insights into the efficiency and accuracy of the engine:

- **Runtime**: Measures the execution of the training process time in seconds.
- **Memory Usage**: Peak memory consumption measured in Megabytes (MB).
- **Accuracy**: Evaluate the trained model outputs with actual analytical values.

4.2 Experiments

In order to evaluate the `Synapse Engine` project, I have designed a set of experiments to compare all the results from the metrics set. Specifically, the experiments compare `Synapse Engine` with `LinearRegression` and `SGDRegressor` from Scikit-Learn library [36]. These experiments focus on both single-variable and multi-variable linear regression models. During the experiments, each algorithm was executed 10 times to get a consistent measure of runtime, peak memory usage, and accuracy. The experimental procedure is done into three steps:

1. Execute the algorithm 10 times and record runtime and memory usage at each.
2. In single-variable regression, record bias, slope, and loss for accuracy comparison.
3. Conduct comparative analysis for all the scenarios.

Only the single-variable regression model's parameters and loss were recorded since the chosen had an analytical solution that the parameters could be compared to, unlike the multi-variable model which only the memory and runtime values were compared.

In the experiments, to get a real-world value and accurate results, I have chosen the `Celsius to Fahrenheit` dataset for single-variable regression and the `California House Pricing` dataset for multi-variable regression.

In each experiment, the program load the data, parses it into the structure, train the model then prints the results into the standard output to be parsed by the experiment script. In the case of the Synapse Engine, the whole process is done with the configuration file and with the different components in the project. In the case of the Python programs that use the `Scikit-Learn` library for training the model, it also uses pandas to load the CSV data and parse it into the correct structure.

4.3 Results

4.3.1 Runtime Performance Comparison (Single Variable Regression)

The first experiment conducted was to measure the execution time difference for a single-variable model between the `Synapse Engine` and both Python models `LinearRegression` and `SGDRegressor`.

Figure 4.1 demonstrates the average runtime of each model training after 10 iterations in seconds. The figure shows a significant difference between the industry and education standard and the custom developed engine. From the figure, we can extract the following data:

- **LinearRegression**: averaged at `1.28 seconds` after 10 iterations.
- **SGDRegressor**: averaged at `1.02 seconds` after 10 iterations.
- **Synapse Engine**: averaged at `0.14 seconds` after 10 iterations.

This figure shows the critical performance improvements compared to the python equivalents. The Synapse Engine was 9 times faster than LinearRegression and 7 times faster than SGDRegressor.

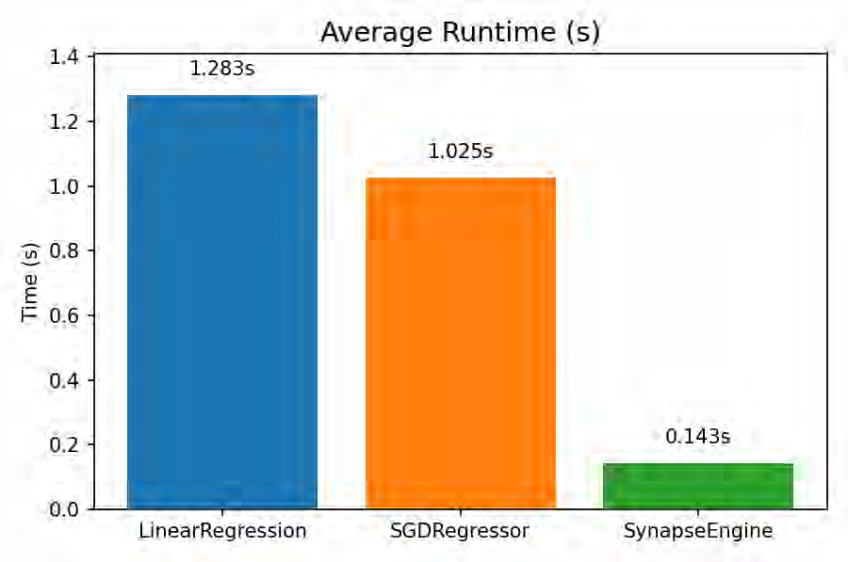

Figure 4.1: Runtime Benchmark (Single Variable regression). Source: Own creation

4.3.2 Memory Performance Comparison (Single Variable Regression)

The second experiment was conducted to measure the peak memory usage difference for the same single-variable model, again between `Synapse Engine` and Python's `LinearRegression` and `SGDRegressor`.

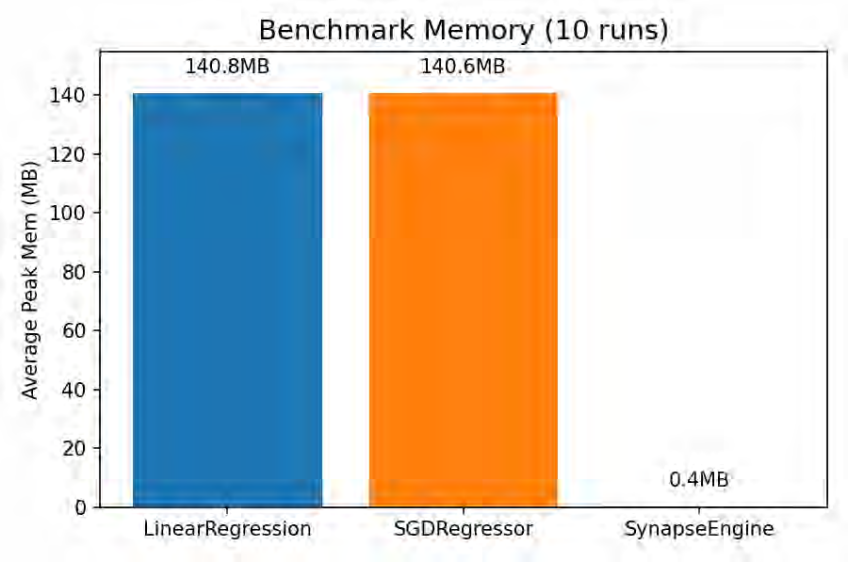

Figure 4.2: Memory Benchmark (Single Variable regression). Source: Own creation

Figure 4.2 demonstrates the average peak memory usage of each model in the training process after 10 iterations in MB. The figure shows a huge difference between the industry models and the custom developed engine. From the figure, we extract the following data:

- **LinearRegression**: averaged at `140.8 MB` after 10 runs.
- **SGDRegressor**: averaged at `140.6 MB` after 10 runs.
- **Synapse Engine**: averaged at `0.4 MB` after 10 runs.

The results of the experiments show a huge difference in the memory usage between the optimized engine and the industry way. From the extracted data, we can calculate that `Synapse Engine` uses `99.72%` less memory.

4.3.3 Accuracy Analysis

From the previous experiments, we know that `Synapse Engine` is faster and uses less memory than the equivalents. But the question that arises is: Does the actual performance accuracy compare to the actual value and the python models. In this part, we will look at the trained parameters and loss and compare them to the actual analytical solution.

4.3.3.1 Bias Estimation

First, we will explore the bias result from the trained model between `Synapse Engine` and Python's `LinearRegression` and `SGDRegressor`.

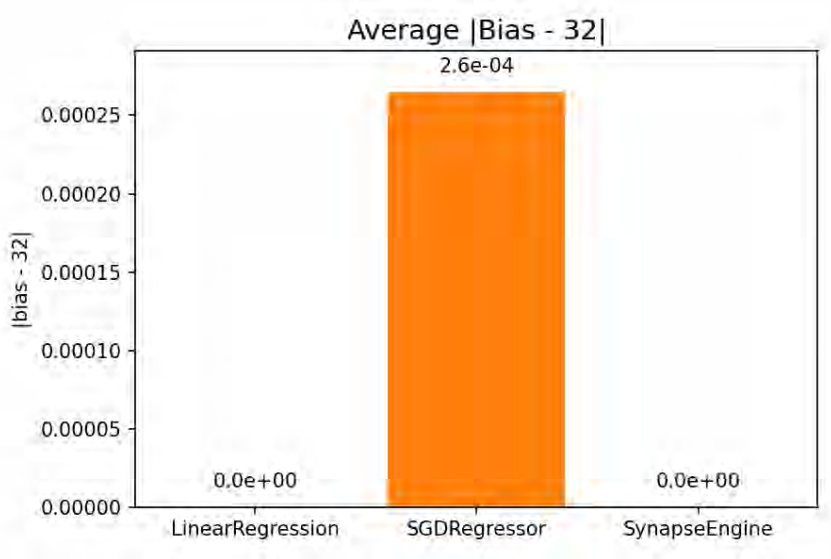

Figure 4.3: Bias Accuracy. Source: Own creation

Figure 4.3 shows the average absolute bias error $|Bias - 32|$ for each model. From the figure, we extract the following data:

- **LinearRegression**: averaged at `0.0` after 10 iterations.

- **SGDRegressor**: averaged at `2.6e-4` after 10 iterations.
- **Synapse Engine**: averaged at `0.0` after 10 iterations.

From the data of the figure, we can see that both the `LinearRegression` model and `Synapse Engine` model got perfect results, unlike `SGDRegressor` model.

4.3.3.2 Slope Estimation

Second, we will explore the slope results from the trained model between `Synapse Engine`, `LinearRegression`, and `SGDRegressor`.

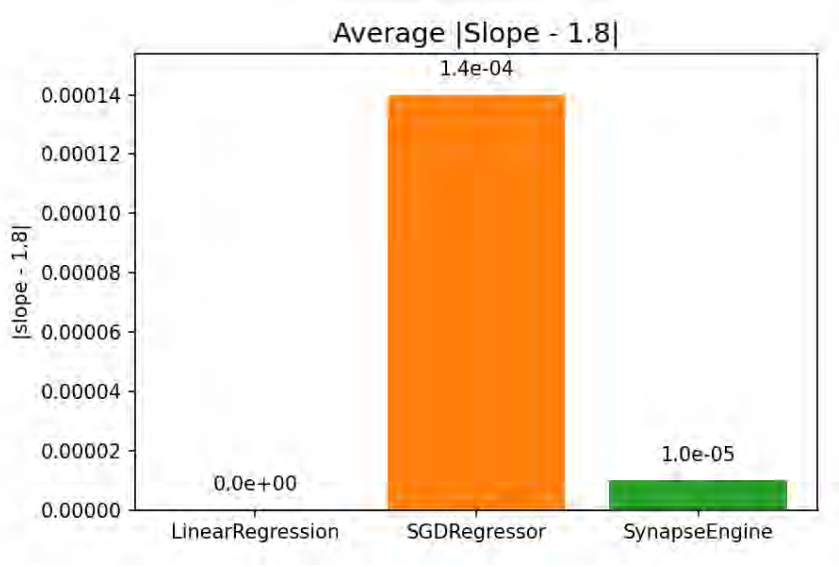

Figure 4.4: Slope Accuracy. Source: Own creation

Figure 4.4 shows the average absolute slope error $|Slope - 1.8|$ for each trained model. From the figure, we can extract the following data:

- **LinearRegression**: averaged at `0.0` after 10 iterations.
- **SGDRegressor**: averaged at `1.4e-4` after 10 iterations.
- **Synapse Engine**: averaged at `1.0e-5` after 10 iterations.

From the data of the figure, we see that the `LinearRegression` has got the perfect analytical solution, whilst the `SGDRegressor` had the highest slope error. `Synapse Engine` has a low loss compared to SGDRegressor, and it comes to around a floating point precision.

4.3.3.3 Loss Estimation

Finally, we will explore the average loss results from the training phase of all three models and compare them.

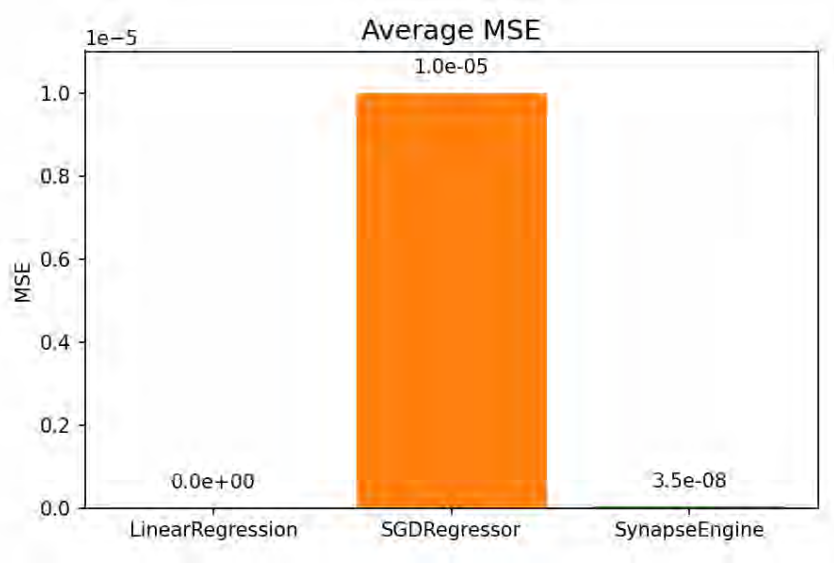

Figure 4.5: Loss Benchmark. Source: Own creation

Figure 4.5 shows the average loss of each model after 10 runs. From the figure, we can extract the following data:

- **LinearRegression**: averaged at `0.0` after 10 iterations.
- **SGDRegressor**: averaged at `1.0e-5` after 10 iterations.
- **Synapse Engine**: averaged at `3.5e-8` after 10 iterations.

From the data, we can see that `LinearRegression` had a perfect score due to the fact it had the perfect analytical solution. `SGDRegressor` had the highest loss on average, and the custom developed engine `Synapse Engine` had a very low loss.

4.3.4 Runtime Performance Comparison (Multi Variable Regression)

The fourth experiment conducted was to measure the execution time difference between `Synapse Engine` and `SGDRegressor`. This experiment was done on the `California House Pricing` dataset containing 20640 entries.

Figure 4.6 demonstrates the average runtime of each model training after 10 iterations in seconds. The figure shows a huge difference between the industry standard and the custom develop engine again. From the figure, we can extract the following data:

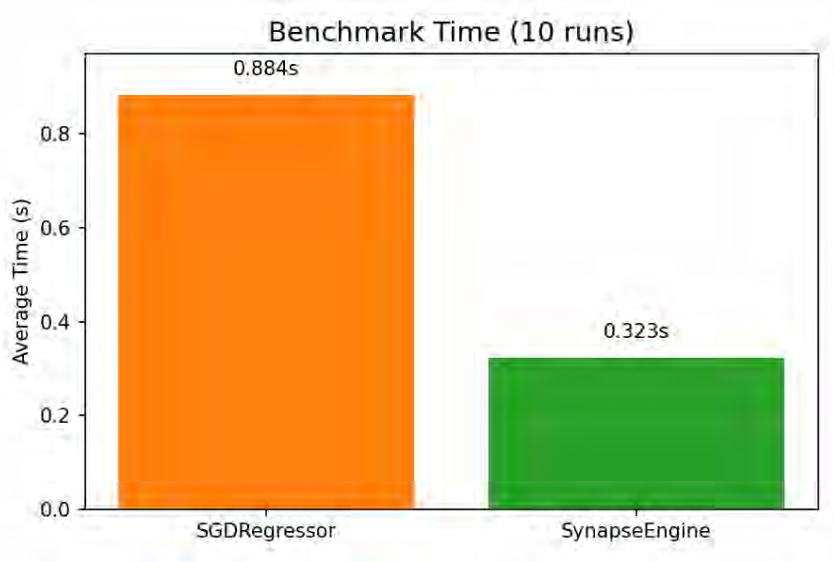

Figure 4.6: Runtime Benchmark (Multi Variable regression). Source: Own creation

- **SGDRegressor**: averaged at `0.88 seconds` after 10 runs.
- **Synapse Engine**: averaged at `0.32 seconds` after 10 runs.

From the data, we can see that the custom developed engine is about 3 times faster than the equivalent.

4.3.5 Memory Performance Comparison (Multi Variable Regression)

The fifth and final experiment that was conducted measures the peak memory usage difference for the same multi-variable model, again between `Synapse Engine` and `SGDRegressor`.

Figure 4.7 demonstrates the average peak memory usage of each model in the training phase after 10 iterations in MB. The figure shows a significant difference between the two models. From the figure, we extract the following data:

- **SGDRegressor**: averaged at `150.2 MB` after 10 iterations.
- **Synapse Engine**: averaged at `22.8 MB` after 10 iterations.

The results of the experiment shows a big gap in the memory usage between the optimized custom developed engine and the industry way. From the extracted data, we calculated that `Synapse Engine` uses `84.83%` less memory.

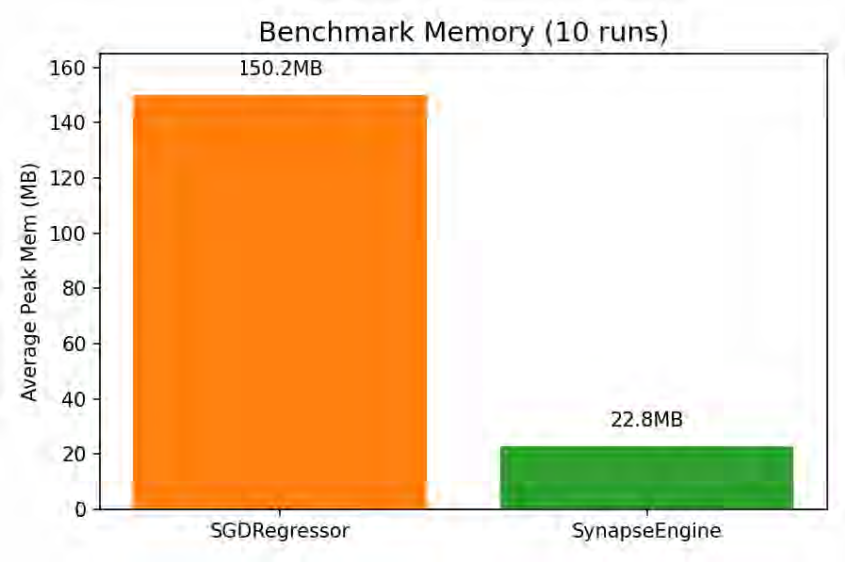

Figure 4.7: Memory Benchmark (Multi Variable regression). Source: Own creation

4.4 Conclusion

The evaluation demonstrates that `Synapse Engine` offers considerable advantages in performance efficiency, low memory footprint, and prediction accuracy compared to traditional industry ways. It matches, and sometimes beats, the traditional algorithms such as `LinearRegression` and `SGDRegressor`. These attributes make Synapse Engine highly suitable for use by professionals and educators, and the performance gains especially in memory makes it suitable for deployment on resource-constrained and performance-critical systems such as real-time applications and embedded systems.

Chapter 5

CONCLUSIONS

5.1 Conclusion

The `Synapse Project` composed of `Synapse Engine` and `Synapse Kernel` have successfully fulfilled its mission: enabling efficient and easy AI workflow on embedded and resource-constrained systems. Through the two components, the project lowers the entry barrier of powerful machine learning to newcomers and professionals.

At its core, the Synapse Engine, a stand-alone AI Engine, aims to lower the entry barrier for newcomers, researchers, and provide high performance for resource-constrained systems. To achieve the highly performant yet accurate results that compare and sometimes beat the industry standard way, the engine was developed into modules using different systems technologies. The final product is a static binary that contains all the functionalities needed, from parsing the initial data to training the model and outputting a human-readable format for the final parameters.

To complement the design philosophy of the AI Engine, a custom DSL has been developed from scratch to make it easier for newcomers and educators to quickly create their first model without the need to learn a programming language and many tools and frameworks.

The Synapse Kernel serves a complement to Synapse Engine. It provides an alternative for embedded and resource-constrained systems. The kernel is optimized for ARM-based systems. The kernel architecture also serves as a helper for hardware-software AI acceleration, unlike traditional kernels that only provide general purpose memory management, this custom kernel implements another hardware-software memory subsystem for AI usage only. This new architecture proves to be efficient and helps with the AI optimizations.

After developing the project, I have designed an experiment series to evaluate the performance of the AI engine and compare it to the standard way of implementing the same algorithms for students, hobbyist, or industry professionals. After running the evaluation process, the Synapse Engine has proven to be high-performance and

accurate. In some cases it matches the analytical solution and beats the standard algorithm. Whether it is the raw speed or the peak memory usage, the AI Engine has proven to be much more performance and thus make it suitable for resource-constrained systems.

While the project has proven its capabilities, it still has many limitations in its current form. Mainly, the engine has only one algorithm implemented and the kernel has still some bugs and drivers to be developed in order to bring the full potential of the project.

During the development, I have faced many challenges that were a great learning experience for me. Some were easy and were fixed right away, and some were harder. One of the primary challenges I faced during the development of this project was the lack of ARM kernel development resources. Unlike the x86 architecture which was more popular, ARM has only gained the light recently and this posed a problem because there wasn't any easy resources on the development for ARM-based systems. To solve this, I have used the official ARM documentation extensively with the help of previous x86 OS development and reading through some ARM embedded projects. The other big challenge I faced was the design of the DSL to be easy yet flexible and extendable. This issue was solved by iterating over many designs of the language and gaining feedback over each detail before settling on the final implemented design.

5.2 Future Work

Even with the achievements made by the project, it still needs improvements. There are many promising directions for future update and development that could enhance the performance and ease of the project:

- The Math Engine should make more use of Single Instruction Multiple Data (SIMD) operations provided by some hardware architectures for improved and faster performance.
- More algorithms should be implemented into the AI Engine, possibly including advanced ML and DL.
- The AI Engine could have a GUI to make it even easier and more appealing to newcomers and hobbyists.
- The kernel should implement more drivers to be able to work on more embedded hardware such as SD Card storage, USB, and a user interface.
- The kernel should make use of more memory protection from the hardware when possible, such as paging, Memory Protection Units (MPUs), and Memory Management Units (MMUs).
- The kernel and engine should make use, whenever possible, of the integrated GPU and the Neural Processing Unit (NPU).

- The kernel should implement developer tool chains to further extend the capabilities of the kernel and the AI Engine.

In summary, while many challenges were a roadblock at first and the cheer amount of knowledge compiled for the development of this project took some time, the final product was developed and proved the benefit of this new design architectures for sustainable AI in the real-world.

Bibliography

[1] United Nations. Sustainable Development, 2025. URL www.un.org. Accessed: 17-07-2025.

[2] Muthappa Satwik. Stochastic Gradient Descent (SGD), 2020. URL https://www.linkedin.com/pulse/stochastic-gradient-descent-sgd-satwik-muthappa-p-ph-d-. Accessed: 17-07-2025.

[3] Spyros Makridakis. The forthcoming artificial intelligence (ai) revolution: Its impact on society and firms. *Futures*, 90:46–60, 2017.

[4] Carole-Jean Wu, Ramya Raghavendra, Udit Gupta, Bilge Acun, Newsha Ardalani, Kiwan Maeng, Gloria Chang, Fiona Aga, Jinshi Huang, Charles Bai, et al. Sustainable ai: Environmental implications, challenges and opportunities. *Proceedings of machine learning and systems*, 4:795–813, 2022.

[5] Ajay Bandi, Pydi Venkata Satya Ramesh Adapa, and Yudu Eswar Vinay Pratap Kumar Kuchi. The power of generative AI: A review of requirements, models, input–output formats, evaluation metrics, and challenges. *Future Internet*, 15 (8):260, 2023.

[6] David Mhlanga. Artificial intelligence and machine learning for energy consumption and production in emerging markets: A review. *Energies*, 16(2):745, 2023.

[7] Joachim Von Braun and Heike Baumüller. Ai/robotics and the poor. In *Robotics, AI, and Humanity: Science, Ethics, and Policy*, pages 85–97. Springer International Publishing Cham, 2021.

[8] Muhammad Nabeel Asghar. A review of ARM processor architecture history, progress and applications. *Journal of Applied and Emerging Sciences*, 10(2): pp–171, 2020.

[9] Eben Upton and Gareth Halfacree. *Raspberry Pi user guide*. John Wiley & Sons, 2016.

[10] Gerald Coley. Beaglebone black system reference manual. *Texas Instruments, Dallas*, 5:2013, 2013.

[11] Patrick Paul Walsh, Aparajita Banerjee, and Enda Murphy. The un 2030 agenda for sustainable development. In *Partnerships and the sustainable development goals*, pages 1–12. Springer, 2022.

[12] Mbugua Samuel Thaiya, Korongo Julia, and Samuel Mbugua. On software modular architecture: concepts, metrics and trends. *International Journal of Computer and Organization Trends*, 12(1):3–10, 2022.

[13] Ken Schwaber. Scrum development process. In *Business object design and implementation: OOPSLA'95 workshop proceedings 16 October 1995, Austin, Texas*, pages 117–134. Springer, 1997.

[14] Youssef Bassil. A simulation model for the waterfall software development life cycle. *arXiv preprint arXiv:1205.6904*, 2012.

[15] Earl B Hunt. *Artificial intelligence*. Academic Press, 2014.

[16] Matthew G Hanna, Liron Pantanowitz, Brian Jackson, Octavia Palmer, Shyam Visweswaran, Joshua Pantanowitz, Mustafa Deebajah, and Hooman H Rashidi. Ethical and bias considerations in artificial intelligence/machine learning. *Modern Pathology*, 38(3):100686, 2025.

[17] Ethem Alpaydin. *Machine learning*. MIT press, 2021.

[18] Zhi-Hua Zhou. *Machine learning*. Springer nature, 2021.

[19] Freek Stulp and Olivier Sigaud. Many regression algorithms, one unified model: A review. *Neural Networks*, 69:60–79, 2015.

[20] Xiaogang Su, Xin Yan, and Chih-Ling Tsai. Linear regression. *Wiley Interdisciplinary Reviews: Computational Statistics*, 4(3):275–294, 2012.

[21] Eva Ostertagová. Modelling using polynomial regression. *Procedia engineering*, 48:500–506, 2012.

[22] Werner H Greub. *Linear algebra*, volume 23. Springer Science & Business Media, 2012.

[23] Gholamreza Hesamian, Faezeh Torkian, Arne Johannssen, and Nataliya Chukhrova. A learning system-based soft multiple linear regression model. *Intelligent Systems with Applications*, 22:200378, 2024.

[24] Torben Ægidius Mogensen. *Introduction to compiler design*. Springer Nature, 2024.

[25] William Stallings. *Operating systems*. Prentice Hall PTR, 2000.

[26] Harvey M Deitel, Paul J Deitel, David R Choffnes, et al. *Operating systems*. Pearson/Prentice Hall, 2004.

[27] Randall Hyde. *The art of assembly language*. No Starch Press, 2003.

[28] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. Pearson Educación, 1988.

[29] Harvey M Deitel and Paul J Deitel. *C++ how to program*. Pearson Educación, 2003.

[30] Brad J Cox. *Object oriented programming: an evolutionary approach*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[31] Robert Love. *Linux kernel development*. Pearson Education, 2010.

[32] Tom Shanley. *x86 Instruction Set Architecture*. MindShare press, 2010.

[33] David Seal. *ARM architecture reference manual*. Pearson Education, 2001.

[34] Zig contributors. In-depth Overview - Zig Programming Language, 2025. URL https://ziglang.org/. Accessed: 18-07-2025.

[35] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.

[36] Oliver Kramer. Scikit-learn. In *Machine learning for evolution strategies*, pages 45–53. Springer, 2016.

Appendix A

INSTALLATION MANUAL

The installation manual serves as a guide to get the synapse engine up and running on your machine. To facilitate the user installation at early stages and to stay in line with the ease-of-use guideline, the project has been deployed to docker and to the homebrew package manager. In the following sections, I will explain how to get the project up and running with both tools.

A.1 Prerequisites

Before beginning the installation process, make sure you either have docker installed on your machine or the homebrew package manager installed.

- For Windows, Mac, and Linux users, you can use docker as it provides both the arm64 and amd64 versions of the project.
- If you have a Mac and you prefer running it natively, you must install the homebrew package manager before proceeding.

A.2 Docker setup

After installing docker and making sure it is running, all you need to do to install and run the synapse project is the following two commands:

1. `docker pull fedinabli/synapseengine:latest`
2. `docker run -it fedinabli/synapseengine:latest`

After running these two commands, docker will pull the engine image and run, puts you into a Linux environment with the program installed. It will also put you in

a `EngineTest` folder that contains the Celsius to Fahrenheit and California House pricing datasets and configuration (SYNJ) files. Now you can start typing the command to create and use your first model.

A.2.1 Docker Hub

The image is public and available at docker hub, with detailed instructions that will guide you in the project.

Docker Hub Link: https://hub.docker.com/r/fedinabli/synapseengine

A.3 Homebrew

Homebrew is a macOS package manager used to install and manage the packages at your system. I have published the native binary for the AI Engine on homebrew for both intel and Mas (x86) and M series (ARM) processors.

To install the binary globally on your system, you need to run the following two commands in order:

1. `brew tap fedi-nabli/synapse`
2. `brew install synapse`

After running these two commands in order, you will have the binary installed globally in your system. Now you can proceed to create your first model and using it.

Appendix B

USER MANUAL

B.1 Synapse Engine Description

The `Synapse Engine` is a modular, performant, and efficient AI ML engine that streamlines the workflow for creating, training, and running inference on models with a custom DSL. The project is a self-contained binary program that makes it easy to install and get to creating the first model in a matter of seconds.

B.2 General Commands

The project does not include any GUI as of this version, but provides an easy-to-use CLI tool with custom logging and many commands. When you first install the engine, there are two general commands that can help you navigate and understand the different commands that the tool offers.

B.2.1 Help Command

Figure B.1: Help Command Source: Created by me

Figure B.1 shows the first general command: The `help` command. This command outputs a message detailing the commands, their usage, and the required parameters

if exists. To get the help message, you can simply type `synapse help` in your terminal of choice.

B.2.2 Version Command

Figure B.2: Version Command Source: Created by me

Figure B.2 shows the second general command: The `version` command. This command prints details about the currently installed version of the synapse project, such as the version number and the build date. To get version details, simply run `synapse version` in your terminal.

B.3 Model Training

To train a model, the CLI tool offers a `train` command that take in the configuration file (SYNJ) and optional flags to start the training process and outputting the final model data.

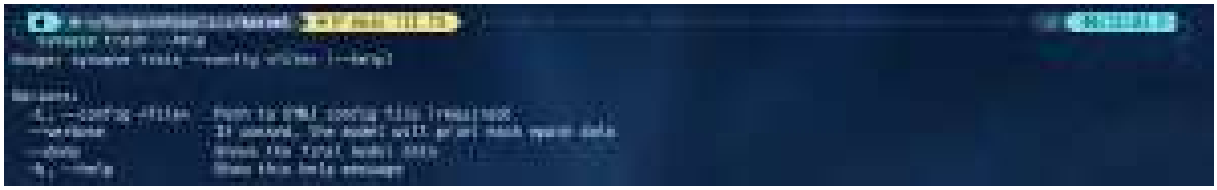

Figure B.3: Train Help Command Source: Created by me

Figure B.3 shows the help flag for the train command in action. The train commands accept a configuration file and two optional flags for showing the final model data and outputting each iteration data while it is training. To get the help message, run `synapse train --help` in your terminal.

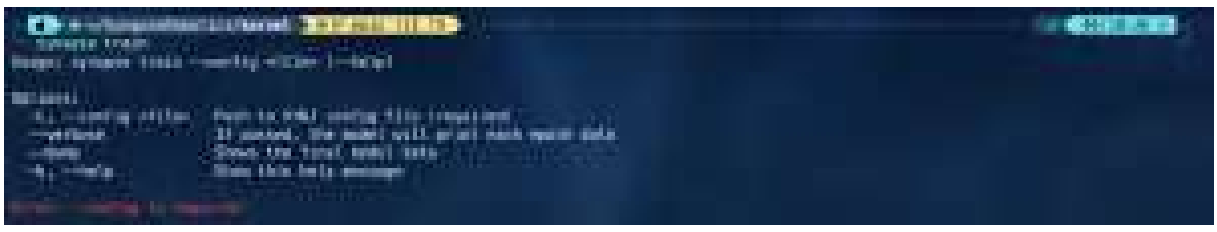

Figure B.4: Train Error Command Source: Created by me

In case the train command didn't get the required argument and the help flag was not provided, the command is invalid and thus fails, as Figure B.4 shows. In case of failure, the CLI tools show the help message with the error indicated below as to show what was the problem.

To successfully train a model, the configuration file and the data files must be provided. Figures B.5 and B.6 respectively show a valid SYNJ configuration and an example dataset for converting Celsius to Fahrenheit.

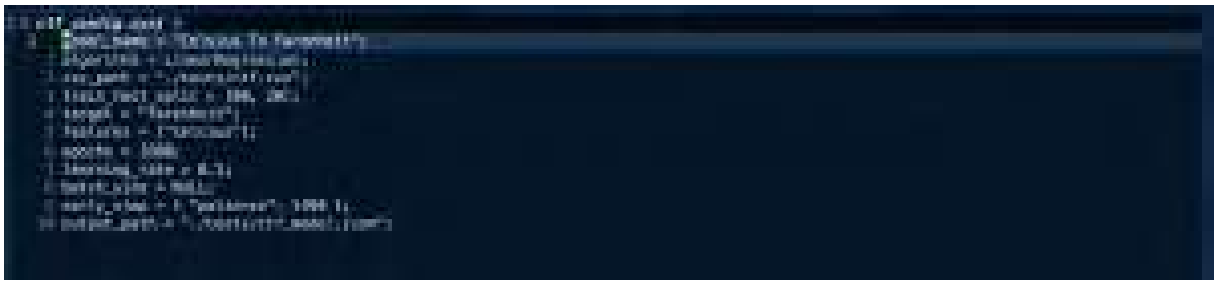

Figure B.5: Train Configuration File Source: Created by me

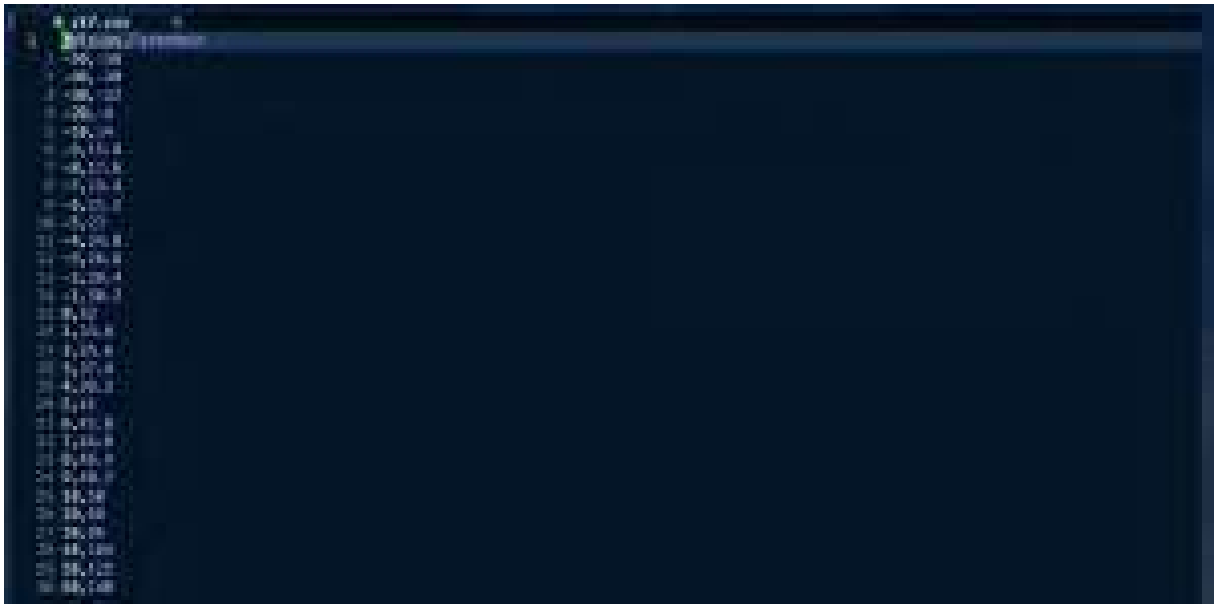

Figure B.6: Train Data File Source: Created by me

After writing the configuration and data files into your favourite text editor, you can run the full training command with the path to the configuration file and your wanted options. In the example provided in Figure B.7 I have opted to use the `--dump` option to show the final model data. The full correct command now is `synapse train -C ./config.synj --dump`. The figure shows that the training process was successful and outputted the final model data.

Finally, after the training is done, you can check the JSON file containing the data to use it later or to validate the output. Figure B.8 shows the final JSON file saved to the location provided in the configuration file. We can see that the data matches from the train command output. The terminal's stdout formatter rounds a very large number, and this is why we see some very small precision point mismatch.

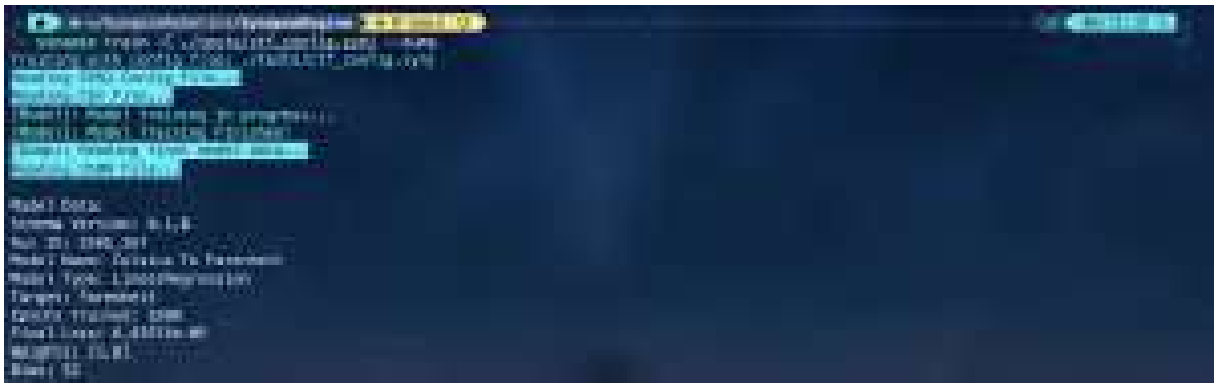

Figure B.7: Train Success Command Source: Created by me

Figure B.8: Train Output File Source: Created by me

B.4 Inference on a Trained Model Commands

To run inference on a trained model, the CLI tool offers a `predict` command that take in the outputted file (JSON) from the previous step and the input data to run the prediction on.

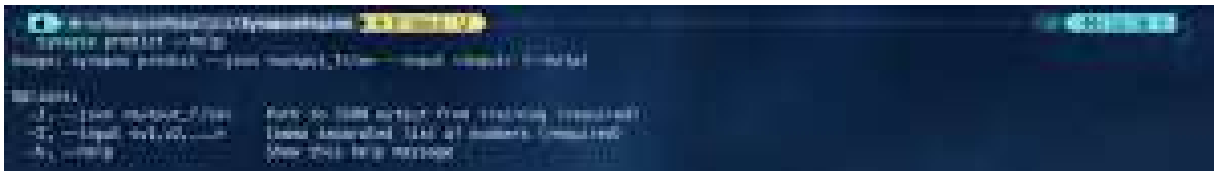

Figure B.9: Predict Help Command Source: Created by me

Figure B.9 shows the help flag for the predict command in action. The predict commands accepts the final model data file from the training process and the input flag that contains the data to run the model inference on. To get the help message, run `synapse predict --help` in your terminal.

In case the predict command didn't get the required arguments and the help flag was not provided, the command is invalid and thus fails, as Figure B.10 shows. In case of failure, the CLI tools show the help message with the error indicated below as to show what was the problem.

To successfully run the predict command, you must provide the path to the model data and your new data to get the results. Figure B.11 shows the final complete predict command and the output. Here the full command looks like `synapse predict -J`

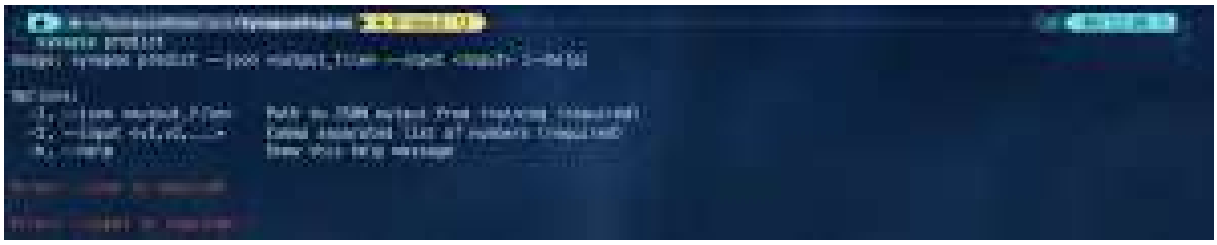

Figure B.10: Predict Error Command Source: Created by me

`./ctf_model.json -I 70`. After running, the command prints the predicted value and exits.

Figure B.11: Predict Success Command Source: Created by me

B.5 Viewing Model Details Commands

The final command offered by the CLI tool is the `dump` command. This command is used to see the final model data formatted in the CLI for a better user experience.

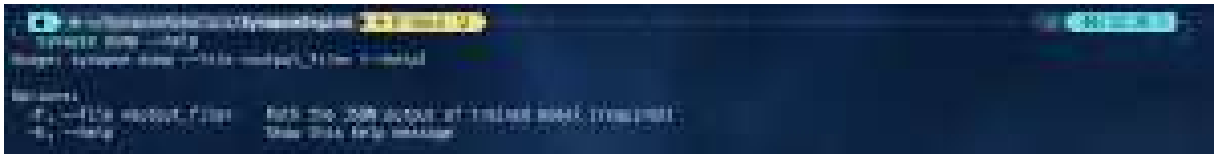

Figure B.12: Dump Help Command Source: Created by me

Figure B.12 shows the help flag for the dump command in action. The predict commands accepts the final model data file from the training process. To get the help message, run `synapse dump --help` in your terminal.

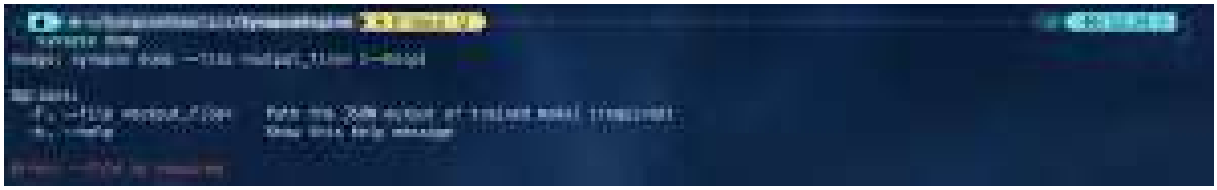

Figure B.13: Dump Error Command Source: Created by me

In case the dump command didn't get the required argument and the help flag was not provided, the command is invalid and thus fails, as Figure B.13 shows. In case of failure, the CLI tools show the help message with the error indicated below as to show what was the problem.

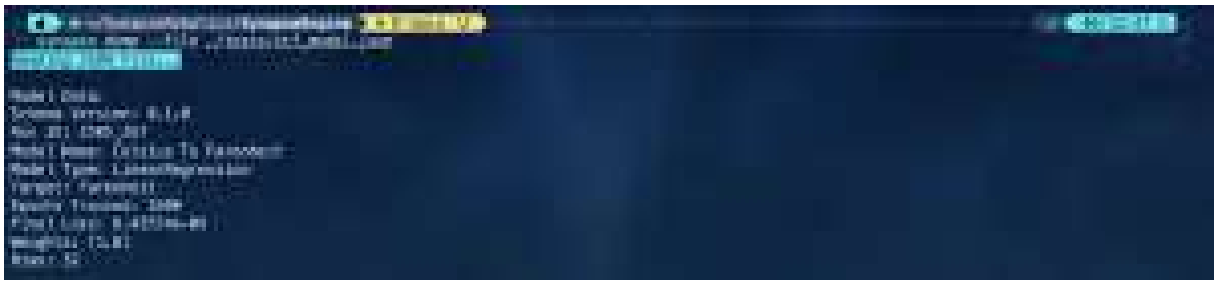

Figure B.14: Dump Success Command Source: Created by me

To successfully run the dump command, you must provide the path to the model data file. Figure B.14 shows the final complete dump command and the output. Here, the full command looks like `synapse dump --file ./ctf_model.json`. After running, the command prints the user-friendly formatted model data and exits.