Department of Aerospace and Geodesy
TUM School of Engineering and Design
Technical University of Munich

Development of an Operation Control System PLC Software for a Hyperloop Demonstrator

Entwicklung einer Steuerungssoftware Architektur für einen Hyperloop Demonstrator

Scientific work for obtaining the academic degree
Bachelor of Science (B.Sc.)
at the TUM School of Engineering and Design of the Technical University of Munich

Supervisor Prof. Dr.-Ing. Agnes Jocher
Department of Aerospace and Geodesy

Advisor Domenik Radeck, M.Sc.
Department of Aerospace and Geodesy

Submitted by Hannes Pravida
Clemensstraße 125
80796 München
+49 178 402 1521

Submitted on February 02, 2023 in Garching

Abstract

In this bachelor thesis, we develop an architecture for a PLC control software. The work is situated in the context of the TUM Hyperloop project, in which a fully functional Hyperloop Demonstrator is being developed. The application of the developed architecture is done to the so-called Main Controller, which is a component of the control system of the Demonstrator.

The present control system consists of centralized and decentralized components and shows similarities to already proven systems from ultra high-speed ground transportation systems. The Main Controller acts as a decentralized control component and serves as an interface between the central operation control system and the microprocessors mounted decentrally on the vehicle. The Main Controller is responsible for controlling its own sensors and actuators, generating reference values for the XMC boards, and distributing control commands and actual data between the participants of the control system. The communication between the participants takes place via EtherCAT.

In this work, a hierarchical structure for the Main Controller is developed, on whose levels state machines are implemented. These are modeled using UML diagrams and tested for functionality using various test scenarios.

Zusammenfassung

In dieser Bachelorarbeit entwickeln wir eine Architektur für eine PLC-Steuerungssoftware. Die Arbeit ist im Rahmen des TUM Hyperloop Projekts anzusiedeln, in dem ein voll funktionsfähiger Hyperloop Demonstrator entwickelt wird. Die Anwendung der entwickelten Architektur erfolgt auf den sogenannten Main Controller, der eine Komponente des Steuerungssystems des Demonstrators ist.

Das vorliegende Steuerungssystem besteht aus zentralen und dezentralen Komponenten und weist Ähnlichkeiten zu bereits bewährten Systemen aus Ultra High Speed Ground Transportation Systemen auf. Der Main Controller fungiert als dezentrale Steuerungskomponenten und dient als Schnittstelle zwischen der zentralen Betriebssteuerung und den dezentral am Gefährt angebrachten Mikroprozessoren. In seiner Position ist der Main Controller dafür zuständig, eigene Sensoren und Aktoren zu steuern, Referenzwerte für die auf den XMC-Boards implementierten Stromregler zu erzeugen und Steuerungsbefehle sowie Istdaten zwischen den Teilnehmern des Steuerungssystems zu verteilen. Die Kommunikation der Teilnehmer untereinander erfolgt via EtherCAT.

In dieser Arbeit wird eine hierarchische Struktur für den Main Controller entwickelt, auf deren Ebenen Zustandsmaschinen implementiert sind. Diese werden mithilfe von UML-Diagrammen modelliert und durch verschiedene Testszenarien auf Funktionsfähigkeit geprüft.

Contents

Chapter 1

Introduction

The neologism "flight shame" describes the feeling of personal shame regarding the use of airplanes. The term comes, just like climate pioneer Greta Thunberg, from Sweden and found its way into dictionaries worldwide since 2017 [28]. In addition to its literal meaning, the term stands for an increased environmental awareness that leads people to rely less on or not at all on airplanes as a means of transportation. At the same time, however, it also indicates the difficulty of this resolution. In most cases, climate protection means sacrificing comfort and speed. In the course of this paradox, it is the task of science to develop new forms of mobility that do not have any disadvantages compared to current standards. In this paper, we will focus on the concept of the Hyperloop, a novel ultra-fast ground transportation system that could offer a way out of this paradox.

1.1 Motivation

According to figures from the European Environment Agency (EEA), commercial air traffic is responsible for 13 % of CO_2 emissions in the transport sector within the EU [11]. If this value is related to the number of passengers and kilometers traveled, aviation is the lonely leader among the most environmentally damaging means of transport [29]. This is mainly due to the high energy consumption of aircraft compared to ground transportation. The high energy density required is derived from fossil fuels, which are considered a catalyst for climate change. While road traffic is still the main source of CO_2 emissions (72 %), efforts to reduce emissions have been underway for some time. Emissions from road traffic are expected to fall by almost 50 % by 2040, while the figure for air traffic could triple. In addition, the trend toward electromobility offers the opportunity to make road transport and its energy producers completely emission-free by generating the required energy from renewable sources such as wind and solar power. In contrast, there is currently no technology to replace fossil fuels or significantly reduce energy consumption in air travel [15].

This poses an immense challenge to policymakers and society. Up to now, measures have been limited mainly to calling on the population to do without climate-damaging means of transport. This mainly concerns domestic flights. For example, the Federal Environment Agency (UBA) in Germany has changed its current travel recommendations to avoid air travel for short trips and domestic travel and instead use more climate-friendly alternatives [29]. Other countries are going even further in this regard. For example, as part of a new climate protection package, France has decided to ban short-haul flights if the destination can also be reached within two hours by train [31].

In the face of the climate crisis, such measures are undoubtedly important and right, but the aspect of doing without also meets with a lack of understanding among large sections

of the population. For example, if you compare the commuter route between Munich and Frankfurt, which is important in Germany, you conclude that traveling by train, which is a more climate-friendly mode of transport, takes three times as long as by plane. To solve this dilemma, new ideas and approaches are needed in the transportation sector.
A new concept for ultra-fast ground transportation system, called Hyperloop, was proposed by Elon Musk in 2013. It aims to achieve speeds close to the speed of sound by drastically reducing air and frictional resistance. It also aims to operate with zero emissions by using climate-friendly energy sources.

The original concept, described by Musk in a 2013 white paper [18], involves moving a vehicle in a partially evacuated tube. It involves the use of a compressor at the front of the vehicle, which directs oncoming high-pressure air underneath the vehicle. That creates a cushion of air on which the vehicle can levitate. This technology promises to almost eliminate air resistance and friction. The advantage over conventional high-speed trains, which can also operate with low friction in tunnels, is that by using the compressor it overcomes the Kantrowitz limit [1].

In the years following the paper's publication, several companies, and research groups have developed approaches for implementing this concept. In 2019, the Technical University of Munich (TUM) also established a student research project, called the TUM Hyperloop Program, to advance research in this area. The goal of the project is to implement a 24-meter test track and a vehicle with a capacity for four passengers. Unlike Musk's concept, the designed vehicle, hereafter referred to as a pod, uses three magnetic systems for propulsion, levitation, and guidance.

Whilst the technology sounds promising, it requires a complex hardware and software structure to operate. In this context, a control algorithm is to be developed that coordinates the different hierarchically sorted control modules. In this work, we focus on the implementation of a software architecture for a component of the decentralized control system, called the Main Controller.

1.2 Objectives

The main goal of this work is the elaboration of a software architecture for the Main Controller. The Main Controller is a programmable logic controller (PLC) and part of the control system of the demonstrator consisting of further PLCs and microcontrollers. A more detailed system description will be given in the following chapter.

The elaboration of the architecture is based on the Unified Modeling Language (UML) according to ISO/IEC 19505 standard [16]. The entire functionality of the Main Controller as well as the communication with other participants of the control system is to be represented by the use of suitable diagrams. In addition, a suitable hierarchical structure is to be defined so that tasks and responsibilities can be determined and assigned. The modeling will be preceded by a comparison with approaches from related industrial fields and possible commonalities will be identified. We will also implement the necessary hardware components, such as sensors and actuators, and check them for functionality so that they meet the requirements of the demonstrator. In addition, software components, such as simulation models from Simulink, must also be included in the modeling.

The implementation of the program code is done according to the IEC 61131-3 standard [14]. Basic paradigms of object-oriented programming (OOP) are to be observed. We want to investigate where the possibilities and limitations of this concept lie in the context of the demonstrator project.

Finally, the implemented system will be tested for completeness and functionality in vari-

ous test scenarios so that it can be used for the operation of the demonstrator.

1.3 Outline

After the topic is introduced in the first chapter and the motivation and objectives of this thesis are presented, the second chapter provides the necessary background knowledge about the demonstrator project to understand the development of the software architecture. In particular, the functionality of the software infrastructure and the subsystems integrated into the Main Controller are discussed.

In the following chapter, existing systems from related industrial fields are presented to elaborate an approach to this project. This includes a comparison with the operating system of the Maglev and the Transrapid, which in many respects can be seen as the basis for the Hyperloop. In addition, this chapter will take a closer look at the AUTOSAR software standard from the automotive industry to transfer possible similarities to Hyperloop. Furthermore, in this chapter, we give a brief overview of the UML and OOP standards on which this thesis is based. In doing so, we introduce the diagram types we use as well as the basic principles and goals of object-oriented software development, which we have made our goal to adhere to in this thesis.

The fourth chapter contains the actual elaboration of the problem, starting with the definition of the different hierarchy levels within the Main Controller. This provides an overview of the overall system behavior of the Main Controller. Based on this hierarchy, the top level is modeled first. This contains the main state machine, as well as important methods for data handling and error handling. In addition, the communication interfaces to other participants of the control system are defined. Subsequently, the subsystems on the second level are elaborated. This includes the modeling of the system behavior by a state diagram as well as the description of the respective task areas. In a further section, reference is made to special subcomponents whose complexity requires separate consideration. These include, above all, the software components that were not developed within the scope of this work. Their temporal behavior will be described more closely with the help of UML sequence diagrams. After the elaboration of the software architecture, the fifth chapter describes the implementation of the code and the subsequent testing. Different test scenarios are presented and applied to the code. In addition, the program code is examined for the usual metrics of code analysis.

The final chapter provides a summary of the lessons learned from this thesis. In doing so, we point out the limitations we encountered during the elaboration and give an outlook on future developments. We also suggest potential modifications that could be used to further extend the elaborated software architecture.

Chapter 2

Demonstrator Overview

This chapter explains the basic knowledge needed to understand this thesis. We give a brief overview of the proposed Hyperloop system and specifically address the areas of the demonstrator that play a role in the implementation of the Main Controller.

2.1 Introduction to the Hyperloop System

First, we provide a brief overview of the proposed Hyperloop system, which is derived from the original Hyperloop Alpha white paper [18]. We then present the three main elements of the demonstrator so that an overview of the overall system can be obtained.

Musk's idea of a Hyperloop is based on air-bearing technology. By using a compressor inside a partially evacuated tunnel, an air cushion is to be created underneath the pod, on which it will float, propelled by an electric linear motor. Due to the reduction in friction and air resistance, Musk believes speeds of up to 1000 kp/h are possible [18]. While this idea sounds appealing, it fails in practical implementation. For example, it turns out that the required airflow rate exceeds the available air in a partially evacuated tube to achieve a levitation height of several millimeters [10]. A low hover height has a detrimental effect on the manufacturing cost of the tube since manufacturing accuracy correlates with cost. However, the demonstrator project is particularly designed to keep the construction cost of the tube as low as possible to achieve good scalability. Thus, Musk's proposed air-bearing technology is not feasible.

The demonstrator project, therefore, proposes the use of an electromagnetic suspension system (EMS). This system has the potential to generate air gaps in the range of several millimeters and is already proven, having been developed to technical maturity in the German Transrapid [26]. From this, the proposed Hyperloop system can essentially be divided into three areas, which are presented below.

2.1.1 Tube and Operation Control Station

The tube with the connected Operation Control Station provides the infrastructure needed for a successful implementation of a Hyperloop demonstrator. The tube has a length of 22.8 m and consists of six prestressed concrete segments. The outer diameter is 4.20 m. The tube is airtight so that a vacuum can be generated inside [25]. The exact mode of operation will not be discussed in detail, as it is not relevant to this work.

All electronics on and inside the tube are executed and monitored from three containers at the end of the tube. These form the Operation Control Station. The electronics container

contains the DC power supply for the levitation, the inverter for the propulsion, and two cabinets for the general electricity. The second container contains the two vacuum pumps and the pneumatic devices and the emergency valves. The third container is the operations control container. This is where human operators will work during the operation and control the demonstrator [22]. The figure fig. 2.1 shows an overview of the demonstrator.

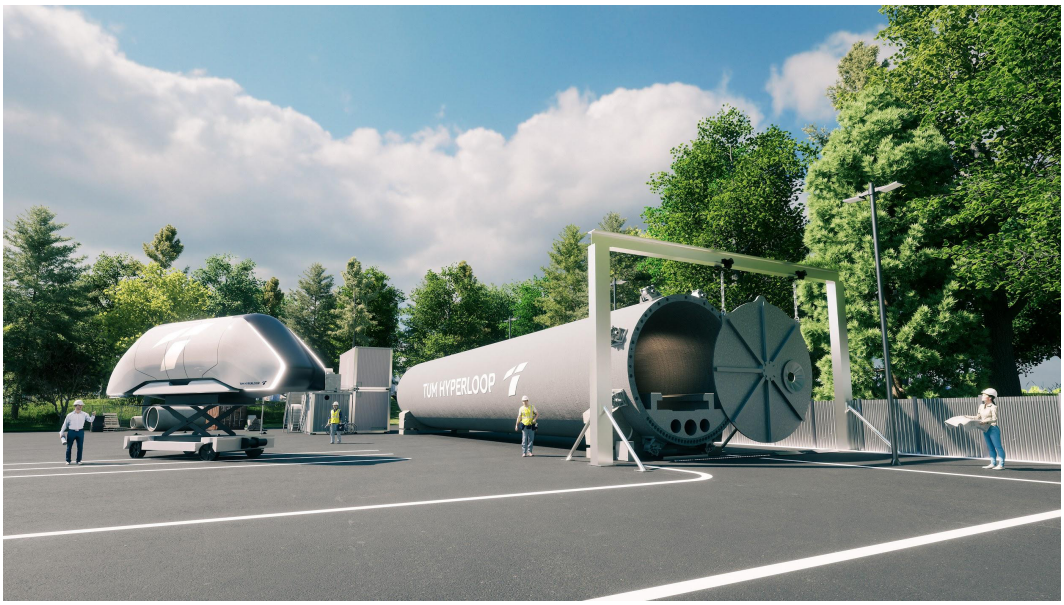

Figure 2.1: Overview of the Tube and the Operation Control Station.

2.1.2 Passenger Module

The Passenger Module (PM) will also be described only briefly, as it is not relevant to the software architecture requirements. As can be seen in Figure fig. 2.2, it contains the pressure vessel with the main door and the emergency door and the desired interior design. The module must provide an engaging and safe environment for passengers. It itself has no propulsion or levitation elements but is only connected to the Service Module via rubber springs [23].

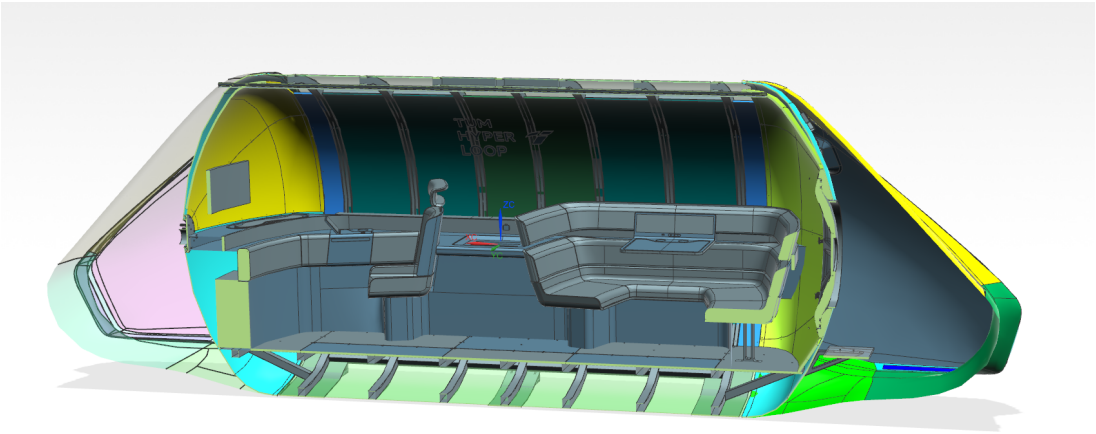

Figure 2.2: View into the Passenger Module

2.1.3 Service Module

The second half of the pod is the Service Module (SM). Unlike the Passenger Module, it is exposed to vacuum conditions, including all its components. It consists of a structural frame and contains almost all technical subsystems, such as levitation, guidance, and propulsion fig. 2.3. Thus, it can also be operated individually as a fully functional platform, which facilitates test operations [24].

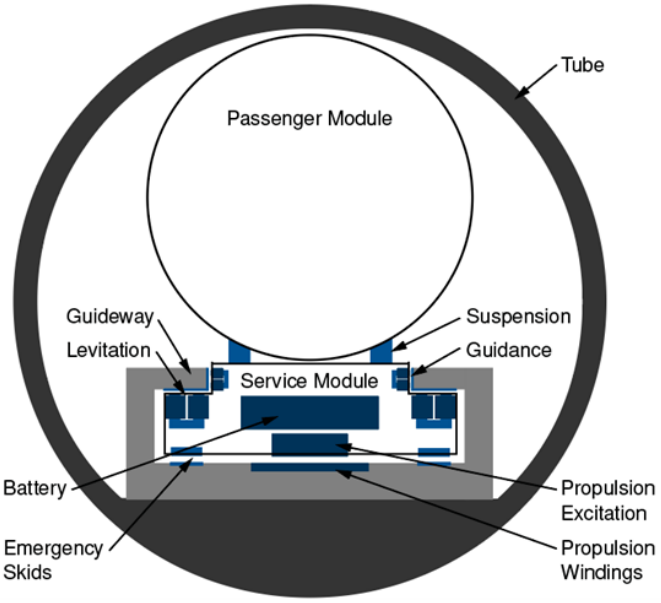

Figure 2.3: Cross section of the demonstrator and technical features of the Service Module.

In addition, the Service Module contains two components that are important for the operation of the Main Controller. On the one hand, there is the so-called IPC Box fig. 2.4, which contains the Main Controller, a 24 V power supply, and an EtherCAT Port Multiplier [21]. The counterpart is the Electronics Box (EB). This contains, among other things, an XMC board, which provides the infrastructure for suspension control. The microcontroller on this low-voltage board handles all sensor and actuator signals and is connected via EtherCAT to a superordinate PLC, the Main controller. The box also contains a high-voltage board for driving the levitation coils. In total, there are four suspension units and thus also four EBs, each with an individual XMC board. The four boards are controlled by the Main Controller [20].

2.2 Introduction to the Control Software

The following section takes a closer look at the components of the control system which, together with the Main Controller, form the software infrastructure of the demonstrator. Understanding how these components work is important because they are in direct contact with the Main Controller and create interdependent relationships. The interfaces between the participants of the control system, therefore, play a dominant role in the elaboration of the software architecture.

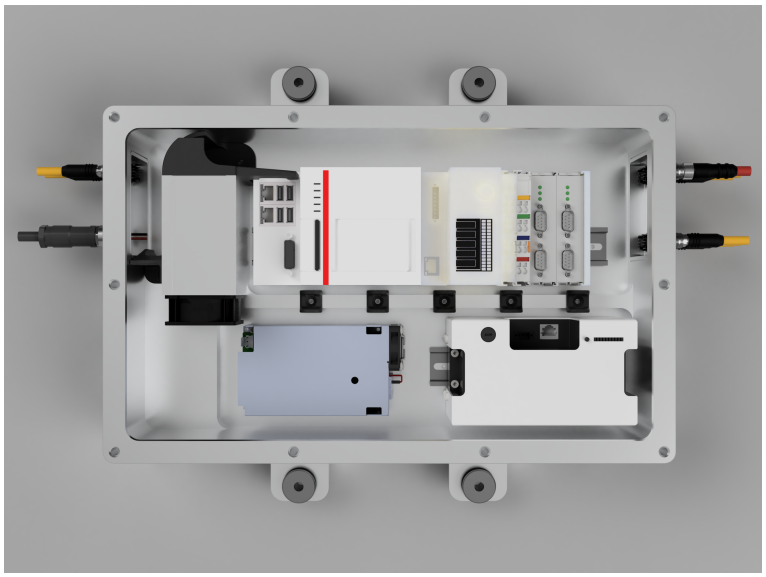

Figure 2.4: View on the IPC Box containing the Main Controller.

2.2.1 Operation Control

The Operation Control System is the entry point to the demonstrator's control system. It receives all instructions given by human operators at the GUI and either implements them itself or forwards them to the Main Controller. In addition, the Operation Controller (OC) collects all data generated during the operation of the demonstrator and sends it bundled back to the GUI so that it is viewable by the human operator. This includes data from its subsystems as well as all operation-relevant data from the Main Controller and the XMC boards.

The Operation Control System in itself consists of six subsystems. The first, the Orchestration Task, is responsible for orchestrating the state machines of the other subsystems and taking action when critical errors occur. This procedure is called the Blue Button Procedure (BBP) and essentially describes a controlled shutdown of the system so that no hardware is harmed. This procedure is initiated when faults occur if it can be ruled out that there is no danger to human life. The second subsystem is called Tube Control. It is responsible for controlling the door system and the lighting system inside the tube and monitors important sensor values for distance, humidity, and pressure. The next important subsystem is called Vacuum Control. It is responsible for the creation and destruction of the vacuum in the tube. To do this, it controls the regulation of the valves, pumps, and cooling units. The vacuum system has two valves that are important for the safety process. One is the pump protection valve, which can be adjusted gradually so that the desired vacuum can be maintained and the pumps get not hurt in the process. The other is the emergency valve, which opens abruptly when an error procedure is initiated so that the tube is filled with air again. The Interior Control subsystem is responsible for monitoring sensor data inside the pod. The other subsystems are Propulsion Control and Levitation & Guidance Control. The latter passes on control commands concerning levitation and guidance to the Main Controller.

The state machine of the Operation Controller consists of the states *INIT*, *OPERATIONAL*, *SHUTDOWN*, *ERROR*, and *BBP*. An individual state machine consisting of these states is implemented for each subsystem. The subsystems do not change the states by themselves, except for ERROR to BBP. The instruction for the transition always comes from the orchestration task. This also checks possible cross-dependencies and transition conditions.

The Operation Controller is an industrial computer (IPC) from Beckhoff and will later be

located in the Operation Control Container. It runs on the TwinCAT 3 computer platform with the eXtended Automation Engineering (XAE) environment [4]. The system is thus real-time capable with minimum cycle times of 50 μs. The communication with GUI and the Main Controller takes place via EtherCAT. The operation controller acts as EtherCAT master to its slave, the Cain Controller.

2.2.2 Main Control

The Main Control System houses the Main Controller and is therefore the main component of this work. The following is a brief description of its functions. A more detailed analysis is provided in the fourth chapter of this thesis.

The Main Control System is hierarchically located in the middle of the control system. It receives the instructions from the GUI that are forwarded by Operation Control, as well as data from the OC that is relevant to the operation. It also receives all data from the four XMC boards on the sensor, actuator, and internal system values. This information is bundled and sorted and passed on to Operation Control. In some cases, the Main Controller also accesses the information itself and uses it to feed its internal state machine. In addition to the XMC data, the Main Controller also sends its own operationally relevant data to Operation Control every cycle. The Main Controller thus forms the interface between the central Operation Control system and the XMC boards distributed locally on the Service Module.

The state machine of the Main Controller will not be discussed in detail at this point. We will elaborate on this in the fourth chapter of this thesis. For this purpose, the following general conditions apply. The Main Controller is responsible for the control and monitoring of levitation and guidance as well as other smaller actuators and sensors. It is not operated directly by the human operator but is called and activated by the OC. State changes should also only occur on command from Operation Control.

The Main Controller is located in the IPC box directly at the Service Module. It is therefore always in the current position of the pod. The controller itself is an industrial PC (IPC) from Beckhoff with TwinCAT 3 as the programming environment. The XAE Runtime enables real-time capable behavior, which is required for the operation of the demonstrator. The communication with other participants of the control system takes place via EtherCAT, which will be discussed in more detail below. On the one hand, it behaves as a slave to the OC, on the other hand at the same time as a master to a connected coupler, to which several terminals for sensors and actuators are attached. The IPC is of the type CX 2040 [7]. The coupler is of the type [5].

2.2.3 XMC Boards

The XMC boards form the third part of the software infrastructure. Each board consists of an XMC4800 ARM-Cortex-M4 microcontroller manufactured by Infineon [13], which is capable of EtherCAT communication. It also has several transceivers and connectors, among others for the connection to the Main Controller and the high-voltage board. Furthermore, a power conversion unit is available, which can convert the 24 V supply voltage to 3 V or 5 V. On the component abstraction layer, the XMC board has various sensors, as can be seen in Figure fig. 2.5.

On the Application Layer, a state machine is implemented fig. 2.6. This is relevant for the implementation of the Main Controller in that state transitions depend on the respective state of the MC. In the *INIT* state, all sensors are initialized and a coil test is performed. As soon as everything is initialized, the system switches to the *STOP* state. On the signal of

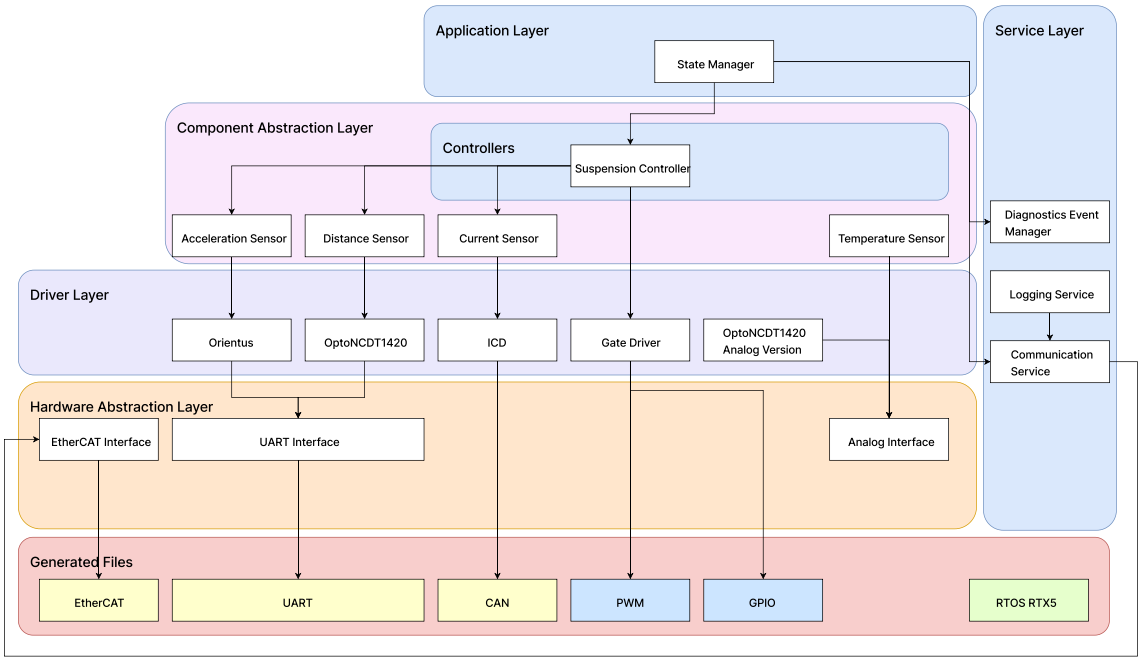

Figure 2.5: XMC Component Overview.

the Main Controller, it is changed into the *OP* state and also back again. In the *OP* state current is applied to the coils so that levitation and guidance are activated. Meanwhile, sensor values are periodically checked for occurrence and validity in the background. If a serious error occurs, the system switches to the *ERROR* state. This is in turn exited by an error acknowledgment signal from the Main Controller in the direction of the *STOP* state. Critical errors and warnings are each defined in a 32-bit array, which is cyclically passed on to the Main Controller together with all data relevant to the operation.

There are a total of four XMC boards, each of which is located in an electronics box. These are in turn distributed decentrally on the Service Module. The four XMC boards behave independently of each other and are not aware of the existence of the others. The EtherCAT Connector allows the connection to the Main Controller. The XMC boards act as slaves, the Main Controller as the master.

2.2.4 EtherCAT Communication

The demonstrator's control system mainly uses EtherCAT to communicate between the individual nodes. EtherCAT is the real-time capable extension of the IEEE 802.3 Ethernet standard [8]. Components that do not support EtherCAT are integrated into the network via interfaces.

An EtherCAT system consists of an EtherCAT master that initiates the communication by sending a specific Ethernet frame. This frame embeds the actual EtherCAT frame in an Ethernet header and a Frame Check Sequence (FCS) to verify the data unit. The EtherCAT frame consists of a header and several datagrams. Each of these datagrams is assigned to a station of the system and contains its header, the data part, and a working counter. The data part contains input and output data. The input data is provided to the subscriber node and the output data can be filled by the node while the network packet is forwarded. This forwarding is done solely by a hardware component of the network subscriber, resulting in low latency and real-time capability.

Within the demonstrator, there are two networks linked by the Main Controller. In the

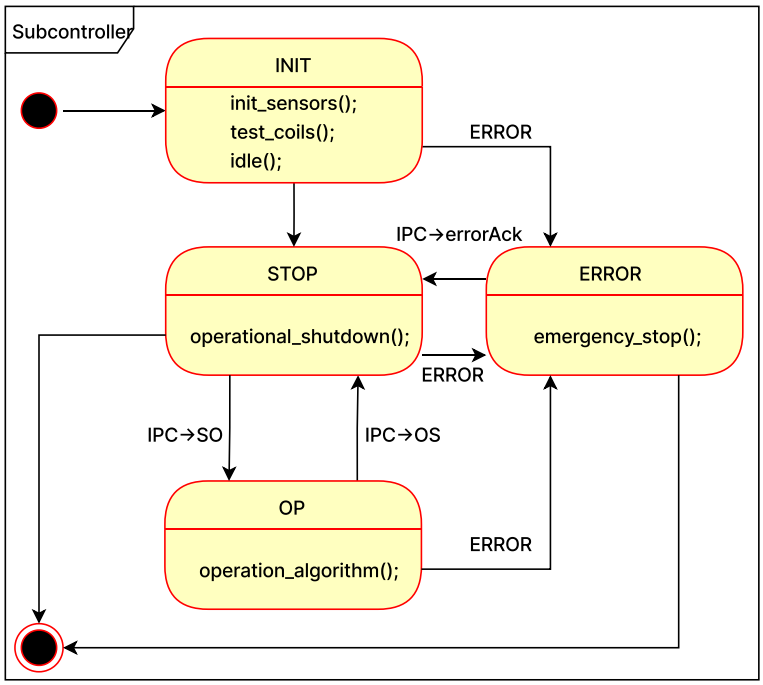

Figure 2.6: Finite State Machine running on XMC board.

communication between OC and MC, the Main Controller behaves as a slave. This means that Operation Control initiates all communication. In the second network structure between MC and the XMC boards as well as the other system components, the Main Controller acts as the master and initiates the sending of the EtherCAT frames. It thus also determines the cycle time. This is important because certain controllers for levitation and guidance depend on low cycle times to be able to guarantee a certain accuracy. In the fourth chapter, we will discuss this problem in more detail. Figure fig. 2.7 shows the EtherCAT communication structure between the nodes of the control system.

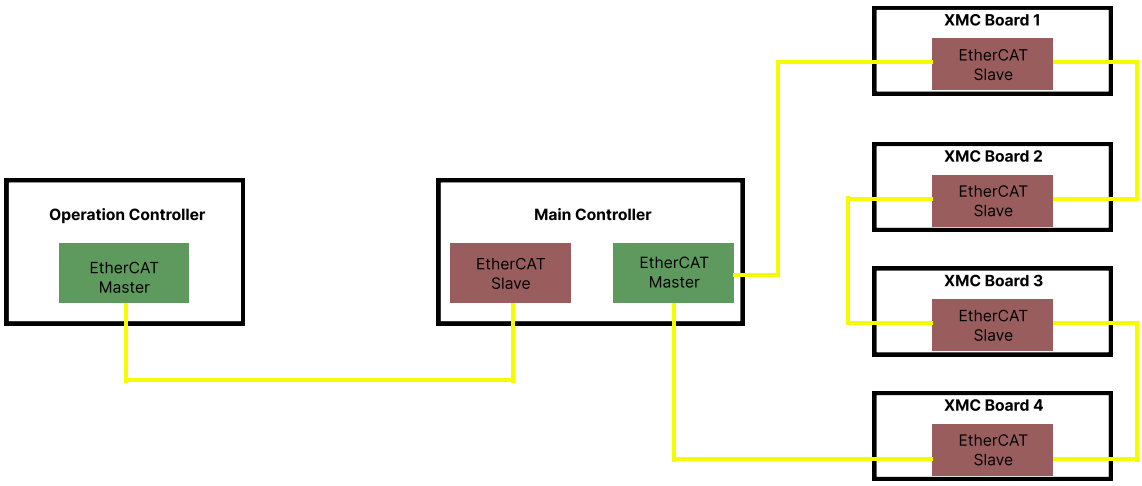

Figure 2.7: EtherCAT communication structure of overall system.

Chapter 3

State of the Art

In this chapter, we present systems from related industrial sectors whose proposed software architectures serve as the basis for the considerations in this thesis. This includes, in particular, the German Transrapid technology, which has generally played a major role in the conceptual design of the demonstrator. We also present the standards and norms for software modeling and implementation that we have used.

3.1 Comparison to Transrapid

The demonstrator is based in many respects on ideas from the German Transrapid technology, developed by Thyssenkrupp between 1969 and 1991, which first found commercial application with the Shanghai Transrapid project in 2002 and is still in operation today [26]. A major advancement in development was the introduction of decentralized distance controllers with the TR-05 model in 1979. The train has several mechanically decoupled suspension modules that individually control the air gap height. This technology is referred to as the Magnetic Wheel [12]. The name comes from the fact that the decentralized approach allows it to respond flexibly to track conditions, much like the wheels on a car. However, the biggest associated advantage is the cost reduction to manufacture the tracks, as they are significantly affected by the manufacturing tolerances. The more precise the tolerances are, the more expensive the track will be. Another advantage is that a fault in a decentralized control system does not necessarily fail the entire system, since the remaining modules that are still functioning can compensate for failed modules. This significantly increases safety.

In the demonstrator project, the conceptual approach was designed to keep track of costs as low as possible to achieve good scalability for long-distance traffic. A decentralized solution for a control system, therefore, shows great potential. However, the decentralized approach also leads to completely new requirements for the software architecture. Above all, the communication structure in the pod must ensure that the sensor values required for control, reach the individual microcontrollers in time. Therefore, in the following, we present the operating system of the Transrapid Model 08, which has also been used in a slightly modified form in Transrapid Shanghai in commercial operation since 2004 [26].

The operation control system (OCS) enables safe and automated operation. All subsystems, such as propulsion, levitation, and guidance, are monitored and controlled by the OCS, resulting in a highly available overall system. The OCS consists of centralized and decentralized components as well as various communication networks. Central components are located directly in the operations center. Decentralized components are located on the vehicle side, in switches or stations. Communication between the individual components takes place via the installed communication networks [27]. To ensure safe operation, all compo-

nents are designed with redundancy. Figure fig. 3.1 shows an overview of the most important components of the OCS.

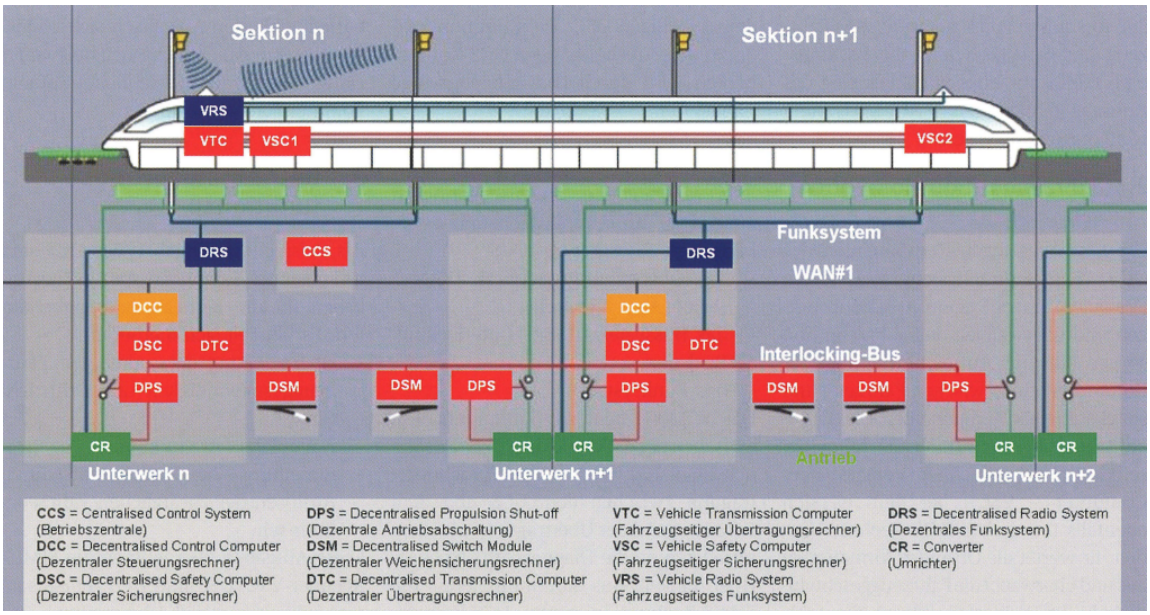

Figure 3.1: The OCS and its components.

The central components of the Transrapid can be compared with the operation controller and the connected graphical user interface (GUI) of the demonstrator. In the Transrapid, the main task of the central components is to control and coordinate the operational processes. This includes, above all, the provision of timetables by an automation computer (AC) and the reaction to operator inputs at the operator terminal (OT). This is comparable to today's GUI. Inputs should be possible at any time, regardless of the state of the automation system. This is also implemented in the demonstrator in that Operation Control cyclically obtains its instructions from the GUI. However, the scope of the OC is significantly slimmed down compared to the central components of the OCS since there is no need for timetable generation (AC) or radio communication with the pod (CRCU) on the short test track of the demonstrator. In the Transrapid, the latter enables voice transmission from the central control system (CCS) to the vehicle. Another component of the CCS is several diagnostic workstations (Diagnostic Terminal System - DTS), where diagnostic messages for maintenance purposes are displayed from the entire system. This system is also used in a modified form in the demonstrator's central control system. Operation Control collects all warnings and error messages occurring during operation and cyclically passes them on to the GUI for output.

In the context of this work, a look at the Transrapid's decentralized operating system (DCS) is essential, as it is comparable to the Main Controller. The Transrapid system divides its track layout into different decentralized areas, each of which is assigned a DCS. All DCS units are connected to the CCS and each other by data networks. In each DCS area, several vehicles could theoretically be managed, secured, and controlled simultaneously. De facto, however, this is not achieved in practice because the drive technology is integrated into the track. Thus, only one vehicle can be moved in a drive area at a time. Each DCS thus takes over a different part of the vehicle, as can be seen in Figure fig. 3.1.

During operation, manual or automatic instructions from the CCS arrive at the decentralized control computer (DCC). This controls the operation in a DCS area. It is the equivalent of a modern IPC such as the Main Controller in the demonstrator project. The DCC determines reference values for travel specifications from the state information available to it and

transmits these to the components required for operation. It also communicates operational state information to the CCS [27]. A first comparison with the conceptual design of the Main Controller shows great similarities. For example, the Main Controller also serves as a link between the central control system and decentralized components responsible for controlling propulsion, guidance, and levitation. However, due to the short track length of the demonstrator, the control system is limited to only one decentralized system that controls the entire track length.

In the Transrapid, the DCC also has other tasks. For example, reference values and status information for non-safety-relevant functions are passed on to the respective executing components directly after processing by the DCC. Safety-relevant information, on the other hand, is first passed on to the decentralized backup computer (DSC) for checking. The DSC is responsible for securing, controlling, and monitoring the operational processes in a DCS area. This includes, above all, the safe location of the vehicle, the safeguarding of the route, and the setting of minimum and maximum speeds. In the demonstrator project, these tasks are largely omitted because the short test track does not require them. Transferred to the demonstrator project, the DSC can therefore be assigned to the range of tasks of Operation Control.

Another component of the DCS in the Transrapid is the decentralized transmission computer (DTS). This secures the decentralized radio system for data exchange between the vehicle and the track. The vehicle also contains another decentralized control system (OCS) to safeguard the components on the vehicle side. Here, too, there is a backup computer (VSC) that manages location information and, in the event of a fault, brings the vehicle safely to the next stopping place. There is no equivalent to this system in the demonstrator. This is because the decentralized control system in the demonstrator is designed directly on the vehicle. The complete decentralized control system is located in the Service Module of the pod, while the Passenger Module no longer requires its own control system. In addition, this approach promises to keep track of costs low, since all software components are housed in the pod. Nevertheless, the comparison with the Transrapid is worthwhile, since a certain basic structure for the Main Controller can be adopted without a doubt. This includes, above all, the function as a link and the internal differentiation between critical and non-critical functions to be able to guarantee safety procedures.

3.2 Comparison to Maglev

The Maglev is also a transport system based on magnetic levitation technology. It has been in test operation in Japan, among other countries, since 1997 and is to be expanded to commercial operation in the coming years [26]. While we have compared the general structure of the control system for the Transrapid, the focus for Maglev is on the Prognostic and Health Management (PHM) architecture. Maglev proposes a decentralized distributed software architecture for monitoring the system during operation and detecting errors. With the help of this architecture, maintenance work should become more efficient, less expensive, and less extensive [19]. It is also intended to reduce the probability of catastrophic accidents.

The Maglev train is divided into 14 different subsystems fig. 3.2, each of which has its own equipment with different parameters to be monitored. Therefore, Maglev train data is characterized by massive, layered, and distributed data packages. Using a centralized monitoring architecture would require a very large network bandwidth to transmit and analyze all data in real-time. For this reason, a hierarchical distributed solution is used [19]. The hierarchy is divided into vehicle, system, and subsystem. The first category accumulates the data and information to be analyzed. Subsystems, such as the suspension control system,

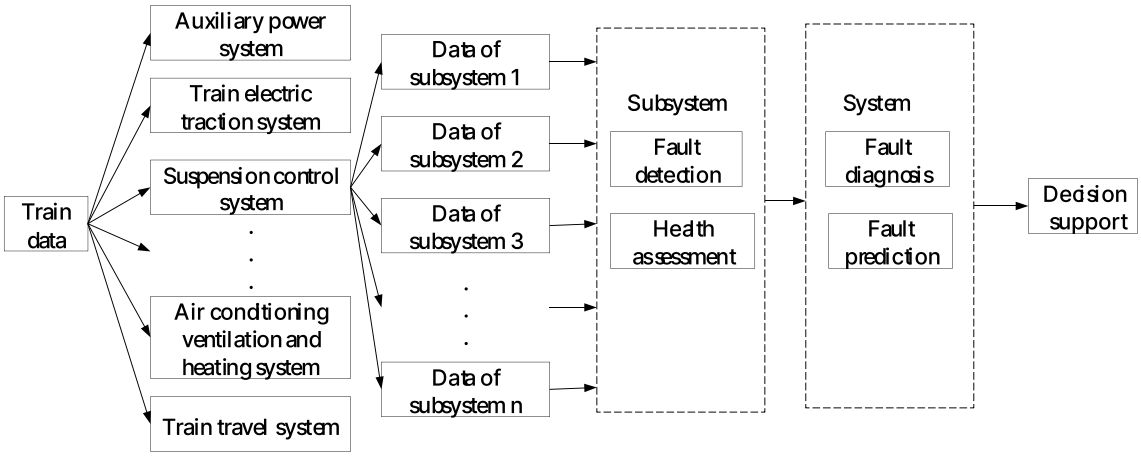

Figure 3.2: The PHM architecture for Maglev based on the distributed hierarchical structure.

collect the data available to them from sensors, actuators, and other components. At subsystem level, faults are then detected and the system's condition is assessed. This in turn is constantly monitored by the responsible parent system. In the event of a system failure due to incorrect behavior, the parent system first looks at the condition of its subsystems to determine where the cause of the error is to be found. The parent system can then request further data from the affected subsystem. Finally, all the information collected regarding the error that has occurred is bundled and sent to the so-called decision support instance. This provides the system with an optimized maintenance description for the maintenance personnel. This instance can be controlled by an automated computer as well as by the intervention of a human operator.

Figure fig. 3.3 shows again an overview of the sequence of actions of the described PHM architecture. It makes sense to implement an error handling system based on this architecture for the demonstrator as well since the decentralized approach can optimize maintenance work and maintenance costs.

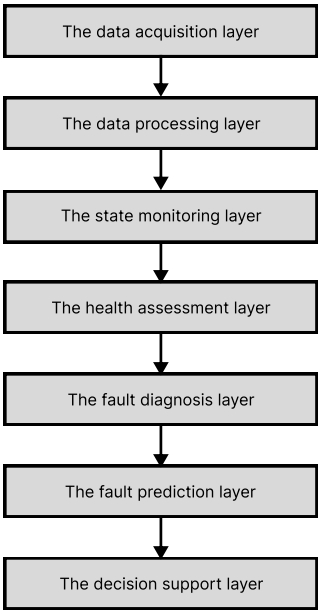

Figure 3.3: The general framework of the proposed architecture.

3.3 Standards and Norms for Modeling

In this section, we provide an overview of the standards and norms we use for modeling and implementing the software architecture. These include, above all, the diagram types of the Unified Modeling Language (UML) according to ISO/IEC 19505 [16] and the paradigms of object-oriented programming for programmable logic controllers. These are defined in an extension of the IEC61131-3 standard [14] from 2010. In addition, AUTOSAR, a standard from the automotive industry, is presented whose proposals for the design of state machines in means of transport can also be applied to the demonstrator.

3.3.1 AUTOSAR standard

Founded in 2003, the international development partnership of automotive manufacturers, suppliers, and other companies from the electronics, semiconductor, and software industries set itself the goal of developing and establishing an open and standardized software architecture for electronic control units (ECUs). In this context, the Automotive Open System Architecture (AUTOSAR) was developed. This serves as a standard for the development of in-vehicle control software for many well-known car manufacturers. Although technologies from the automotive industry do not play an overly important role in the development of the demonstrator, we can certainly draw on generally applicable ideas from this related industrial field when developing our software architecture.

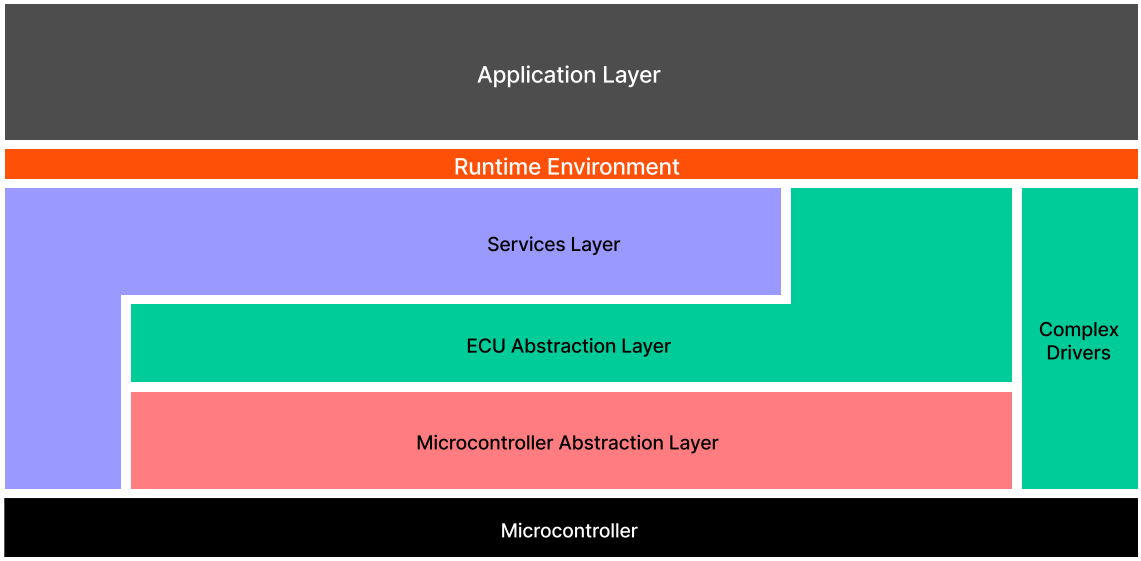

Figure 3.4: Overview of AUTOSAR software layers.

AUTOSAR software is structured in different layers fig. 3.4. In the context of this work, the structure of the basic software between the microcontroller and the application layer is particularly important, since this is where the control programming takes place. At the lowest level is the Microcontroller Abstraction Layer. It contains internal drivers, which have direct access to the software modules of the microcontrollers. It is intended to make higher layers independent of these modules. The ECU Abstraction layer subsequently provides an interface for the drivers of the Microcontroller Abstraction Layer. It also contains drivers for external devices. The main purpose of this layer is to provide a unified Application Programming Interface (API) to peripheral devices regardless of their location and connection to the microcontrollers. In summary, the ECU Abstraction Layer covers access to I/O signals. Thus, higher

software layers are independent of the ECU hardware layout. The Complex Driver Layer has a similar function. It is intended to provide devices with special timing behavior or devices that are not specified in the AUTOSAR standard with suitable drivers. The highest layer of the Basic Software is the Service Layer. Consequently, it also has the highest relevance for the application software. Within this layer, several modules are implemented that execute basic services for applications and the runtime environment. These include vehicle network communication, memory services, diagnostic services, and ECU state management [2]. Above the Basic Software layer is the AUTOSAR Runtime Environment (RTE). All AUTOSAR software components communicate with other components or services via RTE. The task of the RTE is thus to make those software components independent of their assignment to a specific ECU. Above the RTE, the software architecture then changes from layered to component-style.

For this work, the service layer is particularly interesting, since it contains the ECU State management. This is done based on a state machine, which can be seen as the basis for the top-level architecture proposed in this work.

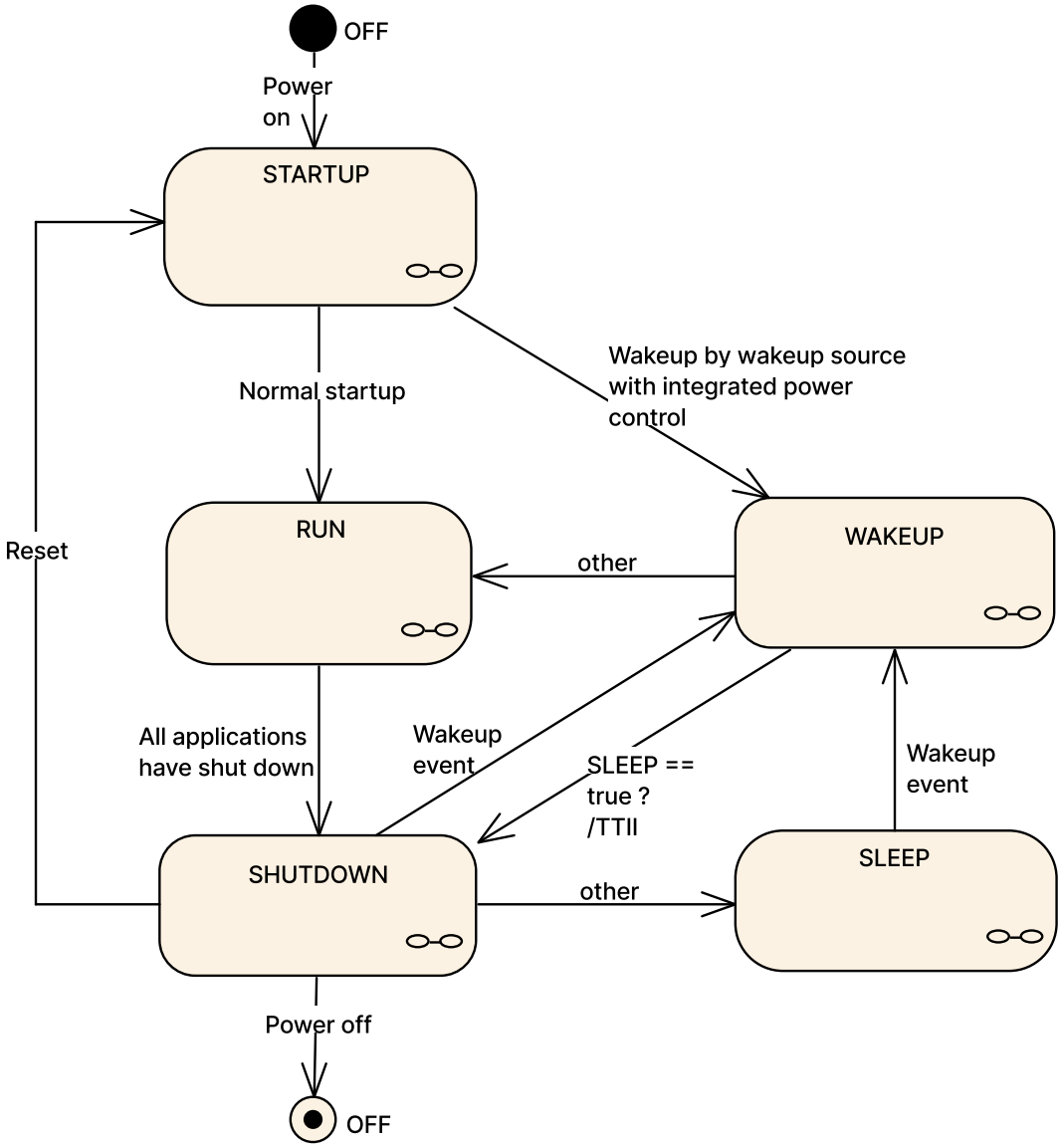

Figure 3.5: AUTOSAR top level diagram with ECU Main States.

The procedure for modeling this main state machine is described in the AUTOSAR specification of the ECU State Manager with a fixed state machine [3]. This proposes to divide

the system hierarchically into the top level, high level, and sub-state. At Top Level, the main state machine runs with the ECU Main States. This state machine is modeled using a UML state diagram and includes the entire system behavior. In it, the life cycle of the system from power-on to power-off is to be represented.

Figure fig. 3.5 shows the top-level state diagram. It contains the following so-called main states: STARTUP, RUN, SHUTDOWN, SLEEP, and WAKEUP. Additionally, there is the pseudo-state OFF.

The purpose of the STARTUP state is the initialization of the basic software modules. The STARTUP state is divided into two parts, the first one is the part before the start of the operating system, and the second one is the part after the start of the operating system and thus when the operating system is running.

The RUN state occurs when all modules from the Basic Software layer are initialized. The state is maintained as long as the application requests RUN. Otherwise, a shutdown is initiated and the system switches to sleep mode. The RUN state is again also divided into two substates. One is the regular RUN state, the other is POST_RUN. This is requested when cleanup or saving of data is required before switching to sleep mode.

The SHUTDOWN state controls the controlled shutdown of the system. This eventually results in a transition to the requested shutdown target. This can be the change to SLEEP or STARTUP by a reset or the complete shutdown to OFF.

The SLEEP state and the WAKEUP state are not relevant for the considerations in this work, since the demonstrator does not have these functions. Nevertheless, for the sake of completeness, they are briefly described. The SLEEP State is a state for saving energy. Within this state, no code is executed, but the system is still supplied with energy. This allows the ECU to be woken up again, which ideally saves time compared to restarting the ECU. When waking up, the ECU runs through the WAKEUP state. In it, a protocol is run through that supports validation of all wakeup events. During wakeup, a distinction is also made between intentional and unintentional.

The OFF State describes the state of the ECU without a power supply. The only requirement for this pseudo-state is that the ECU can be started.

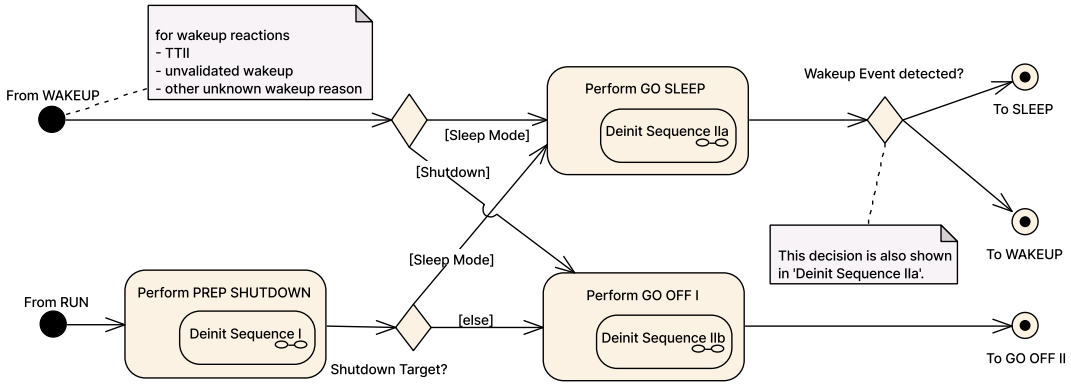

Figure 3.6: Breakdown of a ECU Main State by a sub-state activity diagram.

The top-level diagram in figure fig. 3.5 is limited to a rough overview of the ECU main states and how they are interconnected. No more detailed system behavior is described at this level. Potential substates are also not shown. The ECU Main States are modeled as individual state machines that have internal behavior. This behavior is described at the next level of abstraction by the so-called sub-state diagrams. These are usually UML activity diagrams. Figure fig. 3.6 shows an example of the breakdown of the SHUTDOWN state structure.

Below the top level is the high level. Here five UML sequence diagrams give an overview of the main activities in the respective states and explain how state transitions occur. High-level

diagrams always start with a reference to the preceding sequence and end with a reference to the following sequence. Figure fig. 3.7 shows the sequence diagram of the RUN state as an example. A more detailed description of the purpose and requirements of the different UML diagram types is given in the next section.

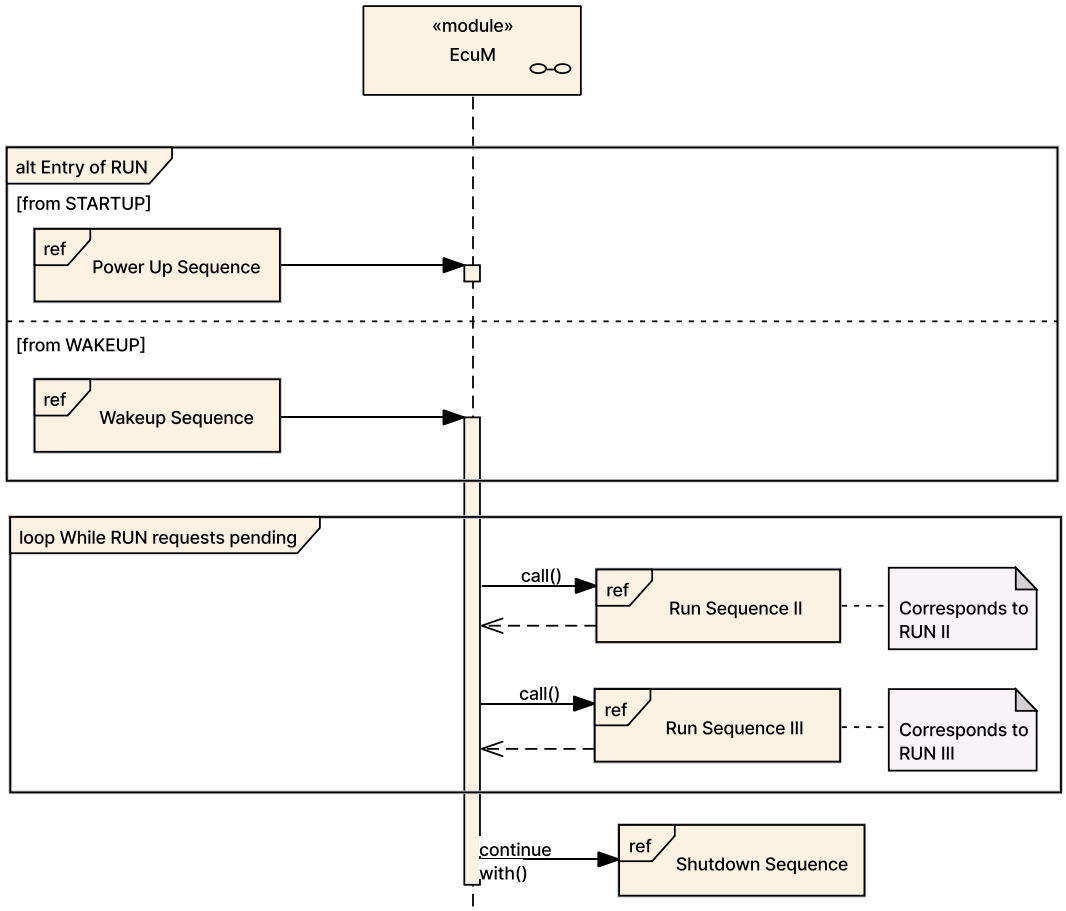

Figure 3.7: High level diagram showing RUN state sequence.

The proposed division of the system into top-level, high-level, and sub-states and the accompanying use of different UML diagram types forms the basis for the elaboration of the Main Controller software architecture in the following.

3.3.2 UML

The Unified Modeling Language (UML) is used for modeling, documentation, specification, and visualization of complex systems. It provides the notation elements of model analysis, design, and architecture and supports object-oriented procedures in particular. Since 1989, it has been standardized in a vendor-neutral manner by the Object Management Group (OMG) [16]. This work is based on version 2.0 of the UML, which was adopted in 2005.

The UML has 14 diagram types, which are divided into structure diagrams and behavior diagrams. Structure diagrams describe the statics of the system, while behavior diagrams represent the dynamics. Behavioral diagrams also have interaction diagrams as a subgroup, which can be notated graphically or in tabular form [9]. In the following, we briefly introduce the diagrams used in this work and explain their respective purpose.

Class Diagram

The class diagram is created early in the project phase. It provides a static view of the system in terms of components, attributes, operations, and relationships. The system is broken down into clear parts, which in turn are connected by clear hierarchical relationships. This facilitates the addition of further functionality. The main components of class diagrams are classes. Class is meant a collection of exemplars that have common properties, constraints, and semantics. Each class is thus an abstracted collective term for a set of like things. The properties of these copies are called attributes and describe characteristics such as frequency and data type. The behavior of classes is described by operations. Apart from the pure existence of the classes the class diagram shows also the exact connections between them. These can be hierarchical, in that classes inherit their characteristics. This creates a parent class and a child class, which has access to all public and protected attributes and operations of the parent class. The parent class can also declare its attributes and operations as private. Only the class itself then has access to this information. Publicly declared attributes and operations, on the other hand, are available to all other system participants, including child classes.

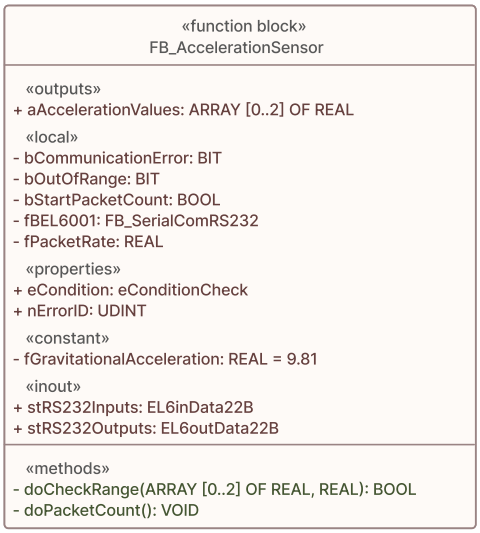

Figure 3.8: Structure of a class.

In the context of this work, a class diagram is used, to provide an overview of all components used and their characteristics. Since programming is done in the IEC 61131-3 [14] language Structured Text (ST), classes do not necessarily mirror real bodies, but function blocks (FBs) and programs (PRGs). These are the basic modules for PLC programming and form a program organization unit (POU). Attributes in this work are consequently all variables declared in the declaration part of an FB or PRG. Local variables are represented as private, and input and output variables as public. In addition, they are marked with their respective stereotype. The properties of an FB or PRG are also displayed in the attribute area. Properties are an object of object-oriented programming [14] and are explained in more detail in the following subsection. Constants are also declared in the attribute area under the corresponding stereotype. Constants are already assigned values as the only attributes. Otherwise only the assignment of a data type takes place. In the operation area of a class, all methods of a FB or a PRG are collected in the context of this work. Again, local or public access is represented by the corresponding character. Methods contain a data type and, if necessary, one or more transfer parameters. This corresponds to the input variables of the method, which are only declared locally during the execution of the method (not resistant).

Figure fig. 3.8 shows an example of a class from the Main Controller class diagram. In this case, it is a function block.

Activity Diagram

The activity diagram is used to model the system requirements. All required tasks and processes are to be represented. The focus is on the abstract overall behavior of the system, not the detailed processes within the individual states [9]. The states of an activity diagram are so-called actions in which operations of a class can be executed. Several related actions can be combined into one activity [9]. In the context of this work, activities form the main states of the top-level diagram. Actions are to be compared with methods of the respective FBs or PRGs. Actions are connected by control flows. These can be assigned a guard. A transition over this edge is then only possible if the condition is fulfilled. In the context of this work, so-called swimlanes are also used. They divide the individual activities according to the origin and thus enable the parallel processing of several systems to be modeled. In the context of this work, each high-level system is represented within a swimlane.

State Diagram

State diagrams in UML describe processes from a detailed complex point of view. They are used within a project phase during component development. Similar to the activity diagram, the system is abstracted into states and transitions that are passed through during operation [9]. For the clear demarcation of the activity diagram states are designated in the context of this work with an adjective. This also clarifies that, unlike in the activities of an activity diagram, only one specific action is executed. For each additional action, a separate state is created. Within these states the operations of classes are executed. The state diagram divides operations into the stereotypes entry, do, and exit. Entry-actions are executed once when entering the state. Exit-actions are similarly executed once when leaving the state. The do-action is active until the state is exited. Exiting the state is initiated by transitions, guards, or after the do action is completed. If the signal of a transition becomes positive, the execution of the do action is aborted and the state is left immediately with the execution of the exit action. If the state transition is occupied by a guard, it can be changed only if the guard occurs. During transitions also transition actions can be executed.

In the context of this work in states subsystems are called and their inputs are set. The setting of inputs always corresponds to an entry or exit action, since this must happen only once. In a do-action subsystems in the form of FBs or functions (FUN) are called and executed cyclically as long as the state is active. In contrast to the activity diagram, transitions are also already precisely defined with correct variable names in the state diagram. This also applies to the actions within the states.

The condition diagrams are thus already very close to the actual program code. In the context of this work, the state diagrams are equivalent to the AUTOSAR state breakdown diagrams [3]. Each high-level system has its state diagram, in which the activity described in the top-level activity diagram is described in more detail.

Sequence Diagram

Sequence diagrams are also used to model system requirements. Unlike activity diagrams, however, they are used rather late in the project phase. They can be used to represent communication processes in detail and to design test processes. In contrast to the activity diagram, only a small section of the overall behavior is shown. Modeling the exchange of information between communication partners is important for presenting fixed sequences, temporal and

logical flow conditions, and loops. Sequence diagrams consist of communication partners represented by lifelines. They receive and send messages. Messages can be modeled in such a way that a response is expected. Otherwise, only operations can be called or signals can be sent [9].

In the context of this work, we use sequence diagrams to model very complex operations in which data exchanges occur with other participants in the control system. They can be compared to the idea of the high-level sequence diagrams of the AUTOSAR [3] standard, except that we use them more on the subsystem level.

3.3.3 Object Orientated Programming

In this work, we program in the programming language Structured Text (ST). This is a language of IEC 61131-3 [14]. This standard describes the structure of the program in program organization units (POUs), which consist of programs, function blocks, and functions.

In addition, since its third edition, IEC 61131-3 also contains an extension with elements of object-oriented programming (OOP). Since these also play a role in the elaboration of this thesis, we briefly introduce the tools we use and go into the most important principles of OOP.

We promise ourselves from the use of object-oriented building blocks improved clarity, modulability, and reusability. OOP has different principles which bring these positive effects. In the context of this work, the design pattern SOLID [17] is mainly applied. SOLID is an acronym for five design principles that support the developer in making software more understandable, flexible, and maintainable. Since the demonstrator project is designed for testing, the occurrence and triggering of bugs will be unavoidable. In this sense, understandable code makes maintenance much easier. This also applies in case the demonstrator is further developed in the future. Thus, due to its flexibility, parts of the code can easily be adopted or replaced.

In the following, the SOLID principle will be briefly described. It should be noted that the rules are by no means considered mandatory, but more as a common thread for our software architecture.

The first principle is called the Single Responsibility Principle (SRP). It states that a functional block should have only one responsibility. If the functionality of a program is changed, this should only affect a few function blocks. By many small function blocks, the code becomes, at first sight, more unclearly, is however simpler to organize. Function blocks should thus have a specific task and not claim to be able to do everything.

The second principle is called Open Closed Principle (OCP). It requires function blocks to be open for extensions but closed for changes. This is because the implementation of changes should only be achieved by adding code, not by changing existing code. An example of this is inheritance. An extension with new functionality is achieved by implementing a new function block that inherits from an existing FB. New functions can thus be added without having to change the existing FB.

In third place is the Liskov Substitution Principle (LSP). It requires that derived function blocks must always be substitutable for their base FBs. This means that derived FBs must behave in the same way as their base FB. Thus a derived FB may extend its base FB, but not restrict it.

The fourth principle is called the Interface Segregation Principle (ISP). It states that many customer-specific interfaces are better than one universal interface. Accordingly, an interface should only contain functions that are closely related to each other. Because by extensive interfaces couplings between otherwise independent program parts develop. Thus, the ISP is the equivalent of the SRP for interfaces only.

The fifth and last principle is the Dependency Inversion Principle (DIP). It is intended to prevent function blocks from being too linearly dependent on each other in one direction. For example, in logging, one FB calls the methods of another FB to write data to a database. There is a fixed dependency between the two FBs. The DIP proposes to resolve this dependency by defining a common interface. This is then implemented by one of the two FBs.

Apart from these principles, the building blocks of object-oriented programming described below are used in this work [30].

In the first place are the methods. These are not independent POUs but are assigned to PRGs or FBs. They contain a sequence of instructions and can return a return value similar to functions. All data within a method is only temporarily valid during execution. This means that all internally declared variables are reinitialized with each call. The object-oriented approach uses methods to separate the code for different tasks into methods within an FB. This allows a clearer structure and readability since one can represent its function by the name of the method.

Another element of object-oriented inheritance is the property. They facilitate parameter passing and make it possible to pass general parameters and states independently of the call of the function block in a firmly defined way. The access takes place by a method pair, which contains in each case a write and a read function. Within these methods still, local variables can be declared. Thus parameters can be subjected already with initialization for example to a range check or a unit conversion. In addition, parameters can be passed via several inheritance hierarchies without having to be cached locally in the individual FBs.

This already leads to the next element of object-oriented programming, inheritance. Here, a new function block is derived from an existing FB. The new FB inherits all the properties and methods of the base FB if its access specifier permits this. In this way, system components can obtain general properties from a common base class and implement individual details themselves.

Interfaces have a similar function to inheritance. Interfaces are a definition of methods and properties. Classes that implement the same interface look identical from the outside and can be treated equally. The exact implementation of the prescribed methods and properties happens individually in each class. In our work interfaces form an important role, to summarize properties of hierarchy levels and to provide them to their assigned systems.

Similar to interfaces is the principle of abstract function blocks. These special FBs contain abstract methods and properties without an implementation part. Function blocks that inherit from them must implement the corresponding methods and properties or are abstract themselves. An abstract FB does not have to be instantiated and can therefore be used in a similar way to an interface. In this work, we want to achieve improved code quality primarily through a combination of abstract FBs and interfaces.

Chapter 4

Software Architecture

This chapter forms the core of this thesis and presents the proposed software architecture for the Main Controller. The chapter is divided hierarchically into four subsections. First, the hierarchical structure of the controller is explained and an overview of the overall system is given using a class diagram. The following sections then go into detail about the modeling of the individual levels. Each level is characterized by modeling through a different UML diagram type so that the respective functionality is revealed in all its complexity.

The focus of this chapter is on the modeling of a functional software architecture for the Main Controller. The associated implementation is only described in rudimentary form in this chapter. Important code excerpts are therefore provided in the appendix.

4.1 Hierarchical Structure and Components

The architecture of the Main Controller is hierarchically structured from three levels. This results in a pyramidal form that is based on the AUTOSAR standard described above [3]. The top level is referred to as the Top Level, followed by the High Level and the Sub Level. These terms are common and proven in the Automotive Software Architecture and find therefore also in this work application. The hierarchy is crucial for several things. On the one hand, responsibilities are derived from it, on the other hand, it provides for a clear basic structure. Each module is explicitly assigned to a level and thus receives its access rights and basic functionality. Different diagram types are used to model each level in the best possible way. The top level is modeled by an activity diagram to give a general overview of the logical processes and the parallelism of events. The High-Level systems are again modeled by an individual state diagram. This is more specific to the internal behavior of the system and the notation is already based on the later implementation in Structured Text. On the sub-level, the two most important components of the Main Controller for levitation and guidance are modeled by a sequence diagram. This is mainly to show the complex communication between the hierarchy levels. First, however, the overall system is considered. This is described by a UML class diagram, see fig. 4.1. A more detailed description is given in the following sections.

4.1.1 Top Level

The Top Level consists of the two programs *PRG_MainController* and *PRG_ErrorHandling*. These form the two most important program modules and are elementary for the operation of the Main Controller. The Top Level is characterized by the fact that its components are

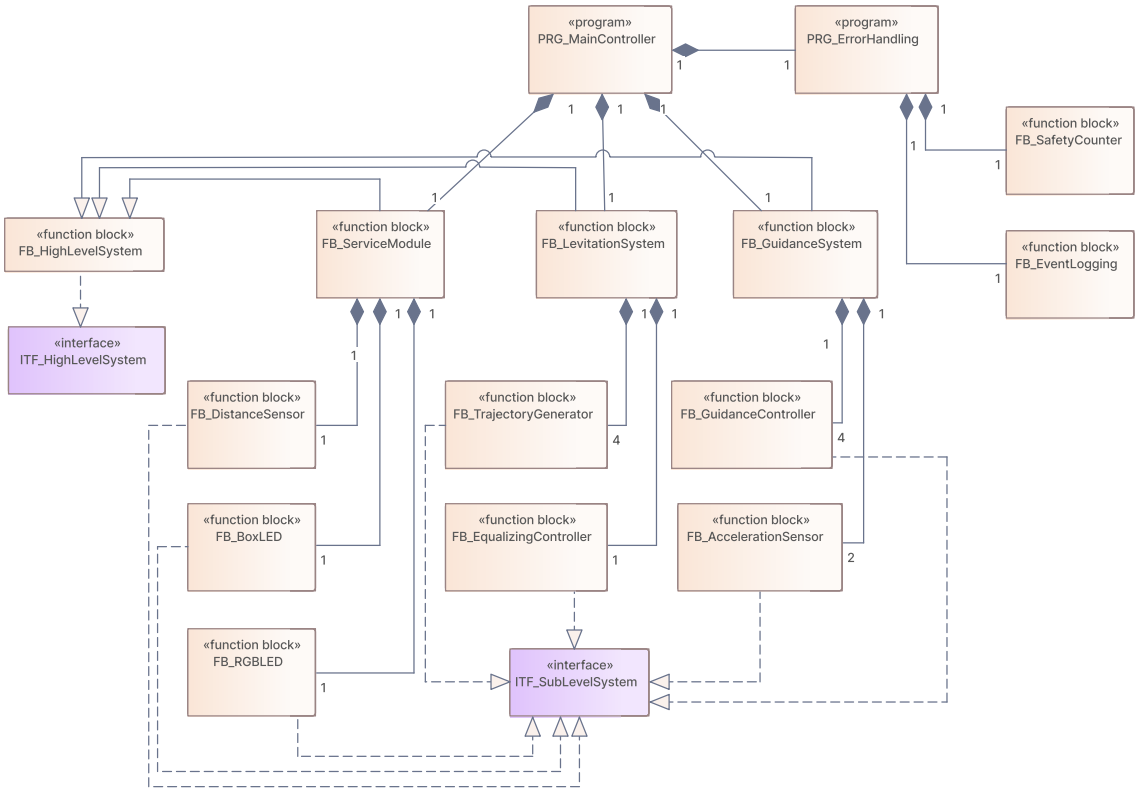

Figure 4.1: Class diagram of overall system.

implemented as a program (PRG) according to IEC 61131-3 standard [14]. Programs are the entry point to a PLC application and the highest hierarchy level. They have only local and persistent variables to which all other system blocks have read access.

The two programs are assigned to a PLC task, which is run cyclically with a fixed cycle time. The cycle time of both tasks is identical in this case and is 1 ms. This time is derived from the requirements of the implemented controllers, which need the smallest possible cycle time for clean calculations. In addition to the cycle time, a task also has a certain priority. This plays a role in the order in which the PLC processes the tasks. In the case of the present application, Error Handling has the higher priority, so that safety-relevant functions are always processed with priority. In principle, it would also be possible to assign both programs to the same task. However, this would result in the loss of safety-promoting prioritization. In addition, splitting the two main components into two different tasks makes debugging and troubleshooting easier, since the two tasks can be tested separately. This is only possible in test mode. To prevent misuse during operation, certain safety precautions are implemented, such as cross-checking.

The Top Level is characterized by the fact that all decisions relevant to operations and security are made here. In addition, the top level is responsible for communication with the other participants in the control system. Seen from the outside, the top level thus represents the entire Main Controller. High Level systems can be controlled and monitored from the Top Level. However, it is not possible to access the subsystems and hardware components below it. The top level is therefore the central data management and control level of the Main Controller.

MainController

The program block *PRG_MainController* is the head of the system architecture. All High Level systems stand in existence-dependent relationship to it. The composition is expressed by the fact that all High Level function blocks are declared and instantiated in it. The module thus has write access to all input variables of the High Level systems and read access to the corresponding output variables.

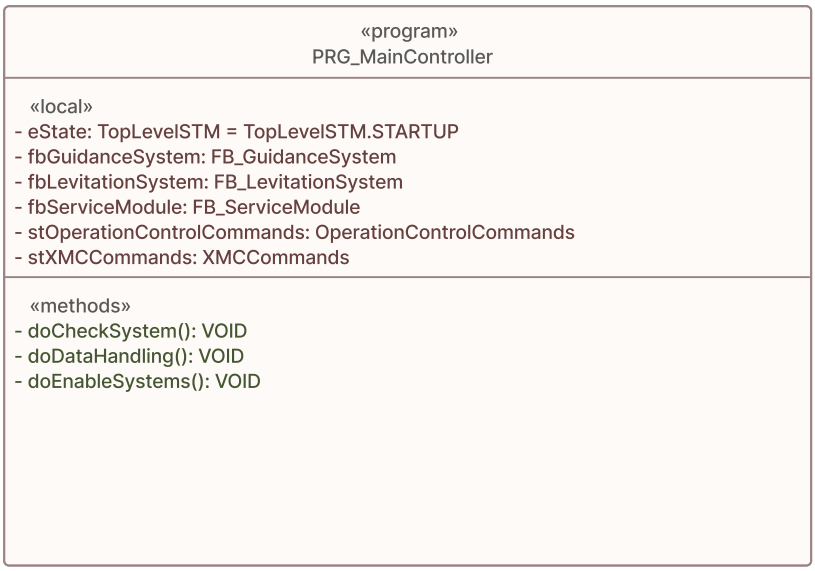

Figure 4.2: Outline of *PRG_MainController*.

The central control organ of the Main Controller is the Top Level state machine, which is implemented in *PRG_MainController*. From it, all High Level systems are addressed. It also reflects the externally visible behavior of the Main Controller. All internal processes on lower levels are not visible or controllable by other participants of the control system. The Top Level state machine regulates the entire life cycle of the Main Controller and is thus the central element of the software architecture. Its logical structure will be discussed in more detail in a subsequent section. In addition to the State Machine, the MainController (MC) also contains the central data handling. Control commands and process data of the communication partners are processed in the method *doDataHandling*. Data that is relevant to the operation of the Main Controller is written to the two structs *stOperationControlCommands* and *stXMCCommands*. They mainly serve as triggers for transitions of the Top Level state machine. Data that is specifically relevant for High Level systems is written directly to the respective inputs of the systems.

The methods *doCheckSystem* and *doEnableSystems* are important for the startup process of the Main Controller. In them, the condition of the hardware components is queried and the appropriate High Level systems are activated at. The Main Controller is designed in such a way that each High Level System can be activated or deactivated individually. This facilitates testing and increases the modularity of the application.

ErrorHandling

All safety functions of the Main Controller are implemented in the *PRG_ErrorHandling* program block. It does not contain any operation-relevant functionality, but collects all safety-relevant information continuously in the background and intervenes in the process flow in the event of an error.

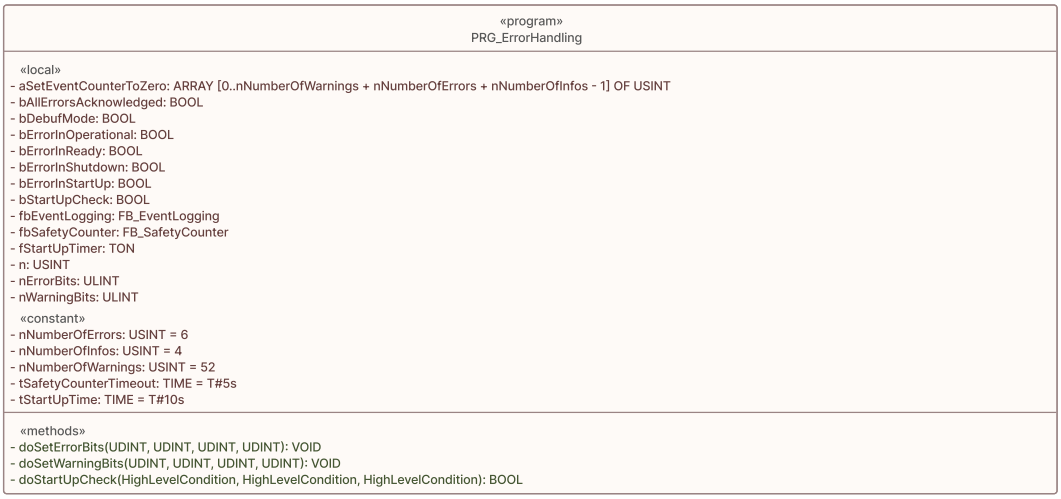

Figure 4.3: Outline of *PRG_ErrorHandling*.

The error bits and warning bits, which are set in the methods *doSetErrorBits* and do *SetWarningBits*, form the central element of ErrorHandling. Each component has an error ID in the form of a *UDINT* variable. One bit is assigned to each error message. The coordination of the error IDs is hierarchical. Subsystems report their error ID to their respective High Level system. ErrorHandling in turn collects the error IDs of the High Level systems and creates a warning word and an error word from them, both in the form of a *ULINT* variable. The ErrorHandling thus takes over the classification of the incoming error messages into warning and error. Errors are always critical and lead to a process stop with a change into the *ERROR* state. As soon as errors occur, ErrorHandling indicates this with a corresponding bit. This is checked in each cycle by *PRG_MainController* and provides in case of an error for the appropriate transitions into the *ERROR* state. The exact logical sequence is explained in one of the following sections.

ErrorHandling has two function blocks dependent on it, which are responsible for additional safety-related functions. *FB_EventLogging* cyclically creates and updates a logging report that records all transitions of the Top Level state machine, all warnings and errors that occur, and all control commands by Operation Control. A list of all possible events is defined in it. The report is stored locally on the PLC in the form of a .csv file and can be retrieved as required for more detailed analyses of error cases.

The *FB_SafetyCounter* function block provides additional protection. It monitors the EtherCAT connection to Operation Control, the four Electronics Boxes, and the own slaves. In the event of a connection failure, an error message is an output after a certain timeout. These are critical errors that lead to a change to the *ERROR* state since safety can no longer be guaranteed if the EtherCAT connection is interrupted. The connection status is checked using a counter, which is increased cyclically. The Main Controller sends its counter to Operation Control and the XMC boards and receives their counters in return. If the counter does not change for a predefined time, a connection termination is indicated.

In addition to the cyclic processing of the errors and the associated monitoring of the system, ErrorHandling performs a system check when the Main Controller is started for the first time or after a change to the *ERROR* state. This is implemented in the method *doStartUpCheck*. The logical processes are described in detail in one of the following sections.

It is to be mentioned that in *PRG_ErrorHandling* also a Top Level state machine is implemented. This follows the state machine of MainController and provokes no own state changes. However, the safety-relevant setting of the error and warning bits as well as the logging and the checking of the safety counters happens outside of the state machine, so that

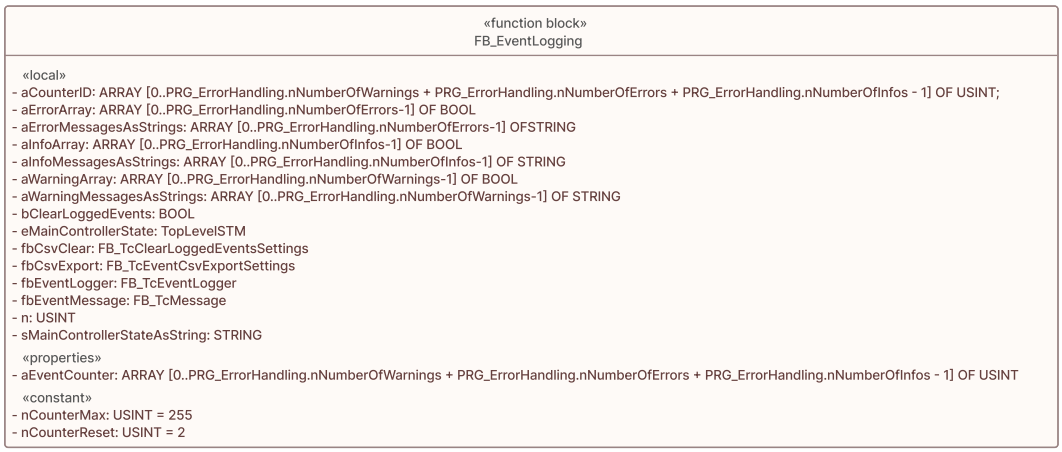

Figure 4.4: Outline of *FB_EventLogging*.

a cyclic run can be guaranteed. Figure fig. 4.3 shows the composition of *PRG_ErrorHandling* and its associated function blocks.

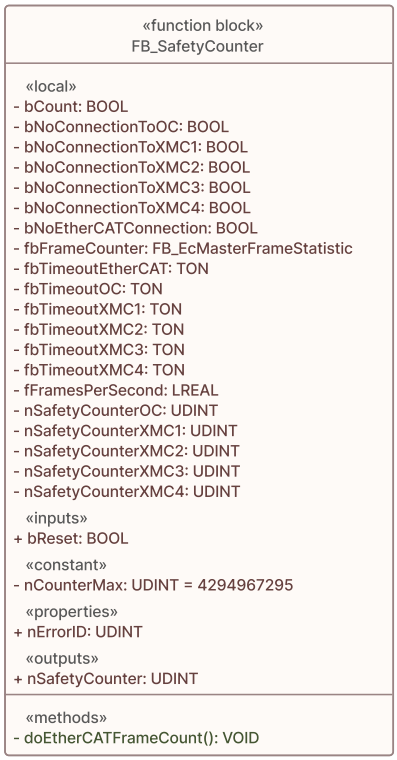

Figure 4.5: Outline of *FB_SafetyCounter*.

4.1.2 High Level

The High Level is the second highest layer in the software architecture of the Main Controller. High Level systems are not real hardware components or controllers but serve as instances that are responsible for specific functionality of the Main Controller. The task spectrum of the Main Controller is roughly divided into three areas: Service Module, Levitation, and Guidance. These are covered by the High Level systems. Therefore, the function blocks of

the respective Sub Level systems are declared and instantiated in them.

High Level systems have a High Level state machine with individual states. The modeling and implementation of these will be discussed in more detail below.

High Level systems take over two important tasks. On the one hand, depending on the state of their state machine, they call the corresponding Sub Level systems and provide them with necessary control instructions. Secondly, they form the interface for data exchange between the Sub Level components and the Top Level system. To handle the large data traffic, a data handling concept is implemented at High Level. This concept provides that inputs for Sub Level systems are stored and processed in IEC 61131-1 properties. MainControl writes incoming control commands and actual values within the method *doDataHandling* to the corresponding property of the High Level system. The set-accessor of the property processes this data and sets the corresponding input variables of the Sub Level system.

Properties have the advantage that they are written cyclically, even if the function block is not executed. This guarantees that current values are always present at the inputs of the Sub Level systems. In addition, properties offer the possibility of already processing data in the first instance. The High Level systems take over, for example, the conversion into the correct unit or the splitting or merging of arrays.

Outputs from Sub Level systems are processed in the same way using the get-accessor of a property. High Level systems thus collect all process data of their Sub Level systems and allow MainControl access to it. They do not have direct access to the data internally, which in turn protects them from unwanted overwriting. Data that is important for the internal logic of the High Level system is written to a local variable of the High Level system using the set-accessor. Thus they have access to the process data of the Sub Level systems, but cannot change these. The use of the properties brings still further advantages with itself here. The explicit use of the get-accessor or set-accessor protects variables against unauthorized write access. In addition, the modularity is increased since properties can be exchanged easily, moved, or replaced.

The properties of a High Level system are thus responsible for the further transfer of process data and guarantee that always only the desired value is transferred. If you compare the concept with a physical analogy, properties indicate how a High Level system must be wired so that current can flow.

Process data relevant to the High Level systems are written directly to the respective inputs of the function block. This includes above all the state of the own state machine and the activation status. State changes in High Level state machines are only ever forced by the Main Controller. High Level systems can therefore not change their state themselves.

The activation status indicates whether the High Level system is enabled for operation at all by Operation Control. If it is deactivated, the High Level System is set to sleep mode. It is still called cyclically, but no code is executed and no behavior is checked by ErrorHandling.

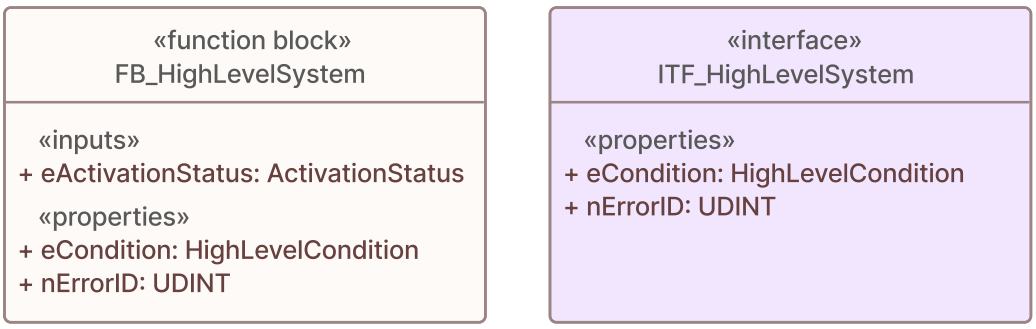

Figure 4.6: Outline of *FB_HighLevelSystem* and *ITF_HighLevelSystem*.

High Level systems are standardized and have the same basic structure. This is implemented in the sense of object-oriented programming [30] in an abstract base function block. The base function block specifies the internally required variables, such as the variable *eActivationStatus*. If new functionality is added to the High Level systems, common variables can be added here. In addition, the basic function block implements the *ITF_HighLevelSystem* interface. Properties and abstract methods are predefined in it, which in turn can be individually programmed out in the respective High Level System. The High Level function blocks extend the abstract base function block and inherit its attributes, methods, and properties. Like the interface, the base function block does not yet contain any implementations. These are only implemented individually in the respective High Level system. In the context of this project, the High Level systems obtain the two important properties *eCondition* and *nErrorID*, which are mirrored from the interface. These are of safety-relevant meaning and show *PRG_ErrorHandling* the system status and the occurring errors.

By using OOP, implementation is mandatory, which ensures safety. Figure fig. 4.6 shows the two objects of object-oriented programming that are used at High Level.

ServiceModule

The *FB_ServiceModule* function block is responsible for all components that are physically located on or in the Service Module. These include a distance sensor and two LED lights that are to indicate the status of the IPC box and the PLC. The distance value is passed on cyclically to Operation Control. Figure fig. 4.7 shows how the High Level system ServiceModule is structured.

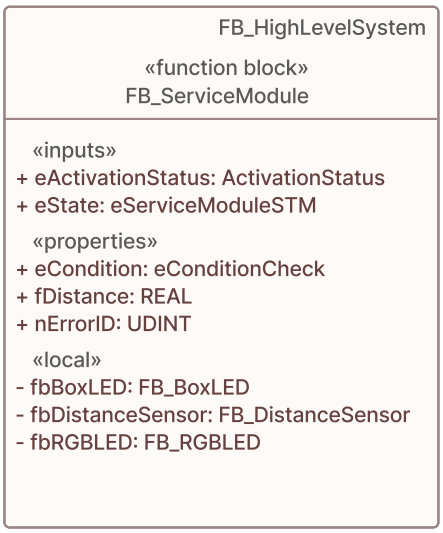

Figure 4.7: Outline of *FB_ServiceModule*.

The structure of FB_ServiceModule clearly shows how a High Level system is structured. It has the inherited variables and properties of the base function block, the locally instantiated Sub Level systems, and finally a further property that is used for the data transfer from the Sub Level to the Top Level. The system itself has no internal variables, but only takes over the coordination of its Sub Level systems.

LevitationSystem

The most comprehensive High Level system of the Main Controller is LevitationSystem. It coordinates the control of the levitation height and the transition process. For this task,

FB_LevitationSystem instantiates the two important Sub Level systems *FB_TrajectoryGenerator* and *FB_EqualizingController*. *FB_TrajectoryGenerator* is available in four versions, once for each XMC board. LevitationSystem cyclically supplies these two controllers with the corresponding input data, consisting of actual values and desired values. The outputs are further processed via property and returned to MainController.

Figure 4.8: Outline of *FB_LevitationSystem* and *FB_GuidanceSystem*.

LevitationSystem also has additional internal variables that are responsible for activating the current controllers on the XMC boards. These are set internally and given as output back to MainController, where they are cyclically passed to the XMC boards. In addition to activation, the operation mode of the controllers is also set based on the incoming control commands. This happens in the *doSetOperationMode* method. The *doCalculateFmagValuesForEC* method calculates the actual magnetic force acting in the z-direction from the incoming sensor data from the XMC boards. It is then used as input for the Sub Level system EqualizingController.

The method *doSetPresetTGValuesForBackToGround* contains predefined reference values that are relevant for the trajectory calculation for a transition back to ground level. In case of error or disconnection to Operation Control, this method can be used to achieve a safe termination of the levitation process.

LevitationSystem has three locally defined variables that are set by properties. They are used internally for calculations and are not visible from the outside. This ensures that operationally relevant process data does not exist more than once on a global level.

GuidanceSystem

The third High Level system is *FB_GuidanceSystem*. It is responsible for the Sub Level systems *FB_GuidanceController*, instantiated once per XMC board, and *FB_AccelerationSensor*. The latter is instantiated twice, each responsible for an acceleration sensor mounted in the Service Module.

GuidanceSystem is similar in structure to LevitationSystem and contains, in addition to its inherited attributes, mainly properties that are responsible for data forwarding. The setting of the operation mode and the activation of the current controllers is done in the same way as in *FB_LevitationSystem*. The *doSetOperationMode* method is responsible for this.

Unlike LevitationSystem, *FB_GuidanceSystem* has no locally defined variables for internal calculations. Figure fig. 4.8 shows the composition of *FB_GuidanceSystem*.

4.1.3 Sub Level

The Sub Level forms the lowest layer of the software architecture of the Main Controller. The associated function blocks either correspond to real components, such as sensors and actuators or are based on imported Simulink models from which PLC code has been generated. Sub Level components are therefore the only modules of the Main Controller that are connected to the clamps connected to an EtherCAT coupler.

Sub Level systems have a standardized structure similar to High Level systems. In the context of the object-oriented design, they receive their attributes from the interface *ITF_SubLevelSystem* fig. 4.9. It defines the two important properties *eCondition* and *nErrorID*. The implementation is done individually in the respective modules. *eCondition* records the current condition of the function block and passes it on to the High Level system. The condition is determined by the error ID and indicates whether the function block is initialized and ready to start.

Each error message is assigned to a bit that is set as soon as an error occurs. *nErrorID* is also passed on to the High Level system.

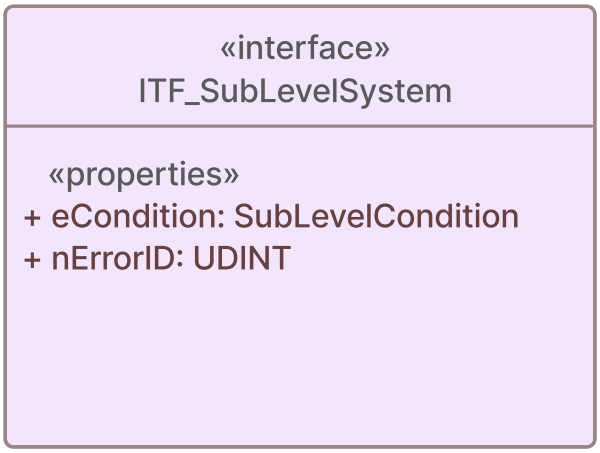

Figure 4.9: Outline of *ITF_SubLevelSystem*.

Each Sub Level system contains its potential error messages as locally declared *BOOL* variables, which are set in addition to the corresponding bits of the Error ID in the event of an error. This facilitates debugging.

Sub Level systems are at the other end of the data processing chain. They calculate new reference values from setpoints and control commands, which in turn are distributed to the participants of the control system. Incoming data are taken as input variables, while outgoing data are written as output variables. Since sub Level systems are always instantiated locally in a higher-level High Level system, inputs and outputs can only be written and read by the latter. Thus, there is no direct exchange between the Top Level and the Sub Level.

DistanceSensor

The *FB_DistanceSensor* subsystem receives the measured values of a distance sensor attached to the Service Module. The distance sensor communicates via RS422. A subordinate function block *FB_SerialComRS422* converts the incoming data into a distance value and checks the validity via CRC checksum. The clamp is the EL6021 model from Beckhoff. The inputs and outputs of the RS422 clamp are linked with variables of a global variable list. *FB_DistanceSensor* writes these to the inputs of *FB_SerialComRS422* and receives the measured distance value in return. The distance value is subjected to a range check in the *doCheckRange* method, which sets the corresponding error bit if it is invalid. A further error bit indicates a communication error during the processing of the RS422 data by *FB_SerialComRS422*.

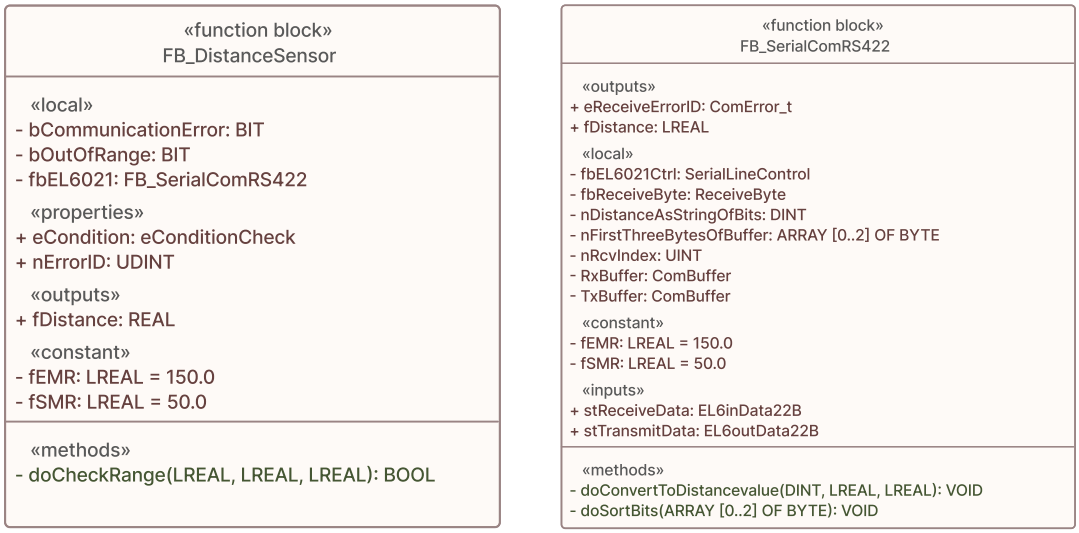

Figure 4.10: Outline of *FB_DistanceSensor* and the subordinate *FB_SerialComRS422*.

BoxLED

The *FB_BoxLED* subsystem is responsible for controlling an LED light. The LED light is located on the outside of the IPC box and displays its status. The LED can be controlled with different operation modes, which are implemented in the form of a method in *FB_BoxLED*. Depending on the desired mode, the subsystem sets the corresponding intensity and light duration. The intensity is proportional to the definition range of an *INT* variable. 100% luminosity corresponds to a value of 32767, no luminosity corresponds to the value 0. The luminosity can be dimmed using a factor. The dimming factor and maximum intensity, as well as different frequencies, are defined as constants in *FB_BoxLED*.

The intensity value is written by *FB_BoxLED* to a global variable, which in turn is linked to the input of the clamp, model EL2564. The outputs of the terminal are error status messages. These are also linked to a global variable and set the error bits declared locally in *FB_BoxLED*. The methods *doSetClampValues* and *doGetClampValues* are responsible for writing and reading the clamp values.

RGBLED

The *FB_RGBLED* subsystem controls three different colored LEDs, all connected to one terminal, model EL2564. The LEDs are located inside the IPC box and indicate the status of the top-level state machine.

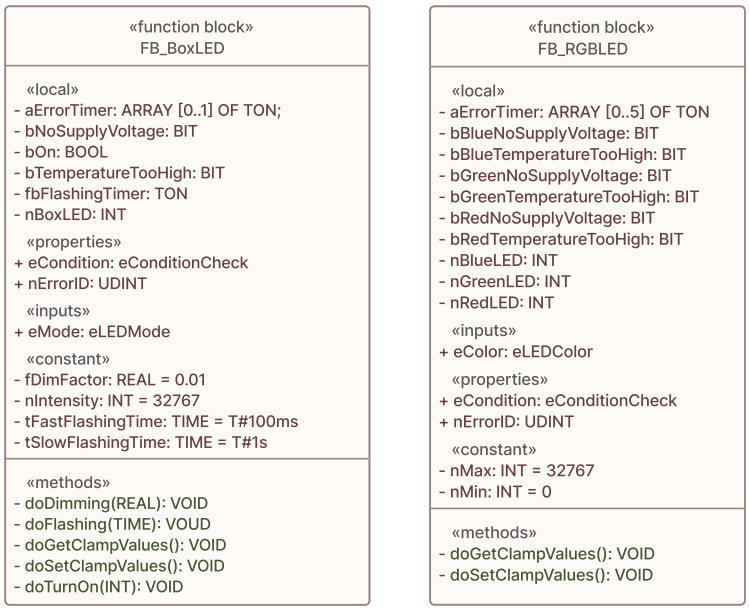

Figure 4.11: Outline of *FB_BoxLED* and *FB_RGBLED*.

Based on the incoming desired color, the intensity of the corresponding LED is set to maximum. The luminosity is defined as a constant variable in *iMAX*. The reading and writing of the terminal values are done in the same way as in *FB_BoxLED* with the methods *doSetClampValues* and *doGetClampValues*. Two error bits per LED indicate a malfunction at the terminal. The temperature bit is triggered as soon as the temperature at the LED exceeds 80 °C.

TrajectoryGenerator

FB_TrajectoryGenerator is the largest system at Sub Level. It is responsible for calculating a trajectory that provides reference values for the transition process. The TrajectoryGenerator (TG) is a controller implemented on Simulink, which is built into the Main Controller as a black box. This means that the Main Controller does not contain any internal logic for computation, but only provides control commands and actual process data.

At the time of this work, the TrajectoryGenerator is still under development, which is why we are working with a provisional dummy version. This already contains all interfaces for control, but no logic for calculating reference values.

The TG gets its operation mode, as well as actual values and desired values from its inputs. In addition, there is the *BOOL* variable *bTransition*, which activates the controller at the beginning of a transition. The TG then calculates a trajectory once based on initial values and desired end values. This happens within the dummy methods *doSetInitialValues*, *doSetDesiredValues*, and *doCalculateTrajectory*. A counter prevents only the actual values at the beginning of the transition are read in. After calculating the transition, a timer counts up the time until the desired transition time. Based on the current timer value the associated new reference values are cyclically put out as output. This is done by the dummy method *doGetTrajectoryValues*.

The TG already performs an internal range check for each calculated new reference value and sets the corresponding error bit in the event of an error.

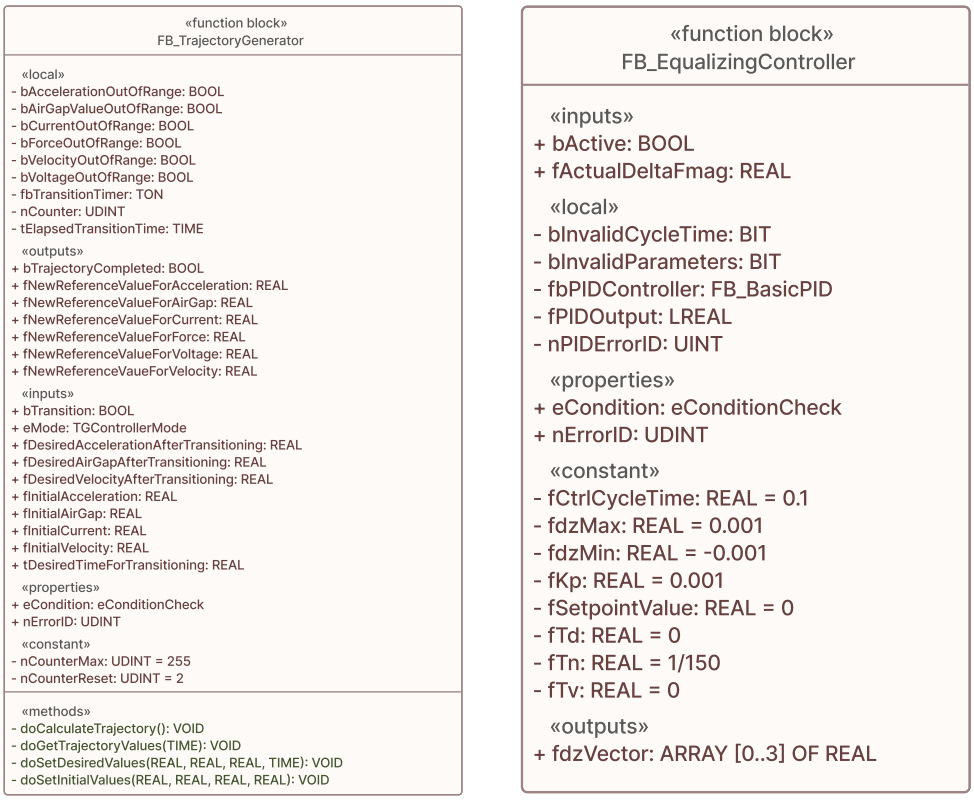

Figure 4.12: Outline of *FB_TrajectoryGenerator* and *FB_EqualizingController*.

EqualizingController

The EqualizingController (EC) is also responsible for controlling the levitation process. Unlike the TG, however, it is not active during the transition, but only while the air gap height is maintained. The EC is also a controller implemented on Simulink, which is inserted into the Main Controller as a black box. Locally declared variables thus originate in Simulink. Inputs and outputs, on the other hand, originate from the Main Controller implementation.

The EC is activated after the transition phase is completed by setting the variable *bActive*. The estimated value of the magnetic force acting on each of the four XMC boards is used as an input for calculating new reference values for the levitation distance of each EB. The output vector consists of four distance values that are added to the levitation height at the four corners of the Service Module, preventing the pod from tilting.

The error bits of the EC are connected to the locally implemented PID controller. In the event of incorrect initialization, it reports the corresponding malfunction.

GuidanceController

The GuidanceController (GC) is responsible for controlling the lateral movement in the y-direction. It ensures that the pod is always kept centered and does not tilt laterally.

The GC contains a complex controller structure that is implemented in Simulink and imported as a black box. Since it is still under development at the time of this work, we use a provisional dummy model that already contains all the necessary interfaces in the form of output and input variables, but no internal logic for calculating new reference values yet. The logic is provided by the dummy methods *doCalculateNewReferenceValueForCurrent* and *doCalculateNewReferenceValueForCurrentAndForce*. The two methods correspond to the two possible operation modes involving calculations based on different input data. The output

variables are the same in both cases and are forwarded cyclically to the higher-level system.

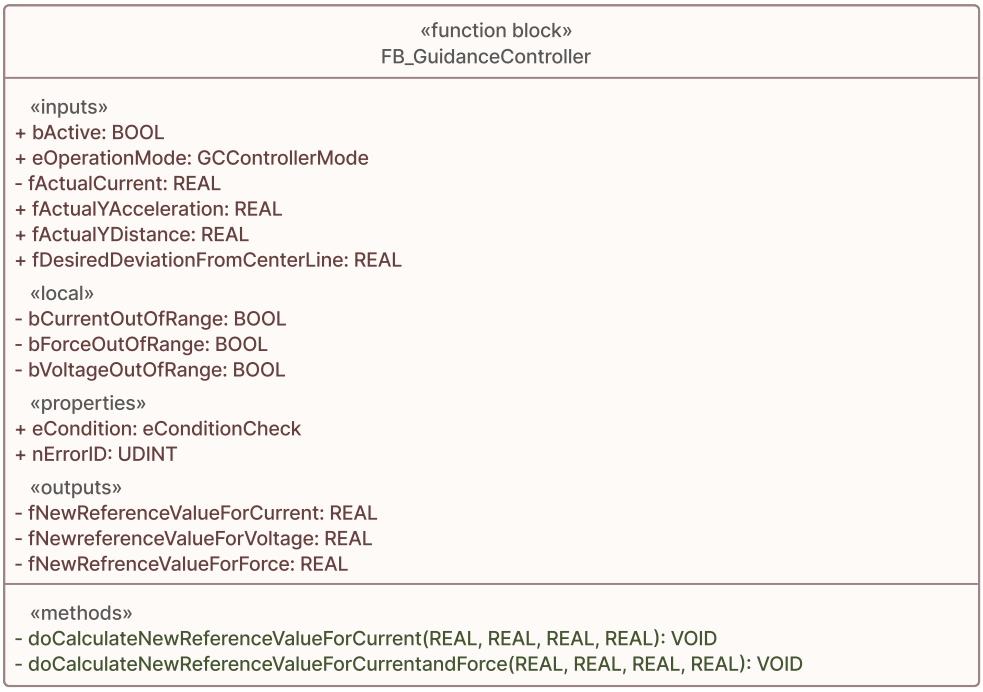

Figure 4.13: Outline of *FB_GuidanceController*.

AccelerationSensor

The Sub Level component *FB_AccelerationSensor* receives data from an Inertial Measurement Unit (IMU) and passes the acceleration values in all three spatial directions to its high-level system. The sensor model is called Orientus from the company Advanced Navigation.

The sensor communicates via RS232 with a Beckhoff clamp, model EL6001, which is connected to a coupler, model EK1100. *FB_AccelerationSensor* forwards the incoming clamp data, which are linked to a global variable, to the subordinate function block *FB_SerialComRS232*. This processes the RS232 data packets into readable acceleration values, which are returned to *FB_AccelerationSensor*. There they are subjected to a range check using the *doCheckRange* method. If it fails, the corresponding error bit is set. The other error bit indicates whether there is a communication error within the RS232 processing.

FB_AccelerationSensor has implemented a local packet counter that can optionally be operated for test purposes. It starts with activating the method *doPacketCount*, which measures the incoming RS232 packet rate. Based on this the latency of the IMU unit can be minimized. Tests have shown that with a cycle time of 1 ms, a stable packet rate at a frequency of 250 Hz is achieved. Figure fig. 4.14 shows *FB_AccelerationSensor* and the associated function block for the serial communication interface.

4.2 Top Level State Machine

This subsection describes the main state machine implemented on the Main Controller. It runs at the Top Level in *PRG_MainController* and *PRG_ErrorHandling*. To distinguish it from other state machines in lower-level function blocks, the state names are written in uppercase.

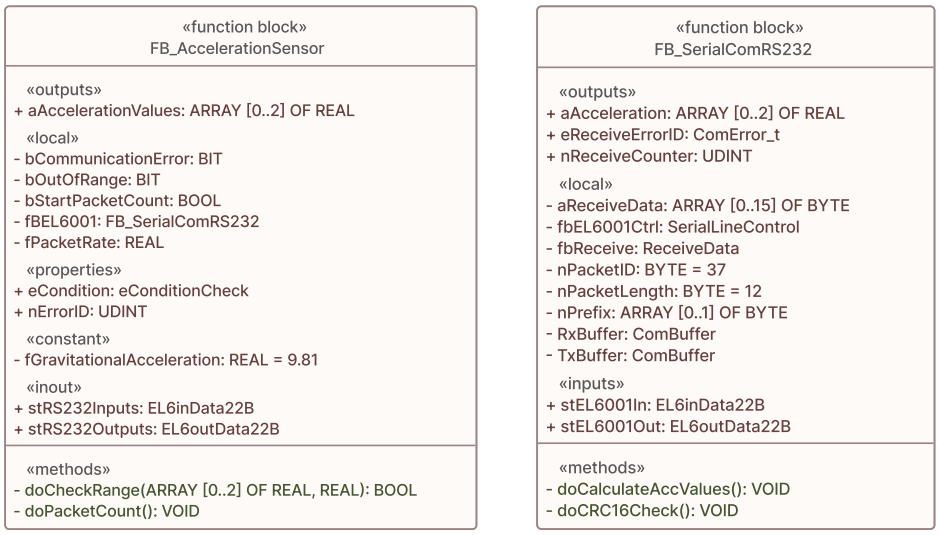

Figure 4.14: Outline of *FB_AccelerationSensor* and the subordinate *FB_SerialComRS232*.

The states of the Top Level state machine are *STARTUP*, *READY*, *OPERATIONAL*, *SHUTDOWN*, and *ERROR*.

The modeling of the Top Level state machine is done using an activity diagram. There are thus two activity diagrams, which are presented in the following. In addition, the data handling is dealt with again in detail.

4.2.1 Main Controller

The activity diagram in figure fig. 4.15 describes the general behavior of the Main Controller. The yellow activities correspond to the states of the Top Level state machine. All actions that are within the activities show the logical processes that take place at High Level. Transitions are already close to the implementation but do not necessarily correspond to the exact nomenclature. Rather, they should be understandable from the outside and clarify the logic of the Main Controller.

The starting point of the activity diagram is the pseudo-state *ON*, which is left as soon as the Main Controller is supplied with power. The actual startup state of the Main Controller is *STARTUP*. This state is used to initialize the system and perform a comprehensive system check. Two important actions are processed in this state.

First, the High Level systems are activated. By default, all systems are always activated after a startup. However, for testing purposes, Operation Control can activate or deactivate each of the three High Level systems individually. As soon as a system is deactivated, its condition is set to SLEEP. This means that the High Level system is still called, but no more code is executed. It is also no longer included in the cyclic condition check.

In *STARTUP* a tightened variant of the condition check takes place. Unlike the check during normal operation, non-critical error messages also cause a change to the ERROR state. This is to ensure that the Main Controller only changes to the operating mode if it is fully functional. When entering the *STARTUP* state a timer is started. Within this time, all High Level systems and their subordinate hardware components must show that they are responsive and error-free. Subsystems must have the *RUN* condition for this. In the event of an error, they have the *ERROR* condition. High Level systems show *INIT* until all their subsystems have switched to *RUN*. If this is the case, the High Level system also displays the condition *RUN*. As soon as all High Level systems indicate *RUN* or *SLEEP*, the startup check is

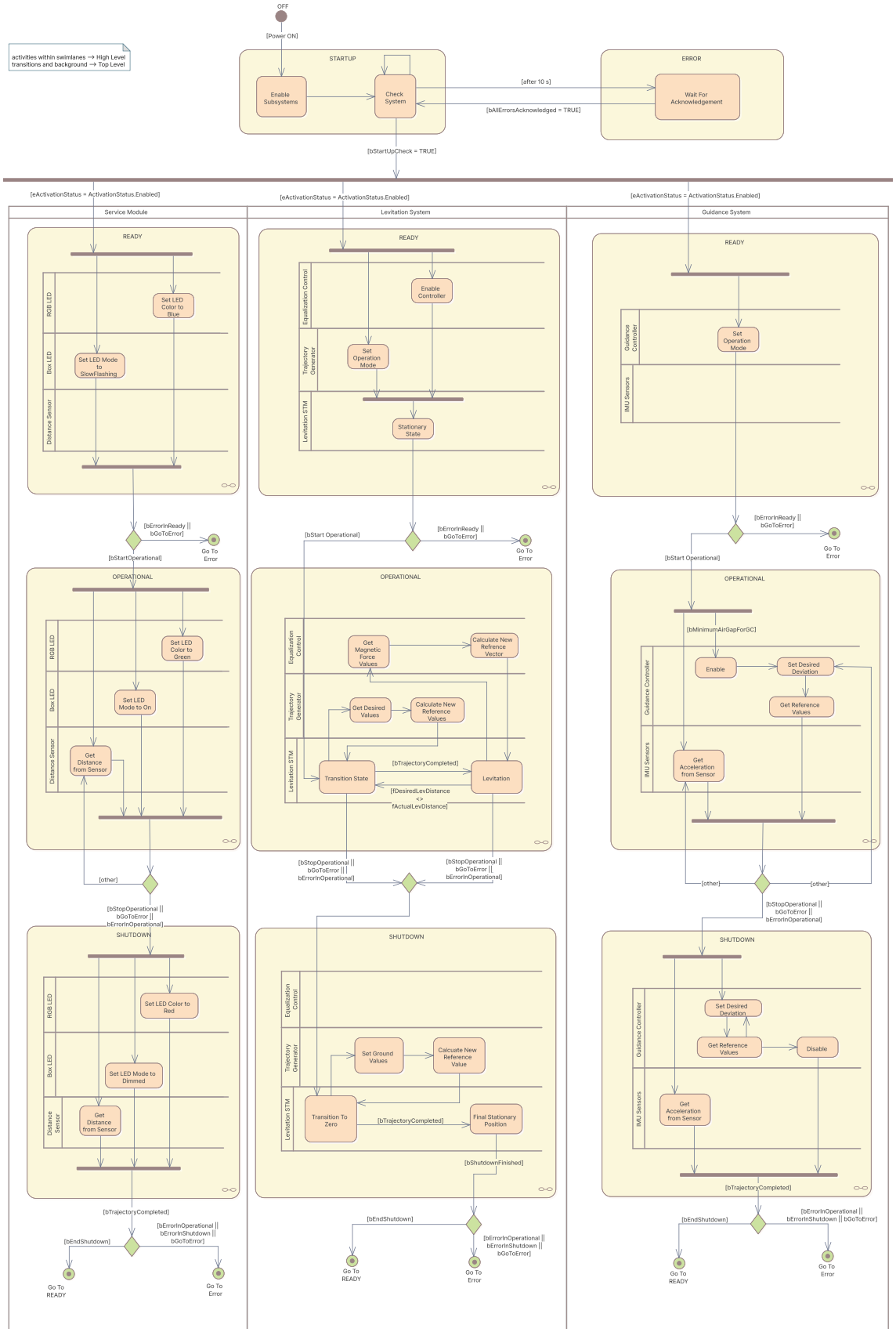

Figure 4.15: Activity Diagram of the Main Controller.

positive and the transition to the *READY* state is released.

If, after the timer has expired, not all High Level systems show *RUN*, the startup check is negative and the system switches to the *ERROR* state. The duration of the timer can be set in *PRG_ErrorHandling* and is 10 s by default.

The main task of the *STARTUP* state is to let the system start up and to detect malfunctions already before the start of operation.

After a successful startup check, the system switches to the *READY* state. For further modeling, the diagram is divided into swimlanes. Each swimlane contains a High Level system and is thus intended to illustrate which processes take place in parallel.

The *READY* state is characterized by the fact that the pod is not yet moving, but is stationary at ground level. It is therefore a stable state in which the occurrence of errors cannot lead to hazards for the hardware.

The LEDs always indicate the current state of the Top Level state machine and are controlled accordingly by the responsible High Level system *FB_ServiceModule*. The distance sensor and the acceleration sensors are also already running, but their values are not yet processed or used in any other way. Based on the control commands given in the GUI during startup, the operation mode of the included Simulink controllers TrajectoryGenerator and GuidanceController is set. The selection of the operation mode is always only possible in the *READY* state and cannot be changed during operation. The parameters of the EqualizingController are initialized in parallel. These are already set by default in the *FB_EqualizingController* but can be changed in maintenance sections. This means that initialization is performed each time the *READY* state is entered so that a problem-free process can be guaranteed.

The *READY* state is maintained until either the start signal is received from Operation Control or an error occurs. In case of an internal error, *PRG_ErrorHandling* provokes a change into the *ERROR* state. If an error occurs outside the Main Controller or if OC changes to error mode, *PRG_MainController* receives the external instruction to change to *ERROR*. Only if these two error bits are negative and a start instruction by OC is present, the READY state is left in the direction of *OPERATIONAL*.

In case of a complete shutdown of the system *READY* is the recommended state since the system is in a safe state and thus no damage is caused to hardware and humans.

OPERATIONAL is the main operating state of the Main Controller. It covers all transition processes and levitation operations. Because the Pod is in motion throughout, *OPERATIONAL* is not a stable state. This means that no direct change to the ERROR state is possible. The system will remain in the *OPERATIONAL* state until the signal to stop the operation is given by OC. In this case, *bStopOperational* is set and the transition to *SHUTDOWN* is initiated.

The most important processes in *OPERATIONAL* are mainly the control of the transition trajectory and the lateral distance control. The High Level system *FB_LevitationSystem* runs through the following logic. First, a check is made to determine whether a transition to a new levitation height is required. If this is the case, actual values and desired reference values are passed on to the TrajectoryGenerator. It calculates new reference values, which are passed on to the other participants in the control system. This process is repeated cyclically until TG reports that the calculated trajectory is completed and the pod is at the desired levitation height. The system then switches to hover mode and activates the EqualizingController. It calculates correction values from the magnetic force acting in the z-direction, which is added to the levitation target distance. This process is also repeated cyclically until either the transition mode is switched to again or a landing process is instantiated.

In parallel, the GuidanceController is activated at a certain minimum hover height. In the currently implemented dummy variant, the command is given externally by OC. When the command is received, the GuidanceController is activated and a signal is passed to the XMC

boards. The GuidanceController then calculates new reference values for current, voltage, and optionally force based on the sensor values of the XMC boards and the desired deviation from the centerline. These are returned to the XMC boards. This is repeated cyclically.

The subsequent state is always *SHUTDOWN*. This state includes the safe landing of the pod on the ground and a return to a stable starting position. Thus, the transition process back to ground level is given its own Top Level state. This is an additional safety precaution so that damage to hardware and people can be ruled out. Theoretically, a transition process at ground level in *OPERATIONAL* is also possible. However, this is not accompanied by an operation stop. For a change back to the *READY* state, *SHUTDOWN* must therefore always be run through. Another safety-promoting aspect is that no input values for calculating the trajectory are required for the transition process in *SHUTDOWN* since these are already pre-implemented. Thus, a safe landing can be guaranteed.

In *SHUTDOWN*, the corresponding LED colors and modes are set first, and sensor values continue to be read out. The levitation system takes over the transition calculation. Apart from the pre-implemented setpoints, this is done identically as in *OPERATIONAL*. The GuidanceController remains active until the minimum hover height is undershot again. The termination of the *SHUTDOWN* state is determined by the TrajectoryGenerator. It signals when the transition process has been completed and ground contact has been regained. *PRG_MainController* then signals OC that the shutdown process has been completed. If a critical error has occurred during *OPERATIONAL* or *SHUTDOWN* or an error instruction from OC is present, the system changes to the *ERROR* state at this point. If not, the Main Controller waits for permission from OC to change back to *READY*. The permission is given by setting the bit *bEndShutdown*.

SHUTDOWN is a bi-stable state. After completion of the transition process, a stable initial position is reached again. In this position, the system can also be switched off without any problems. However, it is recommended to wait for the change to *READY* to be able to analyze potential error messages.

The *ERROR* state is outside the swimlanes. This is because High Level systems are no longer called in it. Thus, the condition and the error ID of the systems are no longer updated. This facilitates debugging and error analysis. The *ERROR* state is a pure waiting state that is provided for analysis purposes. It is only exited after the command is given from OC that all errors have been cleared and acknowledged. This is followed by a change back to *STARTUP*. A change to the *ERROR* state thus always results in a reinitialization of the system. This ensures that the system works error-free again.

4.2.2 Error Handling

The Top Level state machine is also implemented in *PRG_ErrorHandling*. However, all state changes are executed in *PRG_MainController*. This means that *PRG_ErrorHandling* imitates the behavior of *PRG_MainController*. The two program modules are thus always in the same state. Effectively, the Main Controller is limited to only one Top Level state machine.

PRG_ErrorHandling is responsible for all safety-relevant functions and checks. These take place every cycle and are thus outside the actual implementation of the state machine. Within the states, only the error bit corresponding to the state is set, if the error ID of the Main Controller indicates the occurrence of errors. These error bits indicate in which state the error was triggered and are checked in the transition queries in *PRG_MainController* whether a change to the *ERROR* state is required.

Two exceptions are the *STARTUP* state and the *ERROR* state. In the *STARTUP* state, the already mentioned startup check is executed with the method *doStartUpCheck*. It checks the condition of all High Level systems and their hardware components. *PRG_MainController*

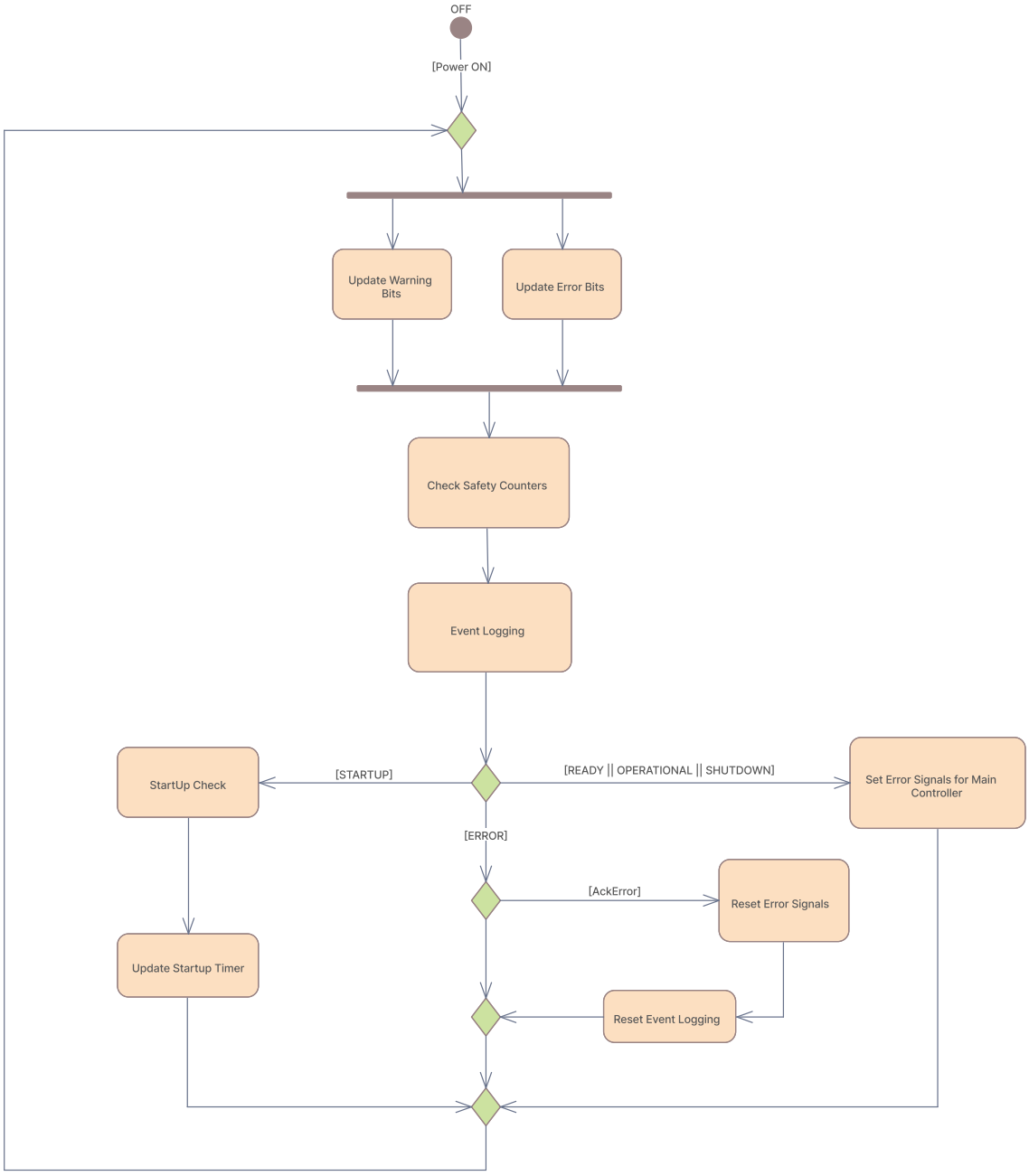

Figure 4.16: Activity Diagram of *PRG_ErrorHandling*.

learns about the success of the startup check by reading the corresponding error bits and changes accordingly to either ERROR or READY.

In the *ERROR* state, it waits for the variable *bAckError* to be set by OC. Then the error ID and the warning ID are reset, the logging and the safety counters are reinitialized and the state-dependent error bits are reset as well. Finally, *bAllErrorsAcknowledged* is set. This shows *PRG_MainController* that a change to *READY* is possible.

In *ERROR* no further systems are executed. The errors and warning IDs that led to the change to *ERROR* remain unchanged. The activity diagram in Figure fig. 4.16 shows the basic logic by which *PRG_ErrorHandling* runs. The most important actions are updating and setting the warning bits and the error bits. This happens in the two methods *doSetWarningBits* and *doSetErrorBits*. For this, the error IDs of all High Level systems are collected and written to the warning bits and error bits of the Main Controller.

The *FB_SafetyCounter* function block, which is responsible for checking the connection to the other nodes of the control system, is executed in parallel. Any errors that occur are also included in the two methods for setting the error bits. Subsequently event logging is performed. This is outsourced in the function block *FB_EventLogging*. The block registers when and if an error, a warning, a change of state, or an error acknowledgment occurs and creates an entry in the log file.

In combination with the setting of the error bits the cyclic sequence of *PRG_ErrorHandling* results. The startup check is to be distinguished from the normal mode, because here instead of the error IDs the condition of the systems is checked. In the normal mode, only the error IDs are decisive for the conditioning of the system.

Event logging outputs stored error messages that provide a brief description of the error. Each error message also has a unique ID. When the system is extended, it is mandatory to update these IDs so that event logging can continue to work correctly. A list of all error messages can be found in table A.1 in the appendix.

4.2.3 Data Handling

Data handling is of central importance for the implementation of the Main Controller since it serves as an exchange link between Operation Control and the XMC boards. This results in a large amount of data that must be processed cyclically. In the development of the controller architecture, we, therefore, paid special attention to the implementation of a suitable system for data processing.

The Main Controller sends and receives all data in packets, each consisting of a *STRUCT* variable. A distinction is made between packets from the data exchange with Operation Control and packets from the exchange with the Electronics Boxes. Consequently, there are two global variable lists whose *STRUCT* variables are each connected to the corresponding EtherCAT inputs and outputs.

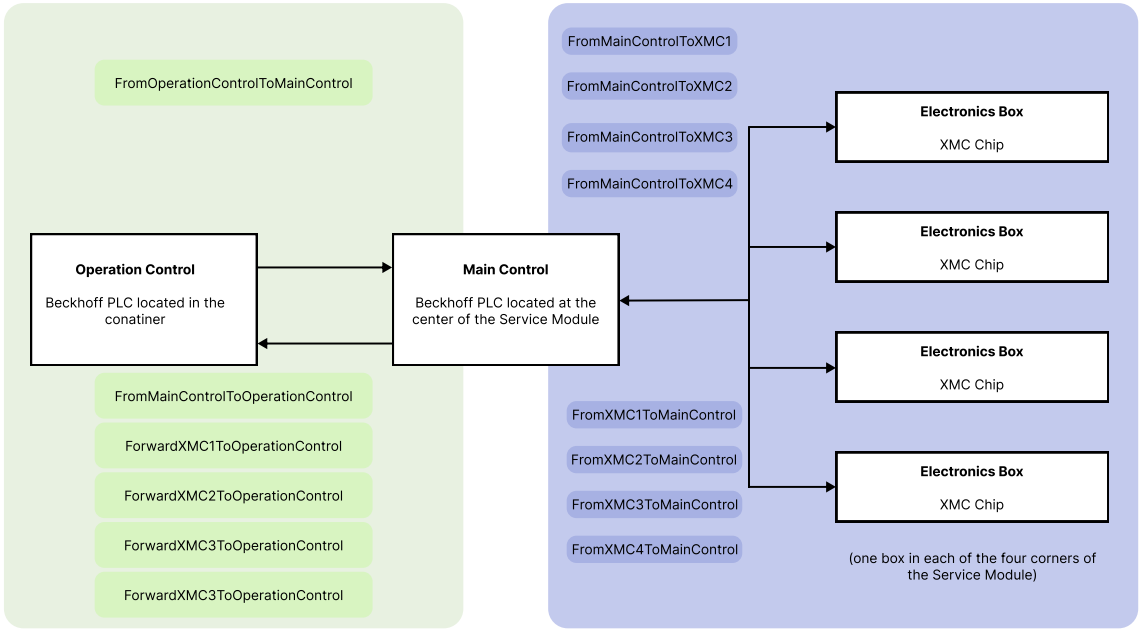

Figure 4.17: Overview of the data handling structure.

The global variable list for Operation Control contains the *STRUCT* variables in the green shaded area. The corresponding global variable list for the data exchange with the EBs

contains the *STRUCT* variables in the blue shaded area in figure fig. 4.17.

The access to these *STRUCT* variables takes place only in *PRG_MainController* within the method *doDataHandling*. This applies equally to writing and reading the variables.

Control commands from XMC boards and OC that are relevant at Top Level are first written to two further *STRUCT* variables. These are subsequently used within the Top Level state machine. The remaining input data from OC and the XMC boards are distributed directly to the corresponding properties of the High Level systems. Thus, these are instantiated at any time, even if the High Level system is not running.

Outputs are also derived directly from the properties of the High Level systems. Additional control status outputs are taken from *PRG_MainController* and *PRG_ErrorHandling*.

The Main Controller also forwards all incoming data from the XMC boards to Operation Control so that it can be further evaluated there.

For the variables that are exchanged between the participants of the control system, a universal naming convention applies that allows conclusions about the origin and destination of the variable. An example is the following variable:

OM_origG_LevitationDisable

OM	–	from Operation Control to Main Control
origG	–	variable originates from GUI
LevitationDisable	–	name of the variable

Table 4.1: Naming convention of global variables.

4.3 High Level State Machines

The behavior of the High Level systems is modeled by state diagrams. On each system, a state machine with individual states is implemented, which is illustrated by the state diagram. High level state machines are characterized by the fact that instructions for state changes always come from external, i.e. from a Top Level system. Moreover, the states can always be assigned to a Top Level state. To distinguish from the Top Level state machine, the states are always named by a progressive form. All High Level state machines have the initial state *Idling*. This is a pseudo-state in which no actions are executed. It is used to initialize the system and is invoked when a startup check is performed in the Top Level state *STARTUP*. In it, subsystems belonging to real hardware components, are called to perform a condition check. The High Level state machine is not executed when the system is disabled by Operation Control.

The modeling of the behavior is divided into entry-, do- and exit-actions. Entry-actions are usually the setting of inputs of Sub Level systems. They are executed once when entering the state. Do-actions are repeated continuously as long as the user remains in the state. They usually correspond to calling subordinate function blocks and executing local methods. Exit-actions are executed once when leaving the state. They are usually also linked to the setting of inputs.

High Level state machines are thus a more finely structured version of the Top Level state machine, whose states are tailored to the real behavior of the subsystems.

The state machines and the respective state diagrams of the three High Level systems of the Main Controller are presented below.

4.3.1 Service Module

Figure fig. 4.18 shows the state diagram of the High Level system ServiceModule. The state machine consists of the pseudo-state *Idling* and the further states *Waiting*, *Running*, *PostRunning* and *PreparingForError*. *Waiting*, *Running* and *PostRunning* are called in the Top Level states *READY*, *OPERATIONAL*, and *SHUTDOWN*. In *Idling* both *FB_DistanceSensor* and the two function blocks of the LEDs are called because they are connected to hardware components. However, the LEDs do not yet receive any control commands.

PreparingForError takes a special position. The state is run through once before changing to the *ERROR* state to set the LEDs to error mode. This is always followed by a change to *Idling* since the system is always reinitialized after the occurrence of a critical error.

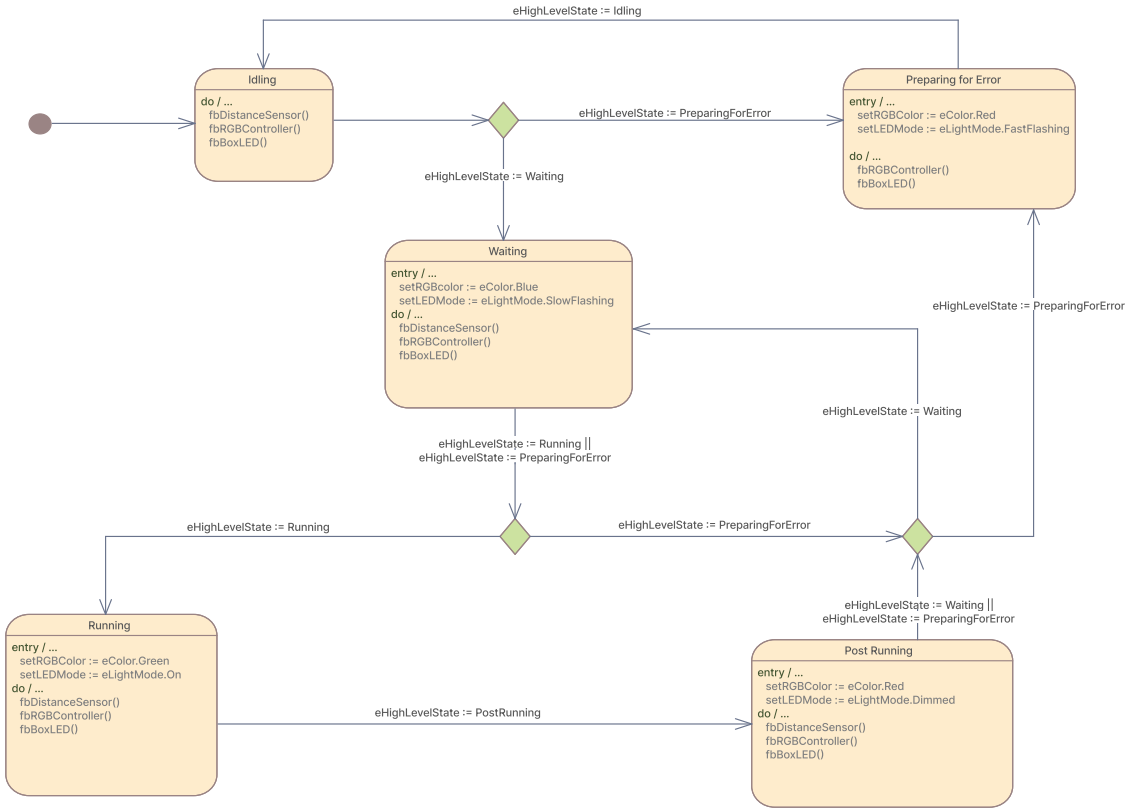

Figure 4.18: State Diagram of the High Level system ServiceModule.

4.3.2 Levitation System

The High Level system LevitationSystem has the states *Idling*, *Stationary*, *Transitioning*, *HoldingAirGap*, and *TransitioningToZero*. The states are strongly based on the behavior of the levitation functionality and partially subdivide Top Level states.

Idling is an empty state since no verifiable hardware components are connected to the system. It is modeled anyway for completeness to make the system accessible for extensions.

Stationary corresponds to the Top Level state *READY*. In it, the operation mode of the TrajectoryGenerator is set. *FB_EqualizingController* is already called to perform the parameterization of the PID controller.

The change to *Transitioning* takes place in the Top Level state *OPERATIONAL* as soon as it is indicated that there is a new trajectory that has not yet been completed. When entering the

state the current controller is activated, when leaving it is deactivated again. Before exiting, the last distance value of the calculated trajectory is set as the hover height.

The *HoldingAirGap* state is also assigned to the Top Level state *OPERATIONAL*. In it, the EqualizingController is executed. As soon as a new desired hover height is available that deviates from the current hover height, the system switches back to *Transitioning* and deactivates the EqualizingController again.

TransitioningToZero corresponds to the Top Level state *SHUTDOWN*. The current controller for the trajectory control is activated once and deactivated again when exited. In addition, the setpoints for a transition back to ground level are set once upon entering the state. When the process is complete, the change back to *Stationary* occurs.

Note that the state diagram has no endpoint. This is because a process stop in any state can be caused by the intervention of an external person. Otherwise, the program is cyclically designed for long-term operation.

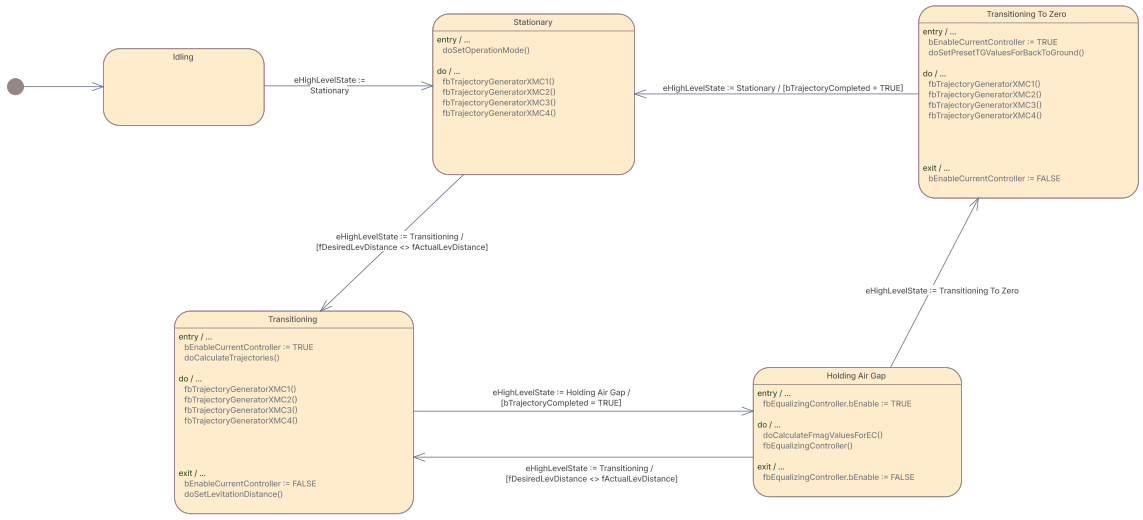

Figure 4.19: State Diagram of the High Level system LevitationSystem.

4.3.3 Guidance System

The state diagram of GuidanceSystem can be seen in Figure fig. 4.19. It consists of the *Idling*, *PreRunning*, *Running*, and *PostRunning* states.

The function blocks containing the acceleration sensors are executed continuously. *PreRunning* is called as long as the lateral distance control is not yet activated. Activation occurs in the Top Level state *OPERATIONAL* when the minimum hover height is reached. When entering the *Running* state, the current controller for the guidance control is activated once. When exiting, it is deactivated again. *PostRunning* is entered when the system is in *SHUTDOWN* and the minimum air gap for the guidance controller is again undershot. This means that no more control takes place in the *FB_GuidanceController* function blocks. The difference to the *PreRunning* state is that there the operation mode of the guidance controller is set again when the state is entered. This is not possible in PreRunning.

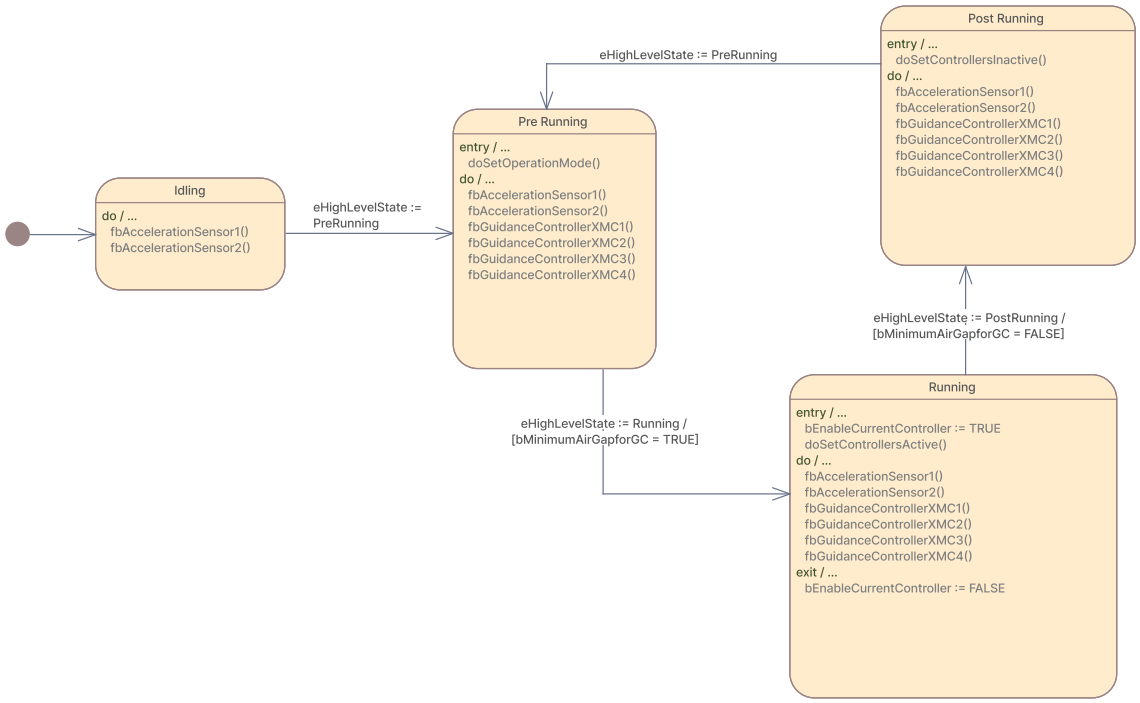

Figure 4.20: State Diagram of the High Level system LevitationSystem.

4.4 Sub Level Architecture

At Sub Level, there are no more state machines, since the systems have a concrete functionality that is called up when needed. For this reason, UML sequence diagrams are used to model this layer.

Sequence diagrams primarily model the communicative flow between system participants. Since Sub Level diagrams sit at the end of the communication chain and both execute control commands and deliver sensor data, this results in a complicated communication structure. For this reason, two subsystems that exhibit a particularly high level of complexity are modeled by a sequence diagram in the following.

4.4.1 Trajectory Generator

The Sub Level system TrajectoryGenerator is responsible for generating a trajectory that is traversed during the transition process. The function block contains a controller structure that operates either as a P-controller or as an I-controller. The operation mode is set before the calculation starts.

The input values of the trajectory generator are the actual values of levitation distance, velocity, acceleration, and current. These are available to the Main Controller from sensor data of the XMC boards. Further input values are the desired final values of levitation distance, velocity, and acceleration after the transition process. A desired transition time is also specified. This data is provided by Operation Control at the beginning of each transition process.

The Trajectory Generator calculates a trajectory based on this input data. At each point in the current transition, it outputs a packet of new reference values. This includes values for levitation distance, velocity, acceleration, current, voltage, and force. These data are passed to the current controllers implemented on the XMC boards.Thus, they are cyclically supplied

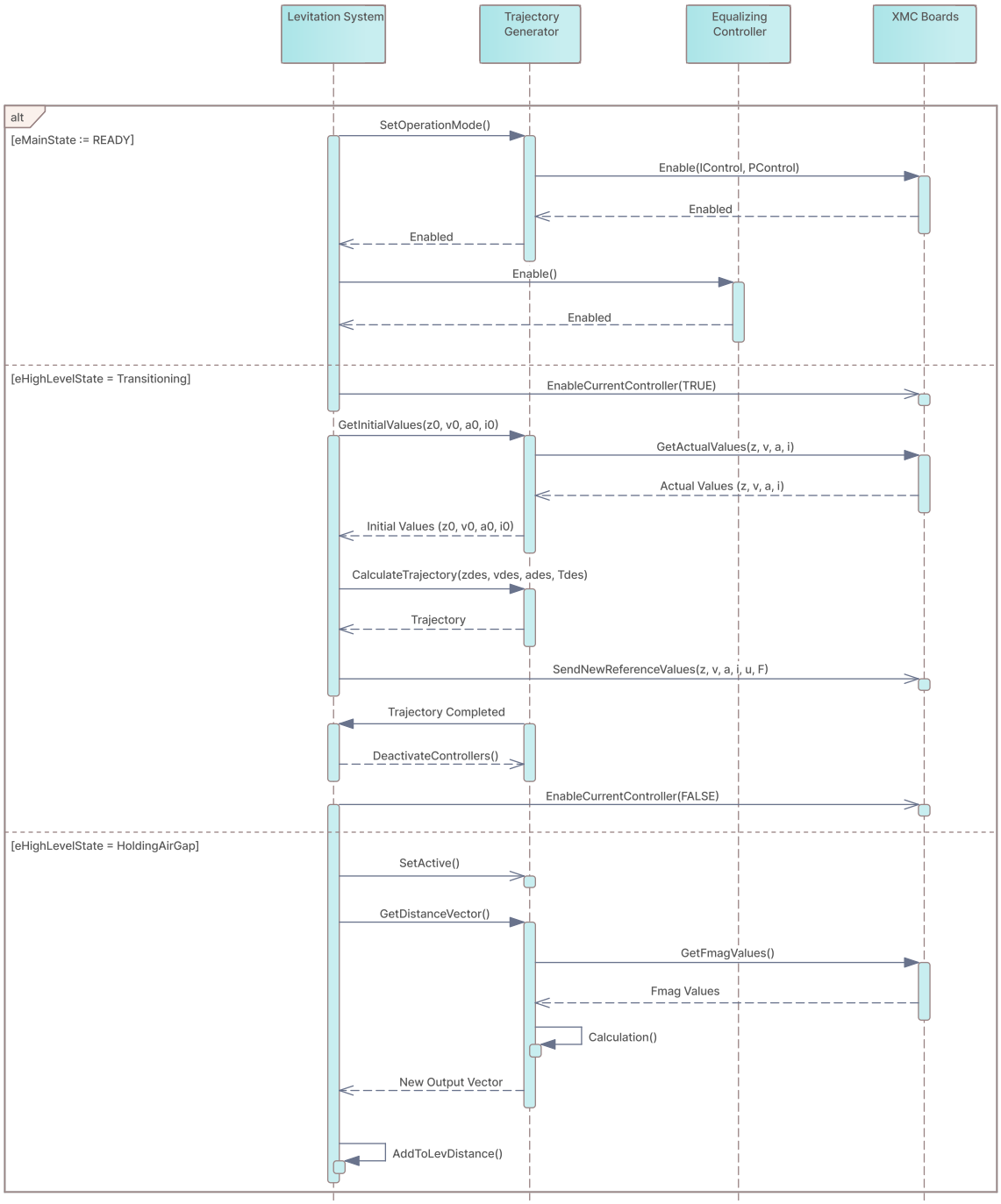

Figure 4.21: Sequential diagram of the communication behavior of the TrajectoryGenerator.

with new reference values. Figure fig. 4.21 shows the sequential flow of this process.

4.4.2 Guidance Controller

The guidance controller is responsible for lateral distance control in the y-direction. It ensures that the pod does not tilt laterally.

The guidance controller is a control structure that can be operated as a P-controller or I-controller. The operation mode is set by an appropriate signal through Operation Control and passed on to the current controllers on the XMC boards.

The controller's input data is current sensor data from the XMC boards on distance, acceleration, and current. In addition, the deviation from the centerline desired by Operation Control is passed on.

Outputs are new reference values for current, voltage, and force. These are cyclically passed on to the current controllers implemented on the XMC boards. These cause a change in the lateral magnetic field by controlling the coil current, which corrects the position of the pod. Figure fig. 4.22 shows the sequence diagram of the Sub Level system GuidanceController.

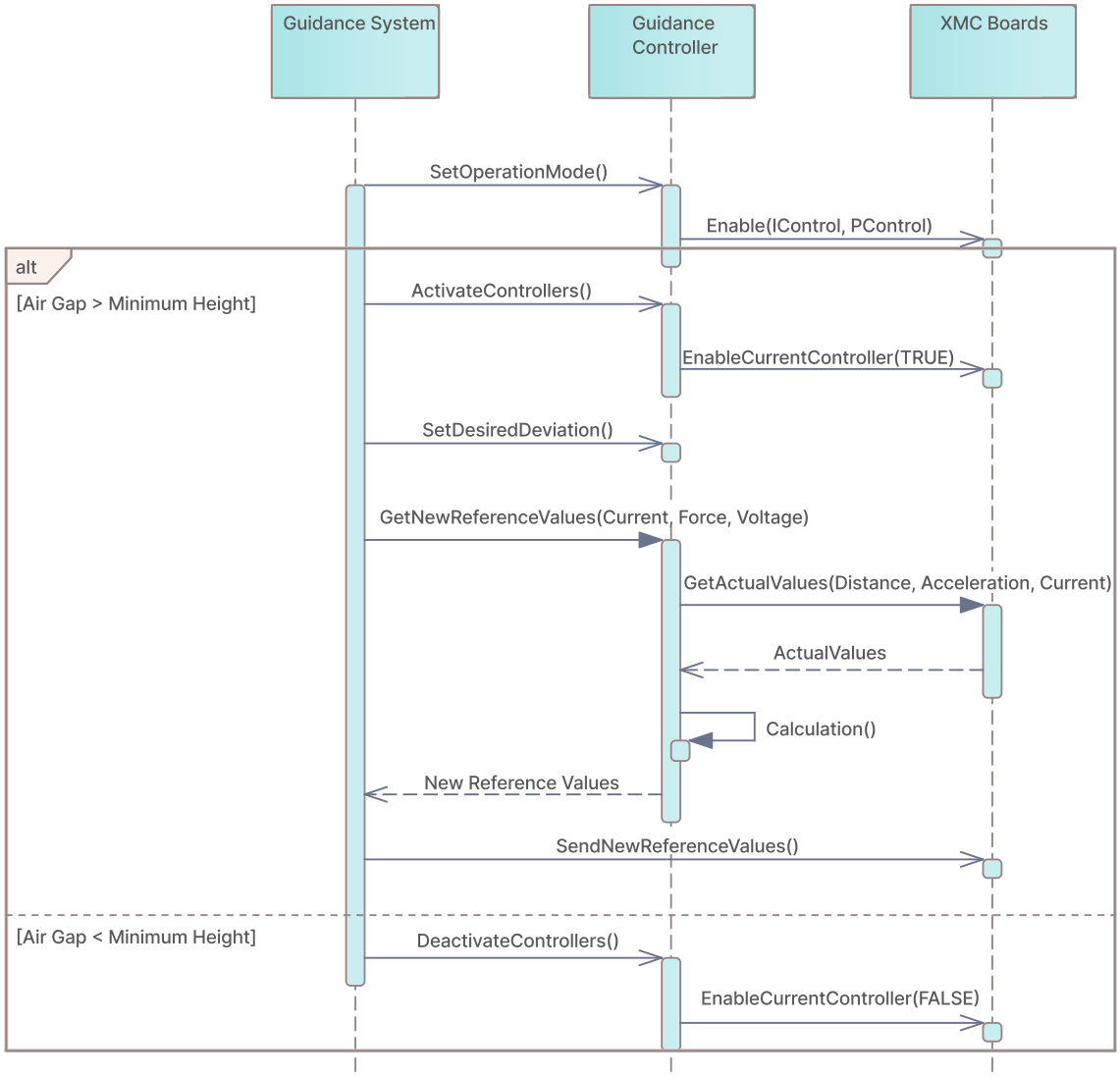

Figure 4.22: Sequential diagram of the communication behavior of the GuidanceController.

Chapter 5

Testing and Validation

This chapter deals with the validation of the implemented software architecture for the Main Controller. The implementation strictly follows the modeling described in this thesis. In this chapter, we present the product requirements for the Main Controller that have been defined in advance of this thesis. For each requirement, we develop a test scenario. A test scenario consists of several program variables that are set to specific values according to a defined sequence. Then, the behavior of the Main Controller is tested against the corresponding product requirement. The second part of this chapter is dedicated to a code analysis through various metrics. The program will be evaluated based on its textual and structural complexity. In addition, the test cases will be ranked based on their code coverage.

5.1 Requirement Analysis

In the requirements analysis, the product requirements on which this work is based are contrasted with test scenarios. This is to check whether the required functionality can be met. Product requirements contain technical information and details about the overall system. They were defined in advance of this work and served as reference points for the system architecture we developed.

5.1.1 Product Requirements

PR1: State Machine

The Main Controller implements a state machine that controls the system behavior based on the internal state of the Main Controller and the external control commands from the Operation Control Station. Operation Control has signals that start and stop the operation mode, provoke a transition to the error state as well as cause it to be exited. In addition, Operation Control can individually enable or disable all High Level systems.

PR2: Service Module

The Main Controller has a subsystem that is responsible for the components of the Service Module. In addition, this system is to control LEDs that indicate the status of the IPC box and the Main Controller. The LEDs shall be adjustable in several colors and can be switched on and off. In addition, a single-color LED shall have multiple blinking modes that indicate the status of the IPC box.

PR3: Trajectory Generator

The Main Controller integrates the controllers developed in Simulink for trajectory generation and equalizing control as a black box. The controllers are supplied with the inputs they require and started at the required point by an internally set start signal. The required inputs are provided cyclically. The required inputs for the levitation control are sensor data on levitation height, velocity, acceleration, and current. These are to be polled cyclically from the XMC boards. The calculated reference values are to be made available cyclically to the other participants in the control system.

PR4: Acceleration Sensor

The Main Controller reads the data from an IMU acceleration unit from the company Advanced Navigation. The model communicates via RS232 with an EtherCAT clamp connected to the Main Controller. The incoming data stream is to be processed in such a way that the latency in the processing of the RS232 data packets is minimized. Measurements are to be carried out to determine an optimum transmission frequency.

5.1.2 Derivation of Test Cases

TC1: State Machine

Test Case 1: State Machine			
Number	**Variable**	**Setpoint Value**	**Desired Behavior**
1	*OM_origO_StartOperational*	FALSE –> TRUE	system changes to *OPERATIONAL* state
2	*OM_origO_StopOperational*	FALSE –> TRUE	system changes to *SHUTDOWN* state
3	*OM_origO_EndShutdown*	FALSE –> TRUE	system changes back to *READY* state
4	*OM_origO_GoToError*	FALSE –> TRUE	system is forced to enter *ERROR state*
5	*OM_origO_AckError*	FALSE –> TRUE	system leaves *ERROR* state and resets error IDs
6	*OM_origG_ServiceModuleDisable*	FALSE –> TRUE	condition of ServiceModule is set to *SLEEP*
7	*OM_origG_LevitationDisable*	FALSE –> TRUE	condition of LevitationSystem is set to *SLEEP*
8	*OM_origG_GuidanceDisable*	FALSE –> TRUE	condition of GuidanceSystem is set to *SLEEP*

Table 5.1: List of all variables tested in TC1.

For the execution of this test scenario, the listed variables are manually set to the desired value. This is done in the order specified by the table 5.1. The results are checked against the log report. For the execution of this test case, the error handling is put into test mode so that the startup check can be passed.

TC2: Service Module

Test Case 2: Service Module			
Number	**Variable**	**Setpoint Value**	**Desired Behavior**
1	*fbBoxLED.eMode*	*LEDMode.On*	turn on Box LED
2	*fbBoxLED.eMode*	*LEDMode.SlowFlashing*	turn on flashing with small frequency
3	*fbBoxLED.eMode*	*LEDMode.FastFlashing*	turn on flashing with high frequency
4	*fbBoxLED.eMode*	*LEDMode.Off*	turn off Box LED
5	*fbRGBLED.eColor*	*LEDColor.Red*	turn on red LED
6	*fbRGBLED.eColor*	*LEDColor.Green*	turn on green LED
7	*fbRGBLED.eColor*	*LEDColor.Blue*	turn on blue LED
8	*fbRGBLED.eColor*	*LEDColor.Off*	turn off RGB LEDs

Table 5.2: List of all variables tested in TC2.

For this test scenario, the High Level System Service Module is separated from the overall system and only the behavior of the LED function blocks is considered. Their inputs are set

manually and the behavior is recorded by logging. The manual setting of the inputs simulates the calling of the function blocks when running through the state machine in normal operating mode.

TC3: Trajectory Generator

Test Case 3: Trajectory Generator			
Number	**Variable**	**Setpoint Value**	**Desired Behavior**
1	*OM_origG_LevControlOption*	1	enable I-Control mode
2	*OM_origG_DesiredLevDistance*	0.01	transition to nominal air gap height
3	*OM_origG_DesiredLevVelocity*	0	end transition process with no velocity in z-direction
4	*OM_origG_DesiredLevAcceleration*	0	end transition process with no acceleration in z-direction
5	*OM_origG_DesiredLevTime*	20000	complete trajectory within 20 seconds
6	*EM_origE_CurrentLev*	0	condition for transition process from ground level
7	*EM_origE_DistanceLevFiltered*	0.25	condition for transition process from ground level
8	*EM_origE_VelocityLevFiltered*	0	condition for transition process from ground level
9	*EM_origE_IMUZ*	0	condition for transition process from ground level

Table 5.3: List of all variables tested in TC3.

The trajectory generator is implemented as a black box on the Main Controller. This means that its internal logic has been developed externally. It is controlled with the commands listed in the table. These are set in normal mode by Operation Control. Data on current sensor values is written from the XMC boards to the variables in the lower part of the table.

TC4: Acceleration Sensor

Test Case 3: Trajectory Generator			
Number	**Variable**	**Setpoint Value**	**Desired Behavior**
1	*bStartPacketCount*	FALSE –> TRUE	start counting the incoming RS232 packets
2	*fPacketRate*	to be checked	determines the average packet rate out of 10 measurements

Table 5.4: List of all variables tested in TC4.

For the optimization of the serial interface for RS232 communication, tests are carried out with different cycle times and baud rates. The *FB_AccelerationSensor* function block is considered separately from the overall system. After the start of the method for the determination of the packet rate, it determines the average packet rate from ten measurements, which are carried out in the time span of 20 seconds. The measurements are repeated several times and summarized in a table at the end.

5.1.3 Evaluation

TC1: State Machine

The comparison with the log report shows that the Main Controller has the desired behavior. The commands defined in table 5.1 are implemented and the desired behavior is displayed. The log report also records several errors and warnings that occur during the testing process. However, these do not lead to a change to the *ERROR* state, since a special test mode is activated.

Severity Level	Event Text	Time Raised
Info	Guidance System disabled	16.01.2023 23:52:39.384
Info	Levitation System disabled	16.01.2023 23:52:34.551
Info	Service Module disabled	16.01.2023 23:52:27.474
Error	Error occured in READY state. Error Message: EtherCAT Master Missing Ethernet Connection	16.01.2023 23:51:54.499
Error	Error occured in READY state. Error Message: Connection Timeout to XMC4	16.01.2023 23:51:54.499
Error	Error occured in READY state. Error Message: Connection Timeout to XMC3	16.01.2023 23:51:54.499
Error	Error occured in READY state. Error Message: Connection Timeout to XMC2	16.01.2023 23:51:54.499
Error	Error occured in READY state. Error Message: Connection Timeout to XMC1	16.01.2023 23:51:54.499
Error	Error occured in READY state. Error Message: Connection Timeout to Operation Control	16.01.2023 23:51:54.499
Info	MainController STM changed to READY state	16.01.2023 23:51:49.499
Warning	Warning Message: DistanceSensor out of Range	16.01.2023 23:51:49.498
Info	MainController STM changed to STARTUP state	16.01.2023 23:51:49.498
Info	All Errors acknowledged and cleared by Operation Control	16.01.2023 23:51:49.498
Info	MainController STM changed to ERROR state	16.01.2023 23:51:40.131
Info	MainController STM changed to READY state	16.01.2023 23:51:28.835
Info	MainController STM changed to SHUTDOWN state	16.01.2023 23:51:20.173
Info	MainController STM changed to OPERATIONAL state	16.01.2023 23:51:11.194
Error	Error occured in READY state. Error Message: EtherCAT Master Missing Ethernet Connection	16.01.2023 23:51:06.204
Error	Error occured in READY state. Error Message: Connection Timeout to XMC4	16.01.2023 23:51:06.204
Error	Error occured in READY state. Error Message: Connection Timeout to XMC3	16.01.2023 23:51:06.204
Error	Error occured in READY state. Error Message: Connection Timeout to XMC2	16.01.2023 23:51:06.204
Error	Error occured in READY state. Error Message: Connection Timeout to XMC1	16.01.2023 23:51:06.204
Error	Error occured in READY state. Error Message: Connection Timeout to Operation Control	16.01.2023 23:51:06.204
Warning	Warning Message: DistanceSensor out of Range	16.01.2023 23:51:01.204
Info	MainController STM changed to READY state	16.01.2023 23:51:01.204

Figure 5.1: Log report of test case scenario 1.

TC2: Service Module

The comparison with the log report shows no abnormalities. The LEDs show the desired behavior in response to the control commands of *FB_ServiceModule*. Furthermore, the LEDs do not indicate any error messages regarding increased temperature or voltage supply and are thus operational for normal operation.

Severity Level	Event Text	Time Raised
Info	RGB LED switched to Off	17.01.2023 00:56:36.420
Info	RGB LED switched to Blue	17.01.2023 00:56:29.781
Info	RGB LED switched to Green	17.01.2023 00:56:21.042
Info	RGB LED switched to Red	17.01.2023 00:56:07.110
Info	Box LED switched to Off Mode	17.01.2023 00:55:53.558
Info	Box LED switched to FastFlashing Mode	17.01.2023 00:55:42.104
Info	Box LED switched to SlowFlashing Mode	17.01.2023 00:55:31.471
Info	Box LED switched to On Mode	17.01.2023 00:55:19.361

Figure 5.2: Log report of test case scenario 2.

TC3: Trajectory Generator

The evaluation of this test case is not yet fully possible at the time of this work, since a fully functional version of the trajectory generator is not yet available. The controller is still under development. However, testing of the data handling structure on the Main Controller shows success. Inputs and outputs are mapped correctly and are available to the hierarchical interfaces.

When executing the test scenario with a fully functional controller structure, the variables in table 5.5 must be observed for their correct behavior.

Test Case 3: Trajectory Generator			
Number	**Variable**	**Setpoint Value**	**Desired Behavior**
1	*ME_origM_NewReferenceLevDistance*	0.01	reference value at the end of the transition process
2	*ME_origM_NewReferenceLevVelocity*	0	reference value at the end of the transition process
3	*ME_origM_NewReferenceLevAcceleration*	0	reference value at the end of the transition process
4	*ME_origM_NewReferenceLevForce*	0	reference value at the end of the transition process
5	*ME_origM_NewReferenceLevCurrent*	to be determined	reference value at the end of the transition process
6	*ME_origM_NewReferenceLevVoltage*	to be determined	reference value at the end of the transition process
7	*ME_origM_LevEnableCurrentController*	FALSE –> TRUE	enables the current controller on the XMC-board
8	*ME_origM_LevEnableIControl*	FALSE –> TRUE	enables the I-Control mode on the XMC-board
9	*ME_origM_LevEnablePControl*	FALSE	disabled since mode is set to I-Control

Table 5.5: List of all variables to be monitored in TC3.

TC4: Acceleration Sensor

Table 5.6 shows the average number of lost data packets in the communication with the serial RS232 communication interface. During the tests, the average packet rate was determined from 10 measurements within 20 seconds. The measurements were made at different base times and transmission frequencies. The most stable packet rates can be assigned to the two lowest frequencies. A frequency of 250 Hz still promises a somewhat lower latency, which is why this is recommended. A pattern emerges for the base times. The shorter the cycle time, the more packets are lost. For measurements with the smallest cycle time, all packets are lost because the corresponding EtherCAT frames are missed.

Therefore a base time of 1 ms and a transmission frequency of 250 Hz is recommended.

Base Time in ms	**1000**	**500**	**333**	**250**
Frequency in Hz	**average number of lost packets**			
1000	809.9	815.5	875.5	1000.0
500	252.7	250.1	329.2	500.00
333.33	6.400	3.600	123.8	333.30
250	1.000	1.000	54.10	250.00
200	0.800	0.700	57.40	200.00

Table 5.6: Average number of lost packets related to frequency and base time.

5.2 Code Examination

In this section, measures of the quality of the implemented software architecture are to be determined. This creates formal comparison and evaluation possibilities. The code of the overall system is considered as well as the code sections that are used in the test scenarios. This should help to validate the assessment of the quality of our proposed software architecture.

5.2.1 Code Coverage

Code coverage is a simple metric for determining the meaningfulness of test cases. It indicates what percentage of the total code is covered by a test scenario. In the following, we consider test case 1, as described in Table 5.1. For this, we determine the number of processed lines

of code and divide it by the total number of source lines of code. Comments and blank lines are excluded.

The calculation is as follows:

$$CC = \frac{\text{processed lines of code}}{\text{source lines of code}}$$

The value for testing the Top Level state machine in *PRG_MainController* is 80.5 %. This means that the code coverage for this test case is within the target interval of 70 % - 80%. The behavior in the test case can thus be validated for normal operation.

5.2.2 Halstead's Metrics

In this section, we want to use metrics to check the structural and textual complexity of the Top Level state machines implemented in *PRG_MainController* and *PRG_ErrorHandling*. For this purpose, the entire program code is examined and a value is determined that qualitatively assesses the implementation.

Halstead's metrics are based on a combination of information theory, logical reasoning, and psychology. They measure the textual complexity of a program by dividing the program into operators and operands. Operators are active elements that denote actions. Operands are passive elements that describe data values. These two quantities are used to determine various measures that allow statements to be made about the quality of the program. The following formulas are used:

Program vocabulary:	$n = n_1 + n_2$
Program length:	$N = N_1 + N_2$
Halstead´s program length:	$H = n_1 * \log_2 n_1 + n_2 * \log_2 n_2$
Average usage:	$P = \frac{N_2}{n_2}$
Failure prediction:	$B = \frac{(N_1 + N_2)\log_2(n_1 + n_2)}{3000}$
Difficulty:	$D = \frac{n_1 * N_2}{2 * n_2}$

where:

n_1 = number of distinct operators
n_2 = number of distinct operands
N_1 = total number of operators
N_2 = total number of operands

The results in Table 5.7 show that the two Top Level program modules are of different complexity. *PRG_ErrorHandling* has a larger vocabulary, but the implementation length is smaller. In addition, *PRG_ErrorHandling* has a greater program length according to Halstead. The average usage and error probability values are similar for both program modules. The overall difficulty turns out to be lower for *PRG_ErrorHandling*. This is mainly because *PRG_MainController* has implemented more interfaces to other function blocks.

Overall, the values do not show any critical abnormalities. However, it can be seen that there is still a lot of potentials to reduce the programs' complexity.

	PRG_MainController	*PRG_ErrorHandling*
n	35	53
N	301	185
H	149.32	264.53
P	3.68	2.11
B	0.51	0.35
D	18.4	11.65

Table 5.7: Halstead´s Metrics for *PRG_MainController* and *PRG_ErrorHandling*.

Chapter 6

Conclusion

This chapter sums up the most important findings of this work. In addition, we will point out difficulties that arose during the elaboration and explore the limits of this work. In the second part of this chapter, we give a short outlook on future developments. We present potential improvements and discuss the changes that could benefit our proposed architecture.

6.1 Conclusion

As part of this work, a software architecture for a component of the demonstrator's control system was developed. The development was done by comparing it with control systems of other ultra high-speed ground transportation systems. In addition, this work draws on cross-industry standards for system modeling software development.

The main task of the software architecture developed in this thesis is to add a decentralized interface to the software infrastructure of the Demonstrator. Computationally intensive tasks of the control components distributed decentrally at the Service Module are to be outsourced to the Main Controller. For this purpose, we put great emphasis on the development of a secure and reliable data-handling structure.

A common thread running through this work is the strict division of the system into hierarchical levels. This structure is important to achieve the reusability of the individual software parts. The interfaces between these components are also clearly defined by an object-oriented approach, allowing independent development at the subsystem level. This shortens the development time and reduces the error rate.

Finally, the bulk of this work deals with the modeling of the control system through UML diagrams. Great care was taken to ensure that all aspects of the implementation are shown to advantage by using different diagram types for different system levels. This facilitates the understanding of the implemented software and leaves room for future developments. This is very important for such a rapidly evolving research program as TUM Hyperloop.

During the development of this work, it was noted that the use of object-oriented programming and its tools for PLC control programming is also beneficial in the context of this project. This has been particularly evident in the development of a clear data structure. Due to the partly large inheritance depth of the program, we were faced with the challenge of how inputs and outputs reach their destination as discretely as possible. On the one hand, no duplicates of variables should appear, which would harm the clarity, on the other hand, values should also be protected from unauthorized access by incorrectly implemented functions. For this reason, we have decided to implement the entire data handling centrally in one place. Control instructions are always passed on hierarchically from top to bottom. Values to be output are written directly to the global variables from publicly accessible interfaces of

the function blocks.

Furthermore, the use of the object-oriented tools properties, interfaces, and abstract classes brought two important advantages for it. First, all the systems used are standardized. Thereby they can be easily extended by further functionality. Also, the addition of new function blocks is easily possible. Secondly, all important system variables are linked with the given properties of the subsystems. Thus, changes can be made to the internal behavior of the function blocks without losing or overwriting important links for the overall system behavior.

This is particularly important for the three function blocks *FB_TrajectoryGenerator*, *FB_EqualizingController*, and *FB_GuidanceController*. At the time of this work, these are only implemented as dummy versions. This means that they do not yet have any functional internal logic, since it is still under external development. Nevertheless, the function blocks are already fully integrated into the overall system and only need to be supplemented internally by the controller structures when they are completed.

Finally, however, it must also be said that precisely because of these missing controller structures, testing of the elaborated software architecture has not yet been completed. The aforementioned function blocks in particular contain the most sensitive software structures, which must be tested in detail before a release for operation on the pod can be issued. Thus, the evaluation of this work is still very general. Moreover, the High Level state machines also require even more intensive testing to achieve higher code coverage. Nevertheless, the sequence tests of the Top Level state machine and the error handling task already show satisfactory results, which provide a positive conclusion of this work.

6.2 Outlook

With the current state of development, this work is completed. Nevertheless, the system is still far away from being completely developed. For example the above-mentioned control structures for levitation and guidance are missing. Apart from that, there are some more open questions, which have not yet been considered in this work.

This concerns for example the general scheme of the Top Level state machine. In our elaboration, we have chosen an approach with five states. However, it can be doubted whether the *SHUTDOWN* state is mandatory. The corresponding High Level states all do not show much difference from the preceding or following state. Furthermore, this would mean alignment with the state machine implemented on the XMC boards. This would greatly simplify debugging and control. However, for such an alignment it must be waited what the exact functionality of the trajectory generator is, respectively which control processes are necessary for the landing process. Furthermore, it has to be discussed whether this change will influence the behavior in case of an error.

Another aspect is the naming convention, which is used in this work. We have followed the naming convention suggested by Beckhoff [6] since we have also programmed in a Beckhoff environment. However, it may be debated whether a nomenclature with data type prefixes is still up-to-date. Current developments rather use the practice of generating variable names without the strict use of prefixes and suffixes. The aim here is to increase the comprehensibility of the code at hand.

A further possibility for improvement exists certainly in the stronger use of global variables. These are only considered in the context of this work to link EtherCAT inputs and outputs to the system. Theoretically, one could take advantage of this by having High Level systems directly pick up their respective control commands from there. This would slim down the very extensive data handling method and reduce variable duplicates. However,

this would be at the expense of clarity. It thus remains to be weighed up whether such a measure would be useful.

In conclusion, this is only a first draft of a software architecture for the Main Controller. In the course of the development progress of the Demonstrator project, iterative change processes will be necessary so that the required behavior is established. Nevertheless, with the completion of this work, a large step has already been taken towards operational capability and certainly also a small step towards the mobility of the future.

Appendix A

Appendix 1

ID	System	Message
		Warnings
0	SM	DistanceSensor Communication Error
1	SM	DistanceSensor out of Range
2	SM	BoxLED Temperature too high
3	SM	BoxLED No Supply Voltage
4	SM	RedLED Temperature too high
5	SM	RedLED No Supply Voltage
6	SM	GreenLED Temperature too high
7	SM	GreenLED No Supply Voltage
8	SM	BlueLED Temperature too high
9	SM	BlueLED No Supply Voltage
10	LS	EqualizingController Invalid Paramaters
11	LS	EqualizingController Invalid CycleTime
12	LS	TrajectoryGeneratorXMC1 AirGap OutOfRange
13	LS	TrajectoryGeneratorXMC1 Velocity OutOfRange
14	LS	TrajectoryGeneratorXMC1 Acceleration OutOfRange
15	LS	TrajectoryGeneratorXMC1 Current OutOfRange
16	LS	TrajectoryGeneratorXMC1 Voltage OutOfRange
17	LS	TrajectoryGeneratorXMC1 Force OutOfRange
18	LS	TrajectoryGeneratorXMC2 AirGap OutOfRange
19	LS	TrajectoryGeneratorXMC2 Velocity OutOfRange
20	LS	TrajectoryGeneratorXMC2 Acceleration OutOfRange
21	LS	TrajectoryGeneratorXMC2 Current OutOfRange
22	LS	TrajectoryGeneratorXMC2 Voltage OutOfRange
23	LS	TrajectoryGeneratorXMC2 Force OutOfRange
24	LS	TrajectoryGeneratorXMC3 AirGap OutOfRange
25	LS	TrajectoryGeneratorXMC3 Velocity OutOfRange
26	LS	TrajectoryGeneratorXMC3 Acceleration OutOfRange
27	LS	TrajectoryGeneratorXMC3 Current OutOfRange
28	LS	TrajectoryGeneratorXMC3 Voltage OutOfRange
29	LS	TrajectoryGeneratorXMC3 Force OutOfRange
30	LS	TrajectoryGeneratorXMC4 AirGap OutOfRange
31	LS	TrajectoryGeneratorXMC4 Velocity OutOfRange
32	LS	TrajectoryGeneratorXMC4 Acceleration OutOfRange
33	LS	TrajectoryGeneratorXMC4 Current OutOfRange

ID	System	Message
34	LS	TrajectoryGeneratorXMC4 Voltage OutOfRange
35	LS	TrajectoryGeneratorXMC4 Force OutOfRange
36	GS	IMU1 Communication Error
37	GS	IMU1 OutOfRange
38	GS	IMU2 Communication Error
39	GS	IMU2 OutOfRange
40	GS	GuidanceControllerXMC1 Current OutOfRange
41	GS	GuidanceControllerXMC1 Voltage OutOfRange
42	GS	GuidanceControllerXMC1 Force OutOfRange
43	GS	GuidanceControllerXMC2 Current OutOfRange
44	GS	GuidanceControllerXMC2 Voltage OutOfRange
45	GS	GuidanceControllerXMC2 Force OutOfRange
46	GS	GuidanceControllerXMC3 Current OutOfRange
47	GS	GuidanceControllerXMC3 Voltage OutOfRange
48	GS	GuidanceControllerXMC3 Force OutOfRange
49	GS	GuidanceControllerXMC4 Current OutOfRange
50	GS	GuidanceControllerXMC4 Voltage OutOfRange
51	GS	GuidanceControllerXMC4 Force OutOfRange
Errors		
52	MC	Connection Timeout to Operation Control
53	MC	Connection Timeout to XMC1
54	MC	Connection Timeout to XMC2
55	MC	Connection Timeout to XMC3
56	MC	Connection Timeout to XMC4
57	MC	EtherCAT Master Missing Ethernet Connection
Info		
58	MC	All Errors Acknowledged and Cleared by Operation Control
59	MC	Service Module disabled
60	MC	Levitation System disabled
61	MC	Guidance System disabled

Table A.1: Table of all Message IDs occurring in event logging .

List of Figures

List of Tables

Bibliography

[1] Arthur, D. K. “Preliminary investigation of supersonic diffusers”. In: *National advisory committee for aeronautics* (May 1945).

[2] AUTOSAR. *Layered Software Architecture*. Dec. 2017. URL: https://www.autosar.org/fileadmin/standards/classic/22-11/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf.

[3] AUTOSAR. *Specification of ECU StateManager with fixed statemachine*. Dec. 2017. URL: https://www.autosar.org/fileadmin/standards/classic/3-2/AUTOSAR_SWS_ECU_StateManager.pdf.

[4] Beckhoff Automation. *TwinCAT 3 eXtended Automation Engineering*. Apr. 2013. URL: https://www.beckhoff.com/de-de/support/downloadfinder/software-und-tools/ (visited on 12/14/2022).

[5] Beckhoff Automation. *EK110x-00xx, EK15xx*. 4.1. Beckhoff. Dec. 2021. URL: https://download.beckhoff.com/download/Document/io/ethercat-terminals/ek110x_ek15xxen.pdf (visited on 12/14/2022).

[6] Beckhoff Automation. *Naming Conventions*. May 2021. URL: https://infosys.beckhoff.com/english.php?content=../content/1033/te1200_tc3_plcstaticanalysis/3471943051.html&id= (visited on 01/17/2023).

[7] Beckhoff Automation. *CX20x0*. 2.5. Beckhoff. July 2022. URL: https://download.beckhoff.com/download/document/ipc/embedded-pc/embedded-pc-cx/cx2000_en.pdf (visited on 12/14/2022).

[8] C/LM - LAN/MAN Standards Committee. *IEEE Standard for Ethernet*. May 2018.

[9] Chris Rupp, Stefan Queins, die SOPHISTen. *UML2 glasklar*. 4th ed. Carl Hanser Verlag, 2012.

[10] Eric Chaidez Shankar P. Bhattacharyya, A. N. K. “Levitation Methods for Use in the Hyperloop High-Speed Transportation System”. In: *Energies 12.21* (Nov. 2019).

[11] European Environment Agency. *Greenhouse Gas Emissions from Transport in Europa*. Oct. 2022. URL: https://www.eea.europa.eu/ims/greenhouse-gas-emissions-from-transport (visited on 12/09/2022).

[12] Eveline Gottzein, Reinhold Meisinger, and Luitpold Miller. “The “MagneticWheel” in the Suspension of High-speed Ground Transportation Vehicle”. In: *IEEE Transactions on vehicular technology* VT-29.1 (1980).

[13] Infineon Technologies. *XMC4700 / XMC4800 Data Sheet*. URL: https://www.infineon.com/dgdl/Infineon-XMC4700-XMC4800-D-v01_01-EN.pdf?fileId=5546d462518ffd850151908ea8db00b3 (visited on 12/15/2022).

[14] International Electrotechnical Commission. *IEC 61131-3*. Feb. 2013.

[15] International Energy Agency. *Energy Technology Perspectives*. Tech. rep. International Energy Agency, 2020.

[16] ISO/IEC JTC 1 Information technology. *ISO/IEC 19505-2:2012*. Apr. 2012.

[17] Martin, R. C. *Clean Architecture*. mitp, Feb. 2018. ISBN: 978-3958457249.

[18] Musk, E. *Hyperloop Alpha*. Research rep. Tech. rep. SpaceX, Aug. 2013. URL: https://www.spacex.com/sites/spacex/files/hyperloop_alpha.pdf.

[19] Ping Wang, Zhiqiang Long, Chunhui Dai. *A PHM architecture of maglev train based on thedistributed hierarchical structure*. June 2019. URL: https://ieeexplore.ieee.org/abstract/document/8781478 (visited on 12/16/2022).

[20] Radeck, D. *Electronics Box*. Sept. 2022. URL: https://wiki.next-prototypes.de/display/next/BE+Electronics+Box#BEElectronicsBox-XMCBoard (visited on 12/12/2022).

[21] Radeck, D. *IPC Box*. Sept. 2022. URL: https://wiki.next-prototypes.de/display/next/BH+IPC+Box (visited on 12/12/2022).

[22] Radeck, D. *Operation Control Station*. Sept. 2022. URL: https://wiki.next-prototypes.de/display/next/D+Operation+Control+Station (visited on 12/12/2022).

[23] Radeck, D. *Passenger Module*. Oct. 2022. URL: https://wiki.next-prototypes.de/display/next/C+Passenger+Module (visited on 12/12/2022).

[24] Radeck, D. *Service Module*. Sept. 2022. URL: https://wiki.next-prototypes.de/display/next/B+Service+Module (visited on 12/12/2022).

[25] Radeck, D. *Tube*. Sept. 2022. URL: https://wiki.next-prototypes.de/display/next/A+Tube (visited on 12/12/2022).

[26] Rainer Schach Peter Jehle, R. N. *Transrapid und Rad-SchieneHochgeschwindigkeitsbahn - Ein gesamtheitlicher Systemvergleich*. Springer Verlag, 2006.

[27] Schünemann, F. "The Operation Control System of the Transrapid". In: *ZEVrail* 35 (Apr. 2004).

[28] Strittmatter, K. "Und sie schämen sich doch". In: *Süddeutsche Zeitung* (Oct. 2019). URL: https://www.sueddeutsche.de/reise/flugscham-schweden-deutschland-1.4649460.

[29] Umweltbundesamt. *Flugreisen*. Apr. 2022. URL: https://www.umweltbundesamt.de/umwelttipps-fuer-den-alltag/mobilitaet/flugreisen#hintergrund (visited on 12/09/2022).

[30] Werner, B. "Object-oriented extensions for IEC 61131-3". In: *Industrial Electronics Magazine, IEEE* 3 (Jan. 2010), pp. 36–39. DOI: 10.1109/MIE.2009.934795.

[31] Wiegel, M. "Frankreich verbietet kurze Inlandsflüge". In: *Frankfurter Allgemeine* (May 2021). URL: https://www.faz.net/aktuell/politik/ausland/frankreich-verbietet-kurze-inlandsfluege-17326781.html.

Disclaimer

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Garching, February 02, 2023

(Signature)