System Design Document

for

Driving Functions in autonomous Truck

Version 1.1 approved

Prepared by Marc Pletz, Marcel Schad, Samuel Schwarz and Johannes Spranger

Gamble GmbH

03.03.2025

Revision History

Name	Date	Reason For Changes	Version
	27.02.25	Basic Setup of the Template	1.0
Final Version	03.03.25	Final Version of the SRS	1.1

1. Introduction

1.1 Purpose

This document describes the system design and architecture for a lane and obstacle detection as well as a Safe-Remote-Control, Platooning and Path-Planning feature as subsystems of an ZF-DHBW Inno Truck. It should be used as a guide to ensure a reliable, safe and efficient system during the software implementation.

1.2 Design Goals

The design of the truck's software architecture is driven by several key goals to ensure reliability, efficiency, and safety in autonomous operation according to ISO 26262 [1]. These goals define the foundation for system decisions and architectural choices:

1. **Reliability & Fault Tolerance**

   The system must function reliably under all operating conditions. It should be able to detect, handle, and recover from faults to prevent system failures or unsafe behavior. The architecture should include mechanisms for redundancy and error handling, aligning with AUTOSAR's fail-operational concepts [2].

2. **Real-Time Performance**

   Since autonomous driving requires rapid decision-making, the software must process sensor data and generate control signals with minimal and deterministic latency. Efficient data flow and computational optimization are essential.

3. **Modularity & Scalability**

   The architecture should be designed in a modular way, allowing for easy integration of new sensors, algorithms, and subsystems. It must also support future expansions and upgrades without requiring major rework.

4. **Safety & Compliance**

   As a critical system in an autonomous vehicle, safety is the highest priority. The software must comply with industry standards and regulations for functional safety, such as ISO 26262 [1]. The architecture should incorporate safety mechanisms, such as AUTOSAR's functional safety guidelines [2], ensuring safe operations even in edge cases.

5. **Efficient Resource Utilization**

   The system must optimize the use of computational resources, including processing power, memory, and network bandwidth, to ensure stable operation on the embedded hardware of the truck.

6. **Maintainability & Extensibility**

   The software should be structured in a way that facilitates debugging, testing, and future development. Clear interfaces, well-documented code, and adherence to software engineering best practices are crucial.

By focusing on these design goals, the truck's software architecture ensures a robust, efficient, and scalable foundation for autonomous driving.

1.3 Definitions, Acronyms, and Abbreviations (Glossary)

This section defines key terms, acronyms, and abbreviations used throughout the document to ensure clarity and consistency.

Term	***Definition***
ADAS	Advanced Driver Assistance System – A system that assists the driver or automates driving tasks.
ROS	Robot Operating System – A flexible framework for writing robot software, used for communication between subsystems.
Pipe-and-Filter Architecture	A software design pattern where data flows through a series of processing stages (filters) connected by channels (pipes).
Repository Architecture	A software design pattern that uses a central storage unit where all subsystems read and write data.
Lidar	Light Detection and Ranging – A sensor technology that measures distances by illuminating targets with laser light.

1.4 References

This section lists all documents, standards, and external sources referenced in the system design document.

- [1] ISO 26262 – Functional Safety Standard for Road Vehicles
- [2] AUTOSAR Adaptive Platform Specification

2. Proposed Software Architecture

2.1 Overview

The overview diagram illustrates the high-level architecture of the autonomous truck system, highlighting the key components and their interactions. The system collects data from multiple sensors through the Sensor Interface, which is then processed in the Sensor Data Processing module. The processed data is utilized by Autonomous Functions to generate control commands, which are transmitted through the Truck Control Unit to the ROS Bridge, enabling communication with the vehicle's actuators. Additionally, a User Interface provides monitoring and interaction capabilities, allowing supervision while maintaining autonomous operation. This modular structure ensures efficient data flow, real-time processing, and seamless integration of various system components.

2.2 Subsystem decomposition

2.2.1 Software Architecture

The software architecture closely resembles a classic pipe-and-filter architecture. There is minimal active interaction, as the truck's ADAS processes sensor data into control signals, which are then sent to the ROS bridge. The truck itself has little direct influence on its environment—except in the rare event of a malfunction leading to an accident, which should not occur. However, the environment does impact the truck, as its trajectory planning and control signals are based on environmental inputs. This unidirectional interaction aligns with the fundamental definition of a pipe-and-filter architecture, where incoming data serves as the system's input. Since the truck's control unit functions primarily as a data processing system, the pipe-and-filter architecture is well-suited to its software design and therefor applied in the truck system.

Of course, other software architectures could have been considered for the software design, such as the Repository Architecture. In this approach, a central storage unit is used, which can be divided into different storage sections. Various types of processed data are assigned to specific storage sections using pointers. For example, bitmaps, dotmaps, and obstacles would each have their own allocated storage sections of the required size. Subsystems can access the necessary input variables via pointers to these specific memory areas and, after processing, store the results in the next section of the repository.

However, a potential drawback of this architecture is that memory must be pre-divided into sections during implementation, which significantly limits scalability compared to the Pipe-and-Filter Architecture. Additionally, performance is a critical factor in the autonomous systems of the truck. Unlike the Pipe-and-Filter Architecture, performance in a Repository Architecture may degrade if multiple subsystems simultaneously request and store data in the repository, leading to potential bottlenecks.

2.2.2 Additional Information to Subsystem decomposition

Subsystem Alignment Structure (see Subsystem Decomposition Diagram)

The subsystems of our system are aligned after the following principle: The main data flow goes from the left (sensor interface) over the data processing units (e. g. Lane Detection, Obstacle Detection or Remote Control Interface) and the decision units (e. G. Lane Following, Emergency Stop (Upon Detected Obstacle) or Safe Remote Control, which create control signals for the Truck to finally the Truck Control Unit on the right.

Data Transmitting (also seen in Global Software Control)

There has been discussed an alternative for the data flow described above. Get-Methods would have been possible to implement. So, the Truck Control Unit had to call the public get-methods of its left neighbored subsystems to get its instructions. In the get-methods of the Truck Control Units left-neighbored subsystems would be called their left-neighbored subsystems get-methods, in which would be called the get-methods of their left-neighbored subsystem again. Finally, the decision was to call the right-neighbored subsystems public start-processing-method and deliver a pointer on the data-structure to process further. Technically both principles would have worked. However, the start-processing method seemed to be more comfortable with multi-threading as the cpu power is not all used in the margin of the Truck Control Unit class. Also, prioritizing the different control signal from the different autonomous function units is easier with the left-to-right push data flow principle.

The Truck Control Unit

The Truck Control Units main task is to handle all various control signals. The autonomous function call the method start_processing_instruction() and deliver the parameters instruction type, instruction value and the instructions priority. A priority is needed if there is an instruction conflict. If the Remote Control's signal is acceleration, but the emergency stop (upon detected obstacle)'s signal is to brake harshly because an upcoming truck, the Truck Control should clearly decide for one instruction. In the start_processing_instruction() method there is created an instruction with priority, timestamp and message. The message is sent to the ROS interface, if the created instruction encounters no other instruction with the same instruction type and an instruction age of below 250 ms and a higher priority. These 250 ms is the reaction time on partial system failures (see more in Error Handling). After the sending to the ROS bridge, the sent message is kept in the instruction buffer for the next instructions to compare with instructions of the same type.

Truck State

Like the subsystem Error Handling, every subsystem has access to information about the truck's actual parameters. Speed, loading/weight and the steering angle are important to know for various classes. The access is granted with the get-method that transfers a pointer on the truck state. In the subsystem decomposition diagram, there is the referring between the truck state and other subsystems not shown to avoid a tremendous disorder.

Sensor Interface

We use different sensors than those of the Document Chapter_ProgramDesign_SRS. As a result, our sensor interface mapping from sensor to the next sensor data processing unit is also different than proposed in Chaper_ProgramDesign_SRS. The Positioning Unit uses the sensors IMU, UWB and the Wheel Steering Encoder, the Lane Detection suffice its Lane Camera and the Object Detection gets its information to the bitmaps of the Front Camera and the dotmaps of the 3D LiDAR.

Error_Handler

The Error_Handler is connected to every main class in every subsystem. It does not check data for errors itself. It only gets called if a class detects an error by itself. If an error is detected, the process_error() function of the Error_Handler will be called. All errors will be saved with their id, the error message (error description) and the error number (error id) in the error_list. After an error occurs, it will forward it to the user interface to inform the operator of the truck with the forward_error_to_ui() function. To allow the development team to investigate the error and under which circumstances it occurs, all data shall be saved in a persistent memory. To achieve this data saving, the Error_Handler calls its persistent_store_all_data() function. This will call an extra function for every class. Each class will save the data at a set memory address. The Truck_Control_Unit needs to know about every error that has occurred up to a specific time to start up the system. To get the errors, the get_errors() function is implemented, which returns a pointer to the first element of the error_list.

The Error_Handler shall never be turned off, so it is able to react to errors, that occur when the system is starting or restarting. It has a status (bool) which is set to 1 if the Error_Handler is active and 0 if it is deactivated. This status can be updated with the set_status() function. It needs to be deactivated so power can be saved and the CPU usage can be reduced. Another benefit of a turned off Error_Handler is that it would not react to any errors that may occur while turning off the system.

2.3 Hardware/Software Mapping

Literally every Software Unit of the system is implemented on the Intel NUC / Nvidia Jetson as shown in the Deployment Diagram. Some artifacts in the LAN router device just shall give an image of the routing software implementation, but do not describe the software routing implementation in any case. To simplify the connections in-between the sensors

UWB, IMU and the wheel steering encoder, the connections between the sensor devices and the LAN router is very reduced on to a LAN connection. The Arduino Mega or SPI busses are left out to keep the understanding easy. Nevertheless, the software engineers do not have to worry about the hardware device connection as it is not part of our system.

2.4 Persistent Data Management

Types of Data

An autonomous vehicle requires extensive data storage. This includes vehicle-specific data such as error logs, charging or fuel levels, as well as environmental data like navigation, maps, and traffic rules (e.g., speed limits). These must be stored before a system shutdown to ensure continued functionality. Files are timestamped since non-volatile memory is limited, and older data is less critical. When the memory is close to full, the oldest, least important entries are overwritten to make space for new ones.

Address Management

The memory address space is divided into multiple sections of varying sizes, each optimized for a specific data type (e.g., error logs, charging status, fuel levels). Each section has a reserved starting address that keeps track of the next available storage location. New data is written to this address, and after writing, the reserved address is incremented. Once the reserved address reaches the next section's boundary, it resets to the start of its section, overwriting older data. To maintain system integrity, all reserved section addresses are stored in a dedicated memory area.

When is Data Persistently Stored?

For autonomous driving functions, minimal persistent storage is needed - typically just the last 10 lane frame calculations for distances between 1m and 10m ahead. This ensures the vehicle can resume movement correctly after a restart. Additionally, it is crucial to determine whether the vehicle moved after the last lane frame calculations (e.g., if it was manually driven). If the route has changed significantly, previous calculations become obsolete. This information is saved before system shutdown, particularly when the vehicle is expected to be stationary for an extended period. In case of errors or before resetting the autonomous system, specific data must be permanently stored, depending on the error type.

This may include:

Error codes,

Latest bitmaps or dot maps,

Lane frames and detected obstacles,

Truck Control Unit (TCU) decisions,

Braking force,

Steering angles,

Position

to support diagnostics.

To support a system recovery after an restart, the position needs to be stored before every kind of system shutdown to ensure that the Feature Positioning and Path Planning have enough information to the current truck’s destination. Of course, other type of data, like fuel level have to be stored before also, but this information is not relevant for the system that the programmers have to implement.

Raw sensor data is not stored - only data that has already been transmitted to the receiving buffer of the USB or LAN interface is saved, as the sensors themselves are outside the system's scope. Diagnostic data is also transmitted via LAN to an external UI host PC, but its storage strategy is beyond this implementation. This strategy ensures that only essential, high-priority data is persistently stored while optimizing limited non-volatile memory.

2.5 Global Software Control

The software processes various sensor input data to control signals. During one system cycle the information has multiple data types, depending on the current subsystem connection. The software can be divided into the software modules sensor interface, sensor data processing, autonomous function unit and truck control. In the beginning, the sensor passes its data via USB or LAN to the Intel NUC Receiving Register Buffer. The information handling of this buffer is determined in the sensor interface. After having recreated the sensors data and saved it as a data type like Bitmap array (for the camera sensors), Dotmap array (for the LiDAR), UWB position, etc. the sensor interface main class calls the public start_processing_data functions of its in the subsystem diagram right-neighbored subsystems and delivers a pointer on the sensors data object of the current system cycle. There is a very detailed description of the on going data processing in the two Global Software Control sequence diagrams of the EA model. To avoid a stack overflow, data objects get deleted after having been processed, under the condition that no error had occurred.

2.6 Boundary Conditions

2.6.1 Startup

A User Interface starts the system with the call of the “system_startup()” function. The Truck_Control_Unit activates the Error_Handler and then creates all Objects. The Error_Handler gets created first so it can check for initialization errors. The Truck_Control_Unit checks for errors before ending the startup process. If no critical error occurs the system will end the startup process. If an error occurs, that could prevent the start of the system, it will shut itself down. The errors are automatically logged by the Error_Handler. The Truck_Control_Unit needs to check for logged errors, because a false initialized object might not call the Error_Handler itself. False initialization is checked with a test sequence, which must always deliver the same results.

2.6.2 Shutdown

The shutdown process can be triggered either through the User Interface or directly by the Truck Control Unit. To initiate the shutdown, the system_shutdown() function is called, after which the Error_Handler status is set to 0 (deactivated). All utilized objects are automatically terminated as they go out of scope. Since no data needs to be preserved, the system powers down without requiring any additional saving operations.

2.6.3 Error Behaviour

The Lane Following system continuously monitors its data for errors. When an issue is detected, it calls the Error_Handler, providing both the error ID and a corresponding error message. The Error_Handler then stores all relevant data in persistent memory, enabling the development team to analyze the root cause of the error. To inform the operator, the error is also forwarded to the User Interface.

Next, the Error_Handler evaluates the severity of the issue. If the error is non-critical, the truck continues operating as usual without interruption. However, if the error is critical, the truck initiates an emergency braking procedure and prepares for a system restart. The Error_Handler sends an instruction to the Truck Control Unit, which monitors the vehicle's state and ensures the speed is reduced to zero. Once the truck has come to a complete stop, the system shuts down and then automatically restarts.