# Master Controller Documentation

By: Jacob Offersen

The following document is the product of an elective Formula Student course done in the spring of 2020 (F20).
The Master Controller was developed for the Viking X.

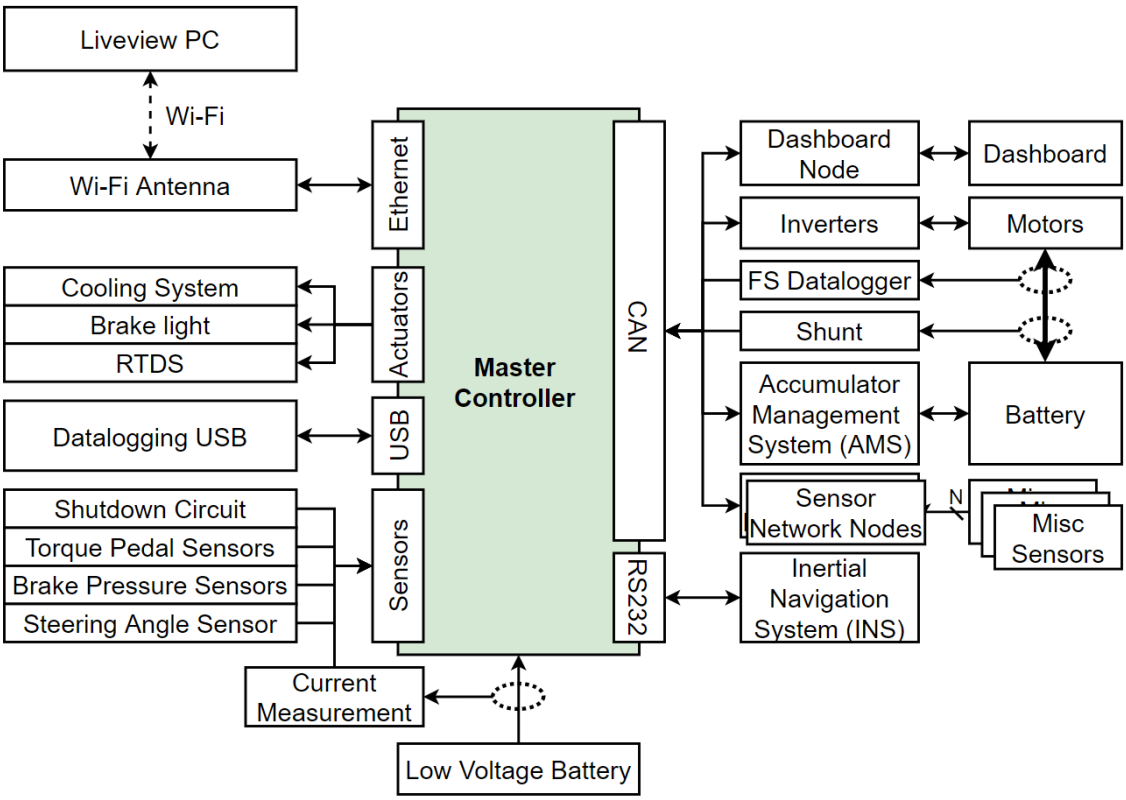


Figure 1: A general overview of the system in which the Master Controller is working.


| Done by | CPR | Exam Nr | Signature |
|---|---|---|---|
| Jacob Offersen | 311294 | 84929345 | |


## Advisor

Karsten Holm Andersen, Associate Professor at SDU

Handed in on 31/05/2020

# Part I
# Introduction

The report contains descriptions of all features implemented on the central Master Controller for the Viking X car build by the Formula Student team SDU-Vikings. The car is build to participate in the Formula Student competitions for the summer of 2020.
The Master Controller has contact with all nodes in the car. The network is design around a star configuration where the Master Controller is the center as can be seen on figure 6.

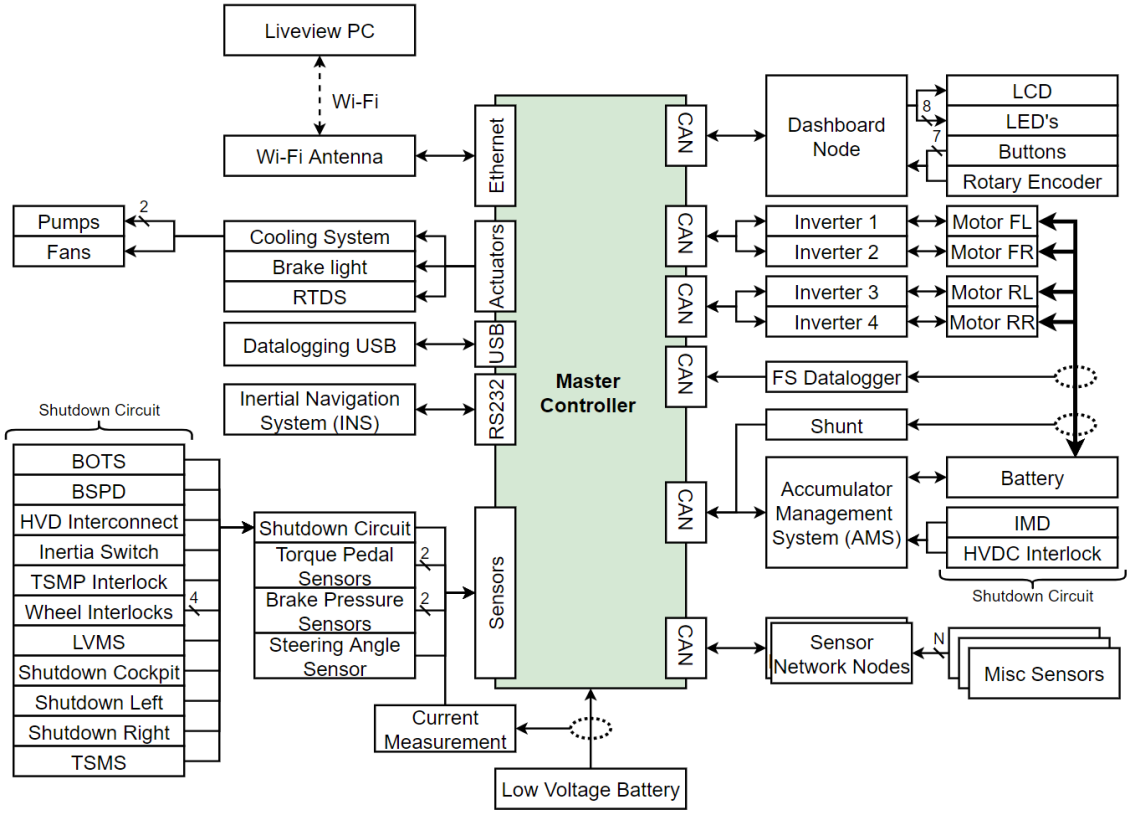


Figure 2: A general overview of the system in which the Master Controller is working.

The Master Controller works with some sensors directly through digital and analog I/O, it communicates with most other nodes in the car through CAN but also uses RS232, USB and Ethernet to communicate.
The information contained in this report is up-to-date at the time of delivery but will most likely be changed in various ways after delivery. For up-to-date information see the source code which can be found on the team's Gitlab repository.
`https://gitlab.com/sdu-vikings/X/master-controller`
Access can be granted to active team members by consulting the team management.

The plan was to have the car up and running and get the car software debugged during the spring of 2020 but due to the closing of the university, this has been

postponed. Therefore the software contains a lot of code only tested with the simulator which might cause unwanted behavior in the physical car. The development of certain features planned for the spring of 2020 has also been postponed of the same reason. These include the INS RS-232 interface, the Shunt CAN interface, the FS Datalogger CAN interface, the vehicle dynamics controller, and a vehicle performance monitoring environment for tuning the vehicle dynamics controllers.

# 1 Preface

This paper documents the master controller developed to control the Viking X car designed and manufactured by SDU-Vikings at the University of Southern Denmark(SDU). The documented code was development in the period January 2019 to June 2020.
The code for the cooling system (section 12) has been developed by Jonas Fuglsang Hansen. Some sections of the code in the car's state machine (section 8), generic CAN module (section 23.3) and other places have been made by Jonas Fuglsang Hansen and Kenn Fischer as well as general bug fixing. The code for the datalogger task discussed in section 22 has been made by the B&R Automation Denmark office and given to SDU-Vikings. The rest of the code development and liveview design has been done by me, Jacob Offersen.
The project is made in collaboration with the Formula Student team of the university, the SDU-Vikings. I want to thank the team members, the team directors and team advisors for their help and support. A special thanks my advisor and team advisor Karsten Holm Andersen, Assistant Professor at SDU, for his help, ideas and feedback.
I also want to thank &R Automation Denmark for the support.

# 2 Reading Guide

The documentation is written so it can be used to lookup features and it does not need to be read in chronological order. Important information needed to understand a feature completely will be referenced when necessary. Features or big coding

restructurings made in the period January 2020 to June 2020 has been labeled with: **(New feature)**

All coding examples have been simplified for the report because the actual versions contain long variable names which is not suitable for a report. For the actual code see the master controller Gitlab repository.
`https://gitlab.com/sdu-vikings/X/master-controller`

Alongside this documentation a *Automation Studio Starter Guide* has been handed in describing the important aspects of working with B&R products through the software IDE *Automation Studio*. For more inside into the B&R software framework consult that document which can be found in appendix J.

**Chapter I** ***Introduction***
Describes the purpose of the project and the problem statement.
**Chapter I** ***Requirements***
Explores the requirements set up for the master controller.
**Chapter III** ***PLC***
Introduction to why a PLC has been chosen and what features it has.
**Chapter IV** ***Software***
Walkthrough of all software components present in the system.
**Chapter V** ***Abbreviations***
List of the abbreviations used in the project.
**Chapter VII** ***Appendix***
**Appendix A** ***Rules***
A compiled list of all relevant rules.
**Appendix B** ***Alarms***
A compiled lisst of all alarms implemented.
**Appendix C** ***AMK Inverter State Machine***
The inverter state machine given by AMK.
**Appendix D** ***AMK Inverter Error Removal***
The inverter error removal procedure given by AMK.
**Appendix E** ***Shutdown Circuit Diagram***
An overview of the car's shutdown circuit.
**Appendix F - H** ***CAN Protocols***
The different CAN protocols used in the car.
**Appendix I** ***Liveview pages***
Pictures of the various visualization pages used in the car's liveview.
**Appendix J** ***Automation Studio Starter Guide***
A starter guide tailored to SDU-Vikings about the B&R programming IDE.

SDU

# Contents

# Part II
# Requirements

The requirements used for software development comes from the rulesets presented by the Formula Student competitions that the team plan on attending to. Most European competitions use the Formula Student Germany (FSG) ruleset but some competitions also have additional rules such as the Formula Student United Kingdom (FSUK) supplementary ruleset.
The rulesets used in this report are

- FSG 2020 ruleset [7]
- FSG 2020 handbook [8]
- FSUK 2020 supplementary ruleset [14]

The important rules for the master controller are referenced throughout the documentation when they are relevant. A full list of the rules referenced throughout the document can be found in appendix A.
Additionally the team has handed in a Failure Mode and Effects Analysis's (FMEA) to the FSUK competition in 2019 [10] and 2020 [11] as per rule *T11.9.5*, describing failure handling of the master controller in this report. For further information about failure mode analysis consult the FMEA documents. The documents are not included here but can be found in the teams data archives.

# Part III
# PLC

PLC's the automation industry are generally highly reliable because errors in this industry can be very expensive. To ensure that potential breakdowns will be as cheap as possible the PLC's themselves and their expansion boards are made modular to allow for each individual board or controller to be easily exchanged resulting in production lines only being offline potentially for a few minutes.
These trades are encouraged for use in a race car. The controller should be robust and reliable because a controller break down could result in fewer points at a competition or in worst case comprimise safety. By having a controller that is not custom made, if a failure happens the controller can be easily replaced and the car can get back on the track.

## 3 Why Choosing a Controller From B&R Automation

A controller from B&R Automation was chosen because it provides a fast and reliable framework to develop the control for the car. It comes with a lot of software functionalities such as data-logging, telemetry, communication interfaces some of which being CAN and RS-232 and a real-time operating system which greatly limits the amount of work needed to get an advanced control system up and running. The controllers come with visualization software which is the top of the line found in the automation industry. B&R Automation also has an office in Odense providing close support if problems occur.

### 3.1 Choosing the Controller

B&R Automation produce a series of controllers named *X90 mobile control systems* which is developed for controlling moving systems such as tractors, trailers and construction equipment. One of the use-cases of the controller is sitting on the outside of farming equipment, and being subjected to water, mud, vibrations and light impacts with objects such as brushes and small trees. This is the reason why the controlling is placed in a rugged aluminium casing. A picture of the X90 controller can be seen in figure 3.

Figure 3: A picture of the X90 mobile controller produced by B&R Automation.

It weighs more than the ideal for a race car, but the casing means it does not need to have extra protection. The advantage of the module controller is also that the I/O's are build to handle vibrations compared to B&R's other PLC solutions. The advantage of selecting a product from a company is that it comes with finished hardware that has been tested and well documented.
The X90 controller comes in 4 different versions which can further be expanded with up to 4 option boards to further extend the controller's I/O capabilities.
The X90 controller chosen was the ***X90CP174.48-00*** which was the most high-end controller option at the time of selection. Compared to the smaller versions the chosen one has a faster CPU, more onboard memory and more I/O's. A list of the most important features can be seen below together with the 4 chosen option boards used to increase the controllers capabilities.
**1pcs**: ***X90CP174.48-00*** [2]: Main controller

- ARM Cortex-A9-650
- 256 MB DDR3 RAM
- 32 kB FRAM
- 1 GB flash memory
- 1 POWERLINK on M12 connector
- 1 Ethernet 10/100BASE-T on M12 connector
- 3 CAN on CMC connector
- 48 multifunction I/O channels

The following 4 option boards
**1 pcs**: ***X90DI110.10-00*** [3]: Digital Input module.

- 10 digitial inputs with sink or source circuits
- Each input has software configurable sinking and sourcing resistors each with the following possibilities: 6.9kΩ/10kΩ/22kΩ
- Input voltage range is 9V - 32V
- Optional software input filters which can be used to filter away switching noise

**1 pcs**: ***X90IF720.04-00*** [4]: Communication module.

- 3 CAN interfaces
- 1 RS-232 interface

**2 pcs**: ***X90PO210.08-00*** [5]: Digital Output module.

- 8 digital outputs each with current measurements
- Maximum output current of 4A
- The output voltage range is 9V - 32V

## 3.2 Introduction to B&R Controllers

The code is developed, compiled, debugged, and tested on target with the program made by B&R Automation called *Automation Studio*.
More information about the B&R Automation framework can be found in the Automation Studio Starter Guide which can be found in appendix J.

# 4 Operating System - Automation Runtime

All B&R PLC's run with the B&R developed operating system called *Automation Runtime*. The operation system is hardware independent and has cyclic task scheduling of up to 8 different task classes each running either hard or soft real-time. Read more about Automation Runtime in appendix J, resource [13] and in the B&R Help under "Real-time operating system".

## 4.1 Tasks Scheduling

All task scheduling is done cyclically with static intervals decided at implentation time.
The system has upto 8 different task classes each with a configurable cycle time. After the different task class has been created each task can then be placed in the one best suited for it.
The smallest possible cycle time for the X90 controller is 1ms. Placing a task here would mean the content of the task is executed with $1kHz$. It is also possible to add tolerance to the cycle time on a task class so the system is allowed to surpass the task deadline without shutting the system down which means the class is running soft real-time. If the tolerance is surpassed surpassed the system is shut down. Hard rea-time can be achieved by setting the tolerance to $0ms$.
The cycle classes, their cycle times, cycle tolerances and usages can be seen in table 1.

| Cycle Class # | Cycle Time [ms] | Cycle Time Tolerance [ms] | Usage |
|---|---|---|---|
| #1 | 1 | 0 | Fast tasks. Used for handling the CAN communication. |
| #2 | 5 | 0 | Reserved for possible future use. |
| #3 | 10 | 0 | Inverter control and handling of I/O's used for the inverter control. |
| #4 | 10 | 10 | Reserved for possible future use. |
| #5 | 50 | 50 | State machine handling, dashboard task, AMS data handling and shutdown circuit monitoring. |
| #6 | 100 | 100 | Data logging. |

Table 1: Task classes in the system with their corresponding cycle times, cycle tolerances and usages.

Task class 1-3 has zero tolerance making them hard real-time and task classes 4-6 have some tolerance making them soft real-time which should be taken into account.

## 4.2 Task Division

For each external device in the car, such as the Dashboard, Accumulator Management System and Inverters, a group of tasks are made available to handle the control of that particular device. Each group is divided up into a set of tasks each with a different abstraction level.
On figure 4 an example of a set of task groups with their respective tasks can be seen. The higher on the figure a task exists the higher abstraction level.

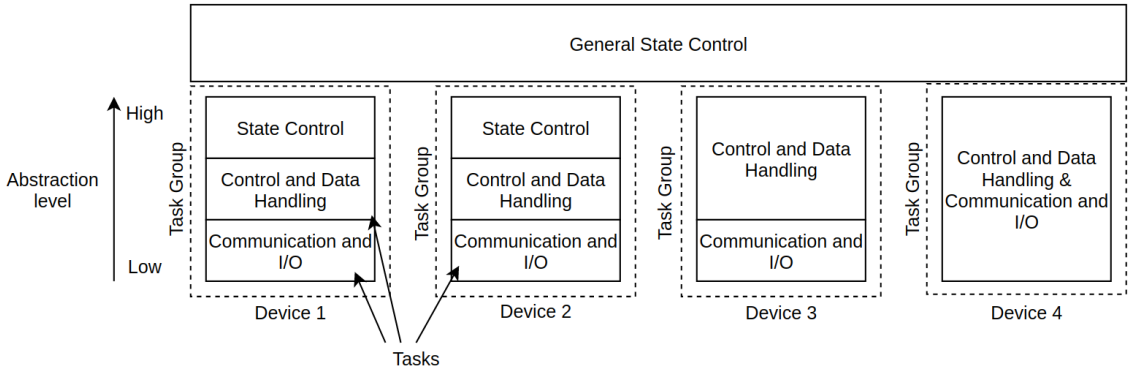


Figure 4: Illustration showing the general idea of the task division principle used in the project.

The lowest level is *Communication and I/O*, then comes the *Control and Data Handling* and in the top of each task group is *State Control*. Above all modules are

the general state control of the car. Depending on the complexity of each device some layers are combined and some (usually state control) completely left out. Examples of this could be the inverters (discussed in section 15) which has a high complexity and therefore has a task in each layer while the dashboard (discussed in section 14) has a low complexity and has a single task spanning all layers. By dividing control of a module up into different tasks it also allows for handling of the tasks at different cycle times. Communication should run at a higher or equal speed than control which should run at a higher or equal speed than state control.

It is important to note that the *Communication and I/O* level do not handle the physical communication but only pushing to output buffers and popping from input buffers. It is thus not strictly necessary to handle communication this fast because it for the most part is consumed slower than it is received but by handling it fast the buffers can be smaller.

## 4.3 Inter-Task Communication

All communication between tasks are done through global variables. The system is mainly based around the producer-consumer principle. Each task or module is assigned a global structure to which it can produce data and from which other tasks can consume data. To maintain data integrity it is important that only one task produces data for any given variable. It is possible for multiple tasks (usually a task and the liveview) to produce data for the same variable but care has to be taken in this case.
Generally there are two types of data that is might be produced by a task and also can be changed during run-time by the liveview and this is commands and configuration data.
The command data integrity is maintained by only allowing the producers to set a command and the consumer of the command will clear it after it has been handled. This can in rare occasions lead to an instance of a command to be missed if two producers write to the same command at the same time. This would lead to the consumer only reading the command once and not twice. For all purposes where commands are used in the system this is not a problem because commands are for the most part used to enable features which can only be enabled once.
The second case is configuration parameters which are generally only set in the initialization part of any task and then they are not touched anymore by the task which means that the variable can freely be changed through the liveview without risking loss of integrity.

## 4.4 Tasks

The different system functionalities are divided up into tasks. Each task handled by the master controller can be seen below in table 2.
It is worth noting that task names are limited to 10 characters in Automation Runtime which often results in being forced to use abbreviations for the task names.

| Cycle Class | Task Name | Task Description |
|---|---|---|
| 1 | AMS_CAN | CAN communication with the AMS and the shunt. |
| 1 | INV_CAN | CAN communication with all the inverters. |
| 3 | INV_CTRL | Vehicle dynamics control of the inverters. |
| 4 | INV_SM | Inverter state machine. |
| 4 | SENSORS | Sensor inputs (torque pedal sensors, brake pressure sensors, steering angle sensor and low voltage battery voltage measurements). |
| 4 | ACTUATORS | Actuator outputs (RTDS and Cooling Pumps) |
| 5 | DASHBOARD | Control, data handling and communication with the dashboard. |
| 5 | AMS_CTRL | Control and data handling for the AMS. |
| 5 | SC | Data handling and monitoring of the shutdown circuit. |
| 5 | SM | Overall vehicle state machine. |
| 5 | SENSOR_NET | Data handling communication with the sensor network. |
| 5 | ALARM_MAN | Alarm Management System. |
| 6 | DATALOG | Datalogging. |

Table 2: List of the tasks implemented on the Master Controller.

# 5 Mapp Technology

*Mapp Technology* (often written *mapp Technology*) is an ever increasing suite of software components created and maintained by B&R Automation for use on B&R products.
The software suite is made to provide application engineers using B&R products with faster development time, lower lifecycle cost and the possibility of even the smallest office branches of B&R or B&R product users with the latest and greatest that B&R technology has to offer. The software is also continuously updated and improved to have the best performance, robustness and quality.
Mapp Technology is divided into a range of categories where only two of these are used in the software for the car. These are *Mapp Services* and *Mapp View*. Mapp Services is a category that itself includes a view range of software components meant contain the basic functionalities that most systems need. It includes alarm handling, performance measurements, user management, file handling and more. Mapp View is the software component responsible for handling visualizations which in the car is used for live telemetry for data monitoring and control.

## 5.1 Alarm Handling

*Mapp Services > Mapp Alarm*
For easy alarm handling the alarm management system *mpAlarmX* is used. The alarm system handles all alarms in the system and supports a wide range of alarm types such as *EdgeAlarms*, *LevelMonitoring* alarms, *RateOfChangeMonitoring* alarms and more.
Each alarm has a wide range of possible parameters that can be configured. These include error numbers, alarm severity, auto/manual acknowledge of the alarm and more. Lastly it comes with a visualization widget that makes it easy to get active alarms shown on the visualization. See how Mapp Alarm is used in the Alarm Management System in section 7.

## 5.2 Data logging

*Mapp Services > Mapp Data*
The services Mapp Data provides can be used to datalog parameter values during runtime. The function *MpDataRegPar* can be used to register any number of variables of any datatype to be datalogged. Read more about how this service is used for datalogging in section 22.

## 5.3 Receipe Handling

*Mapp Services > Mapp Recipe*
In production lines it is often the case that the line should be able to produce multiple different types of produces where each product needs the machine to work a little bit different. This could be a heating element operating at a different temperature, motors running slower or faster, or others. This recipe feature can be used to save configuration data to a file that can then be stored and loaded at a later point in time. Read more about the usage of this service for saving and loading driving missions in section 15.3.10.

## 5.4 Liveview visualization

*Mapp View*
For the liveview functionality the mapp component *mappView* is used. MappView comes with a graphical editor where preprogrammed widgets can be dragged-and-dropped into a number of liveview pages. The widgets can be scaled and styled to fit the wished theme and bound to the PLC variable. When the PLC is running it hosts a webpage through which the liveview visualization can be viewed. Read more about the liveview feature in section 21.

# 6 System Structure

The overall system design is based on the producer-consumer design pattern. Tasks in the system will be producers, consumers or a combination of both. A variable can only be produced by one task but can be consumed by any number of tasks.
Some tasks are producers of a selection of variables fx a *BrakingSystemTask* could produce a *brake_pressure* variable describing the current brake pressure and a *BrakeLightTask* might consume that same variable to determine if the brake light should be on or not.

## 6.1 Folder Division

The software project uses a special folder structure to make it consistent and easy to navigate.
In the following sections all references to code (unless otherwise specified) will be referring to folders placed under *Project/Logical/* which is labeled *Project Level* in figure 5.
Each module has their own folder with a global parameter structure in which they can read and write important information.
On figure 5 the principle behind the folder structure can be seen.

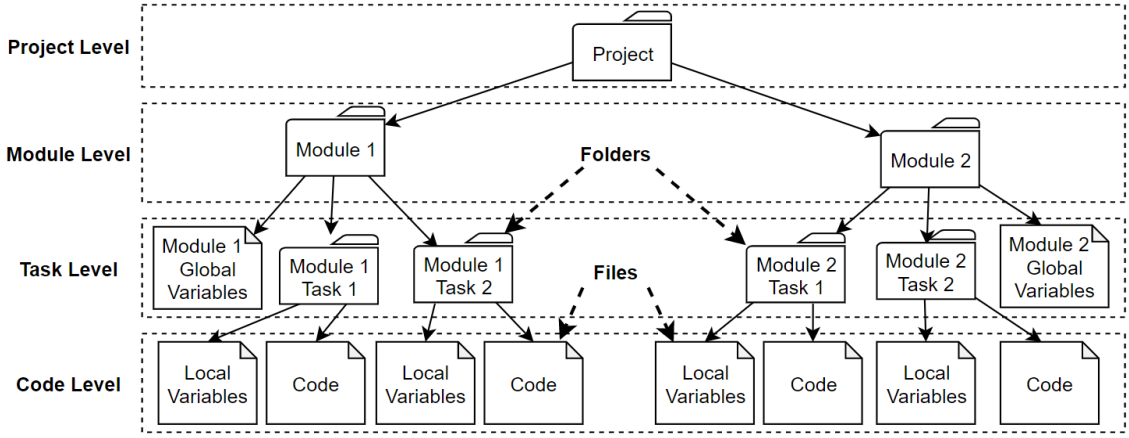


Figure 5: The folder structure principle.

In the top the overall project folder exists which in the code for the project is located at *Project/Logical/*. This folder contains a folder for each module. A module can either be an external node in the system such as the dashboard or AMS or it can be an internal object or device such as the datalogging feature or the vehicle state machine. Each module folder contains a set of folders one for each of its tasks and a file with global variables. The tasks vary depending on the module but for external devices there are usually a *Communication and I/O* task and a *Control and Data Handling* task. Each task folder contains the task code and two files used to define local variables. One file for defining the variable types and one for the variables themselves. These two files have been simplied into one file on the figure.

# Part IV
# Software

Figure 6 shows an overview of the control system implemented in the car. The system is build around a star configuration with the master controller in the center.

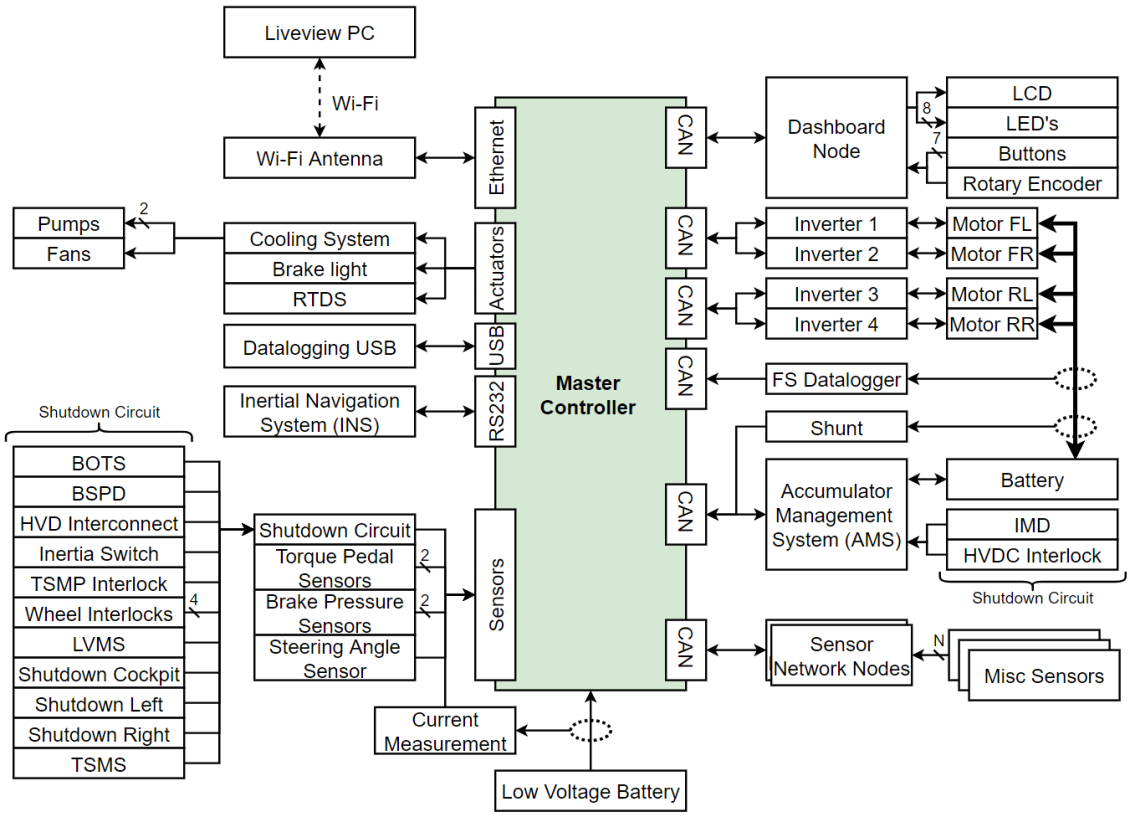


Figure 6: A general overview of the system in which the Master Controller is working.

The master controller communicates with a range of devices through CAN which can be seen on the right side of the figure. These include the *Dashboard Node* which receives commands from the driver and displays important information. The 4 *Inverters* which each drive one wheel. The *FS Datalogger* which is a datalogger unit supplied by the Formula Student events to monitor the vehicle's power consumption. The *Shunt* which also monitors the current and power consumption of the car but is placed in the car by SDU-Vikings. The *Accumulator Management System (AMS)* which is the system monitoring and controlling the accumulator/battery. Lastly it communicates with the *Sensor Network* which is a CAN bus on which a number of sensor nodes can be placed to measure different sensor values.
The master controller also communicates with a WiFi antenna which transmits telemetric data to the *Liveview PC* for live monitoring and control of the car. It communicates with a *Datalogging USB* and through RS-232 it communicates with the *Inertial Navigation System (INS)* which is used to measure location, acceleration, velocity and yaw of the car. Lastly it controls a range of devices through its digital outputs and receives data back through a range of digital and analog inputs.

The master controller is the main brain of the vehicle and it thus has to perform a wide range of tasks. What has been implemented for each of these devices in order for the whole system to work together is described in this chapter.

# 7 Alarm Management System

The alarm management system is used to keep track of all active and inactive alarms that have not yet been acknowledged. The acknowledgement can be done either automatic or manual depending the specific alarm configuration. The alarm management system is used to keep track of potential problems with the car during operation and for the most part is there to help debug the car if unexpected behavior is experienced.
An active alarm in the alarm management system will not prevent the car from driving but will provide valuable information for improving the car and remove potential bugs and errors in hardware and software.
To make the alarm management system the *MappService* called *MappAlarmX* has been used. An overview of the alarm management system can be seen in figure 7. The *ALARM_MAN* is handling the service MappAlarmX and MappAlarmX is handling the alarms. Each alarm is configured in the Alarm Configuration File which can be found in Automation Studio and in this file all alarm are created and configured.

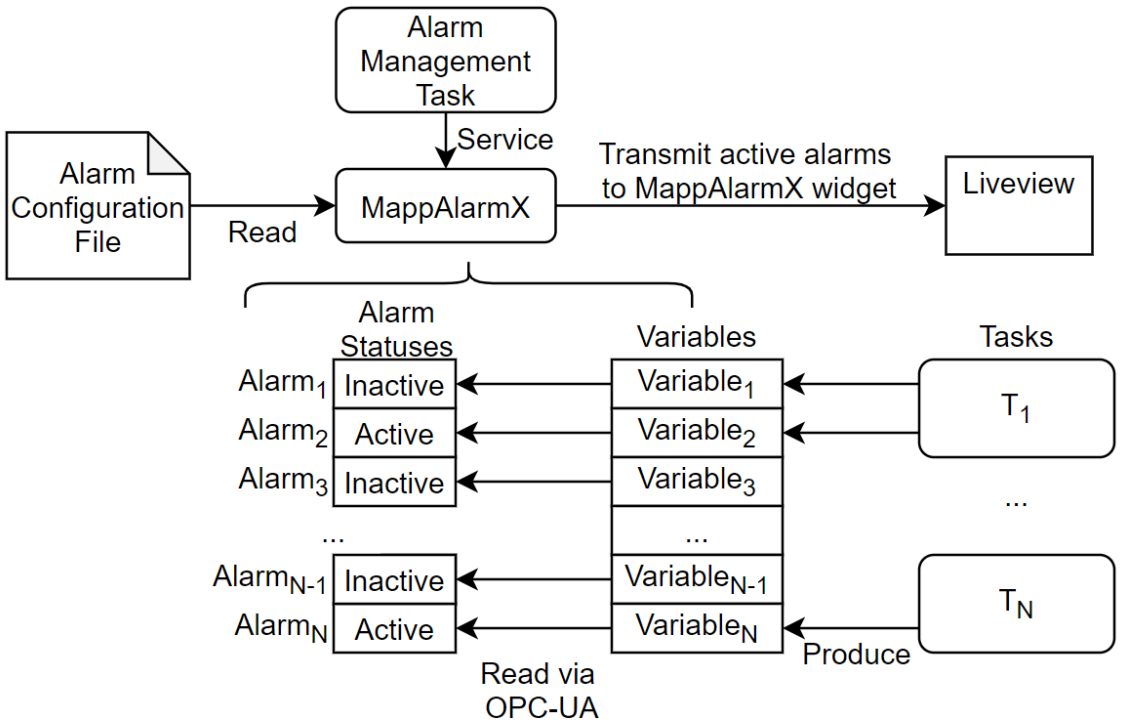


Figure 7: Overview of the functionality of the Alarm Management Task.

Information is moved around by MappAlarmX using OPC-UA which is a communication protocol used by B&R. For more information about OPC-UA see section 21 and section 11.1 in the Automation Studio Starter Guide found in appendix J.

## 7.1 Alarm Creation and Configuration

The Alarm Configuration File can in the Viking X master controller project be found at *Configuration View* > *Config1* > *X90CP174_48_00* > *mappServices* > *Config.mpalarmxcore*. In this file all the alarms that should exist in the system can be made.
Each alarm have many parameters to configure. The most relevant ones for SDU-Vikings are listed below in table 3.

| **Parameter** | **Description** |
|---|---|
| *Name* | The name of the alarm. This is used to reference this specific alarm elsewhere in the project. |
| *Message* | The message that should be shown if the alarm becomes active. |
| *Behavior* | The alarm type. This can fx. be *LevelMonitoringAlarm* or others. Read more below. |
| *Monitored PV* | The parameter which is monitored and its value will trigger the alarm. |
| *Acknowledge*<br>(Locked for some alarm types) | Possible values: *Disabled/Required*.<br>An alarm can be set to auto acknowledge itself if it goes inactive. It can also be set to *Required* to which the team has to manually acknowledge the alarm for it to go away. |
| *Multiple instances*<br>(Locked for some alarm types) | Possible values: *True/False*.<br>Determines if the alarm can only exist onces or if it should produce a new alarm message every time it goes back and forth between active and inactive. |

Table 3: Most important alarm parameters.

### 7.1.1 Alarm Types

Multiple different alarm types exist in MpAlarmX, types such as *Edge Alarm*, *Rate Of Change Alarm* and *Deviation Alarm*. The only alarm type used in the Viking X project as of the hand in of this report is the Level Monitoring alarm.

**Level Monitoring Alarm** The Level Monitoring alarm type compares the Monitored PV parameter to a set of limits and if the parameter passes any of these limits the alarm goes active. The alarm has 4 limits which can be used, these are a High limit, a HighHigh limit, a Low limit and a LowLow limit. The High and Low limits are triggered if the parameter goes above or below the them. The alarm can also have a HighHigh limit which is above the High limit. This limit can fx be used to allow the alarm to give a warning on the High limit and shutdown the system on the HighHigh limit. The same goes for the LowLow limit but in the opposite direction. These additional outer limits are not used in the Viking X alarm system.
The LevelMonitoring alarm has the following paramters.

- **Low Limit**: Possible values: *Static/Dynamic*

  **Limit / Limit PV**: The alarm will be triggered if the Monitored PV parameter goes below this value. *Limit* is used for static alarms and will be a fixed value. *Limit PV* is used for dynamic alarms and will be a pointer to a variable which will be the limit and can change during runtime.

  **Limit Text**: A string associated with the alarm. The string can be included in the overall alarm message. If it is not included this text will not be visible anywhere.

- **High Limit**: Possible values: *Static/Dynamic*

  **Limit / Limit PV**: The alarm will be triggered if the Monitored PV parameter goes above this limit.

  **Limit Text**: Same as for the Low Limit.

See appendix B for a list of all alarms implemented in the system and a short description of each.

# 8 State Machine

The car's general state machine is one of the most important tasks running on the master controller. The state machine task itself only keeps track of the state of the vehicle and makes this information public for all other tasks.
The vehicle state is used by almost every other task in the system to change their behaviors. The vehicle state machine can be seen in figure 8.

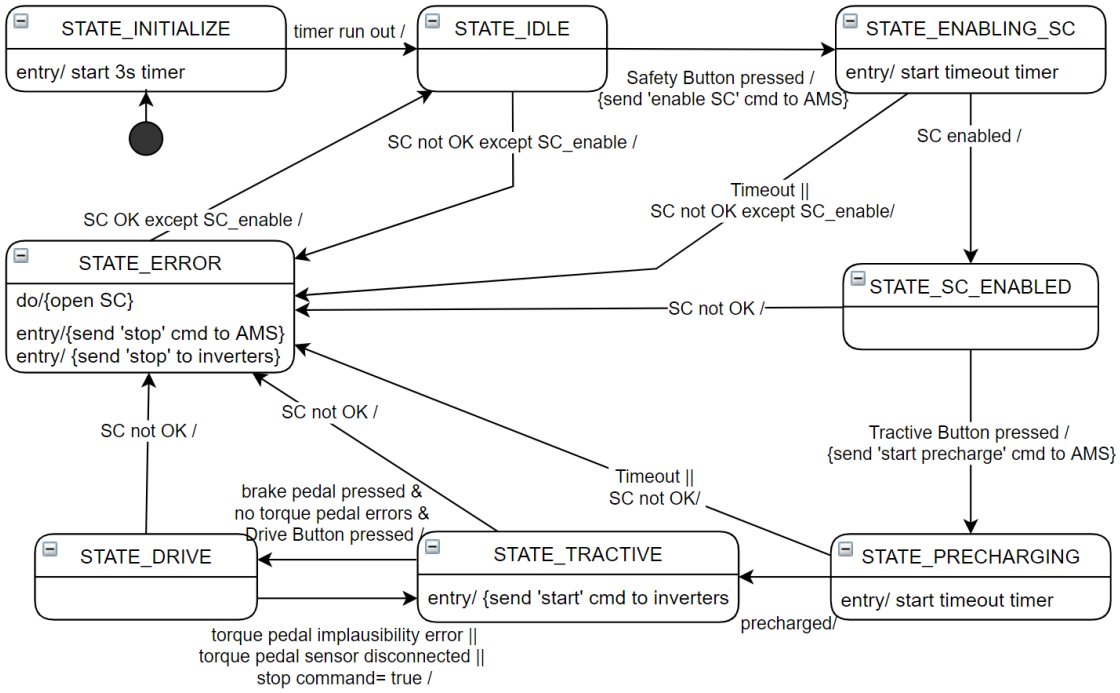


Figure 8: An overview of the vehicle general state machine.

It can be seen that the figure has 8 different states where the car starts in *STATE_INITIALIZE*. After 3 seconds it transitions to *STATE_IDLE* which is the car's default safe state.
On the state machine some actions are enclosed with "{ ... }". These are there to show the action is performed but not directly by the state machine task itself. These actions are performed by others tasks triggered by the state machine task state or state transition.
If an error happens to the shutdown circuit (SC) the state machine will transistion to *STATE_ERROR* in which is will stay until the SC is fixed. The last part of the SC called *SC_enable* is allowed to be open in state *STATE_IDLE* and *STATE_ENABLING_SC* because this element is enabled in *STATE_ENABLING_SC*. From *STATE_IDLE* the state machine can transision to *STATE_ENABLING_SC* by pressing the *Safety Button* and the master controller will tell the Accumulator Management System (AMS) to enable *SC_enable*. A timer is started in this state and if the AMS do not respond back with a confirmation about the SC being enabled a timeout will happen and the state machine transistions to *STATE_ERROR*. From *STATE_ENABLING_SC* a transistion to *STATE_SC_ENABLED* happens if the SC is correctly enabled by the AMS. From here a transistion to *STATE_PRECHARGING* can be caused by pressing the *Tractive Button* and a command is send to the AMS to start precharging. Once again a timeout can occur if no respond sent back from the AMS within the allowed time. If the precharging is successful the state machine transistions to *STATE_TRACTIVE*. In this state the tractive system is active, and a start command is sent to the inverters. This is the state all the competition tests related to safety of the tractive system without the car being in *STATE_DRIVE* is performed.
Before the state machine is able to enter *STATE_DRIVE* it needs to be made sure that the car is safe to set into drive. This includes making sure the torque pedal sensors are connected and working properly. Rule *T11.8.9* states that if a implausibility error between the torque pedals occur the car should exit *STATE_DRIVE* and by rule *T11.8.8* it is allowed to exit back to *STATE_TRACTIVE*. A combination of the rule *T11.8.7* and section *T11.9* results in the competition safety test 276 & 277 found in the FSG inspection sheet [6] where if $\geq 50\%$ of the torque pedal sensors are disconnected the motors should stop. The way this is implemented is to transistion to *STATE_TRACTIVE* if one or both of the torque pedals are disconnected.
Both implausibility and sensor disconnection have also been implemented to prevent the car from entering *STATE_DRIVE* in the first place if deteceted. Read more about the implausibility error detection in section 9.1.3 and torque pedal disconnection in section 9.1.4. When the torque pedal sensors are working properly the brake pedal has to be pressed which is stated in rule *EV4.11.6* together with pressing the *Drive Button* to continue into *STATE_DRIVE*.
In *STATE_DRIVE* the car will drive when the torque pedal is pressed.
For safety reasons a stop commando can be send to the car from the liveview visualization to remove the car from the driving state.
Rule *EV4.11.7* states that *STATE_DRIVE* must be left immediately if the shutdown circuit is opened which can be seen with the transition from *STATE_DRIVE* to *STATE_ERROR*. To exit *STATE_DRIVE* in normal conditions this requirement

has just been reused and the driver should thus just pressing the cockpit shutdown button to leave the state.
The timeout period used in *STATE_ENABLING_SC* and *STATE_PRECHARGING* is set to 5 seconds.

## 8.1 State Changing Delay

To avoid passing through states and only being in a state for 1-2 cycles, a delay has been implemented, reducing the speed at switch states can be changed. It sets a minimum time that a state has to be active before it can be changed.
The feature is mainly implemented to avoid short voltage dips on the shutdown circuit to change states. The timer handling and conditions for transisioning are present in all states seen in figure 8 but have been left out for clarity.

## 8.2 Instructions

Getting a race car to work and especially a formula student car might be difficult for a new driver because the car contains many specialized components and procedures. To make it easier for a new driver or an experienced driver in a stressed situation the instruction feature has been implemented. For each step of the state machine a human readable string is printed out telling the users what to do next to get the car into *STATE_DRIVE* in which the driver can drive the car.
The feature also helps the driver comply with rule *EV4.11.1* stating that the driver must be able to active and deactivate the tractive system from within the car. This rule is mainly focused on the fact that the driver should be physically able to turn the tractive system on and off but if the driver does not know how, then this physical aspect is of no use.
What specific string that is shown to the driver is determined by an instruction index. An index is chosen because the instruction should be shown on both the liveview visualization as well as on the dashboard display used by the driver. By choosing an index then only a single byte needs to be transmitted to the dashboard. If the ASCII characters making the messages should instead be transferred this would result in the instruction message varying in size depending on the current instruction and worst case be 36bytes long. The CAN standard specifies that a message cannot be longer than 8 bytes which would result in 4 x 8 byte messages and 1 x 4 byte message messages being need to transfer a single instruction.
The index method is less dynamic because the two controllers need to agree on the instruction and a change would require both controller to be updated. It was chosen because it reduces system complexity and lowers CAN bus load.
An instruction lookup table can be seen in table 4.

| Index | Instruction |
|---|---|
| 0 | "Initializing. Wait" |
| 1 | "Press Safety Button" |
| 2 | "Wait for SC to be Enabled" |
| 3 | "Press Tractive Button" |
| 4 | "Wait for AMS to be Ready" |
| 5 | "Press Brake Predal" |
| 6 | "Torque Pedal Fault" |
| 7 | "Pre-Charging. Wait" |
| 8 | "Press Drive Button" |
| 9 | "Remove Error" |
| 10 | "Close Shutdown Circuit" |
| 11 | "To Exit Drive: Press Shutdown Button" |

Table 4: Instructions indices and their corresponding instruction.

# 9 Sensors - Input Signals

In the following section the sensors connected directly to the master controller are descripted as well as how their measured values are converted to useful data. It is possible for sensors to be disconnected or break so error checks are being performed on the sensor values to determine if the data is safe to use and to follow rules *T11.8.7*, *T11.8.9*, *T11.9* and inspection sheet points *276 & 277*.

## 9.1 Torque Pedal

The torque pedal uses two linear potentiometers mounted at different angles resulting in them having different transfer functions as is required by rule *T11.8.6*.

### 9.1.1 Convert sensor reading to pedal position

The sensor readings need to be converted to a pedal position in percent where 0% means the pedal is not being touched and 100% is the pedal being all the way down, which is stated in rule *T11.8.3*. To convert the sensor readings to percent a simple linear conversion is used.
A reading is done for the sensor value at pedal position 0% and at pedal position 100%.

$$P_{\%} = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)x - b \tag{1}$$

Where $y_1 = 0\%$, $y_2 = 100\%$, $x_1$ is the sensor reading at 0%, $x_2$ is the sensor reading at 100%, $x$ is the new sensor reading and $b$ is the y-axis crossing which is the sensor reading at 0%. This results in the following.

$$P_{\%} = \left(\frac{100\% - 0\%}{P_{100\%} - P_{0\%}}\right)P_{reading} - P_{0\%} \tag{2}$$

To make sure small variations in the measurements does not result in values under 0 or over 100 the values found are limits to within these two values.

The calculations are done for both torque pedal sensors with different $P_{0\%}$ and $P_{100\%}$ values because the torque sensors have different transfer functions.

**Future work**
A non-linear torque pedal mapping might be implemented later to improve vehicle handling at low speeds.

### 9.1.2 Out-of-Range Detection

For safety reasons it is important to know if the values coming from the torque pedal sensors are reliable and because of rule *T11.8.9*. Therefore a detection is implemented to check is any of the sensor values leave the normal operation window. This type of detection has been named *out-of-range detection*.
The error detection is applied to both torque pedal sensors in the same way so it will only be discussed for a single one.
If the sensor value leaves a predefined value window an alarm is triggered. If the vehicle state machine is in *STATE_DRIVE* and one of the torque pedal sensor values leave the operation window the car will change to *STATE_TRACTIVE* disabling the ability to drive forward. (See more about the vehicle state machine in section 8)
On figure 9 an example of sensor values over time can be seen.

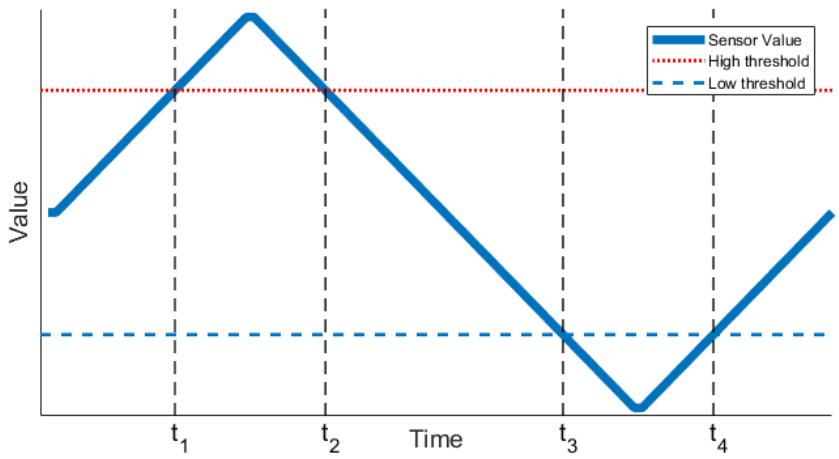


Figure 9: Out-of-range detection

At time $t_1$ and $t_3$ the sensor value leaves the operating window and here two things happen.
1) A boolean variable used to hold the status of the alarm is set high. This variable is used throughout the code to know if the sensor has an *out-of-range error*. When the variable once again enters the working range this alarm will go away which would happen at time $t_2$ and $t_4$.
The code implemented can be seen in code snippet 1. The code has been simplified. In reality the error signals exist in *gTorqueSensor.status*, the sensor input values are in *gTorqueSensor.inputs* and the limits are defined in a struct called *constants*.

```
sensor1_out_of_range :=   torque1 > high_limit1 OR torque1 < low_limit1;
sensor2_out_of_range :=   torque2 > high_limit2 OR torque2 < low_limit2;
```

Code snippet 1: Out-of-Range Detection

The variables *sensor1_out_of_range* and *sensor2_out_of_range* are the boolean variables used throughout the code to know if an error has occurred.
This error handling comes with the downside that an error might be missed because once the torque pedal again enters the working range it disappears. Thus a second error handling is implemented.
2) At the same time as the boolean alarm is triggered, an alarm is also parsed to the alarm management system.
The alarms type used in the alarm management system for *out-of-range alarms* are made to only allow one instance of the alarm to be active at a time and that an alarm has to be manually acknowledged.
Only allowing one instance means if the value moves back into the allowed range and back out of range again no new alarm will be triggered. If the value is only outside the range for a very short period the alarm can still be noticed because the alarm needs a manual acknowledge before disappearing.

### 9.1.3 Implausibility Detection

The rules state that there should be at least two sensors on the torque pedal with different transfer functions. If the torque pedal positions are found in percent and they are more than 10% points different for more than 100ms an implausibility error needs to be raised. See rules *T11.8.8* and *T11.8.9*.
To make it possible to control the car with a single potentiometer for testing it is made possible to disable the plausibility check. The code used to implement the detection can be seen in listing 2.

```
ACTION TorqueSensorsOutOfRange:
  IF cmd_disable_plausibility_check THEN
    plausibility_error := FALSE;
  ELSE
    (* Check if torque sensor 1 and 2 vary more than 10 % *)
    plausibility_error := ABS(torque_1_percent - torque_2_percent) > 10;
  END_IF;
END_ACTION
```

Code snippet 2: Implementation of the torque pedal implausibility detection.

**Warning**
The implausibility detection has to be enabled when used in the physical car.

### 9.1.4 Disconnection Detection

No additional functionality has been added to be able to detect if one of the torque pedal sensors are disconnected because that is part of the lower threshold of the out-of-range detection implemented in section 9.1.2. When wires are disconnected the ADC connected to the input will measure values below the decimal value ”100” so as long as the normal sensor input never goes below this and that the lower threshold is above this everything is handled by the out-of-range detection.

### 9.1.5 Torque Off

As specified by rule *EV2.3.1* the driving torque should be turned off if the braking is pushed hard at the same time as either the torque pedal is over 25% or more than $5kW$ is outputtet. The the torque should not be regained before the torque pedal is lifted to below 5% pedal travel as stated in rule *EV2.3.2*.
A flowchart diagram showing the solution of this, can be seen on figure 10.

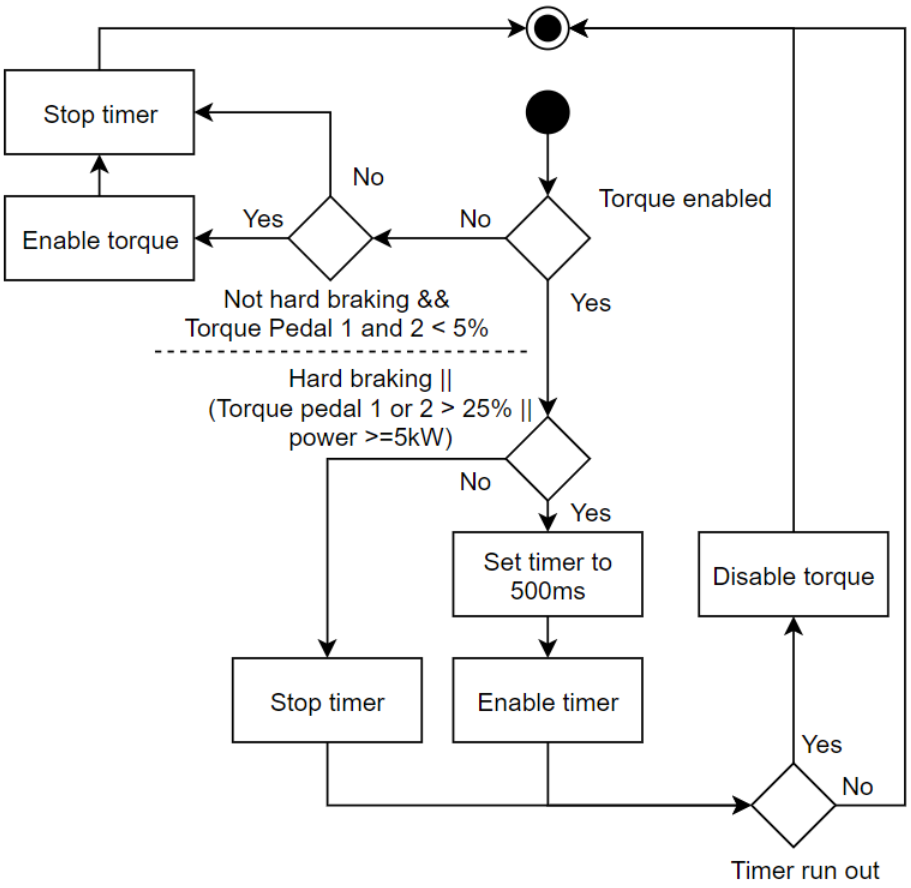


Figure 10: Flowchart diagram for turning off the driving torque.

The timer used is the TON timer described in section 23.2.

```
ACTION TorqueOff:
  IF torque_enabled THEN
  (* Test the criterias for turning off the torque. If they are met start timer *)
    (* Set timer duration *)
    TON_torque_off_timer.PT := T#500ms;

    (* Start timer if all conditions are fulfilled *)
    TON_torque_off_timer.IN :=  (hard_braking AND
                                (torque_1_percent >= torque_off_upper_hysteresis OR
                                 torque_2_percent >= torque_off_upper_hysteresis OR
                                 actual_total_power_kw >= 5.0));
    (* If timer runs out *)
    IF TON_torque_off_timer.Q THEN
      (* Remove torque *)
      torque_enabled := FALSE;
    END_IF;
  ELSE
```

```
    (* Test the criterias for turning on the torque *)
    torque_enabled :=    (NOT hard_braking AND
                          torque_1_percent < torque_off_lower_hysteresis AND
                          torque_2_percent < torque_off_lower_hysteresis);
    (* Make sure the torque timer is stopped *)
    TON_torque_off_timer.IN := FALSE;
  END_IF
  (* Run torque timer *)
  TON_torque_off_timer();
END_ACTION
```

Code snippet 3: Implementation of the Torque Off Feature

## 9.2 Brake Pressure

The brake pressure subroutine's main purpose is to monitor the brake pressure. It has the same out-of-range detection as the torque pedal signals which can be seen in section 9.1.2.
The subroutine also produces two boolean variables *soft_braking* and *hard_braking* which are used around the system for triggering various features. Most notably *soft_braking* is used for brake light control described in section 11.1 and *hard_braking* is used by in the Torque Off feature described in section 9.1.5.

## 9.3 Steering Wheel Angle

Attached to the steering coloumn is an angle sensor that is used to measure the current steering wheel angle. The angle is used for liveview purposes and for torque vectoring discussed in section 15.3.2.

### 9.3.1 Convert to steering angle

The steering angle is measured as a voltage and has to be changed to an angle in degrees.
Equation 3 is used to linearly convert the measured voltage to an angle. The steering angle sensor has its highest value when turned completely to the right and lowest when turned to the left.

$$A_{steer,deg} = \left(\frac{|A_{max}| + |A_{min}|}{A_{right} - A_{left}}\right)(A_{reading} - A_{left}) + A_{min} \tag{3}$$

Where $A_{steer,deg}$ is the steering angle in degrees, $A_{max}$ is the measured value when the steering wheel is all the way to the right, $A_{min}$ is the measured value when the steering wheel is all the way to the left. $A_{right}$ is the maximum value which should be obtainable when turning right and $A_{left}$ is the maximum value obtainable when turning left and $A_{reading}$ is the value read from the steering angle sensor.

### 9.3.2 Out-of-range Detection

Implemented as for the torque pedal signals which can be seen in section 9.1.2.

### 9.3.3 Disconnection Detection

Implemented as for the torque pedal signals which can be seen in section 9.1.4. For safety reasons it is important to know if the sensor is disconnected because the torque vectoring controller use this signal and can behave unpredictable if wrong values are used. Read more about the controller in section 15.3.2.

## 9.4 Low Voltage Battery

The car can behave undesireable when the low voltage battery gets low on charge. It should not be a problem in the actual races if the battery has an appropriate capacity and is fully charged before a race. It can become a problem during testing or if the battery is not fully charged before a race.
Therefore monitoring of the low voltage battery is implemented to warn about the battery being low and that undesireable behavior might soon occur. The function is not used to stop the car but is only used for monitoring and can be used to manually stop the car. The warning is shown both on the liveview visualization and on the dashboard warning the driver.
A voltage monitoring is implemented to warn when the battery voltage gets below a predefined threshold.

### 9.4.1 Voltage monitoring

The discharge voltage curve for a battery follows an 's'-curve where the voltage dips significantly when the battery is near being empty. The low threshold in voltage has been set at $11V$ which is a bit lower than the nominal voltage but higher than the minimum voltage. It should just provide information about when the voltage started dropping away from the norminal voltage of $12V$ provided by the battery.
When the AIR's are engaged a high current is drawn from the LV battery and the voltage dips shortly. To prevent the LV battery monitoring for triggering on this and simular events a timer has been implemented so the voltage needs to be below the threshold for over $1s$ before the warning is triggered. An example of this can be seen on figure 11 where the voltage dips for a short period.

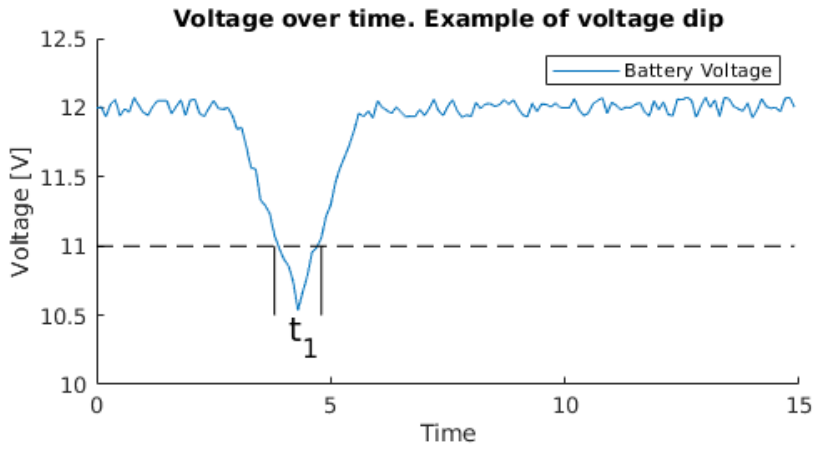


Figure 11: An example of the voltage on the lv battery during a voltage dip. Note that the values in the graph are **not** measurements or simulations but generated arbitrarily to show the discussed scenario.

If the period $t_1$ is over the allowed threshold the low battery voltage error will be given. It is important to tune the threshold time to not get false warnings.
On figure 12 an example of a discharge curve can be seen. When the voltage is below the threshold for more than the allowed time the low voltage warning will be given.

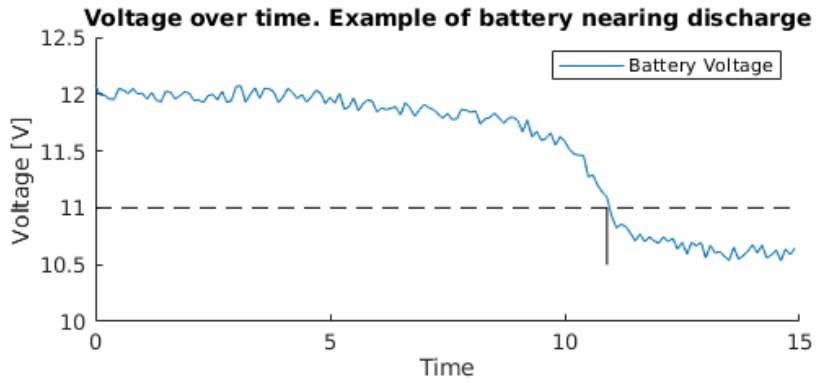


Figure 12: An example of the voltage on the LV battery when the battery is close to being completely discharged. Note that the values in the graph are **not** measurements or simulations but generated arbitrarily to show the discussed scenario.

**Future work**
In the future a coulomb counter might be implemented alongside the voltage measurement to provide more inside into the current status of the LV battery state of charge.

# 10 Shutdown Circuit (SC)

The car has a shutdown circuit which is required by the rules under section *EV6.1*. The shutdown circuit can shut off the motors if a safety critical component is triggered such as an emergency button being pressed. An overview of the shutdown circuit can be seen in appendix E.
To make it easy to identify and locate errors in the shutdown circuit a measurement point has been added between almost all elements of the circuit. The flowchart of how the shutdown circuit is monitored can be seen below in figure 13.

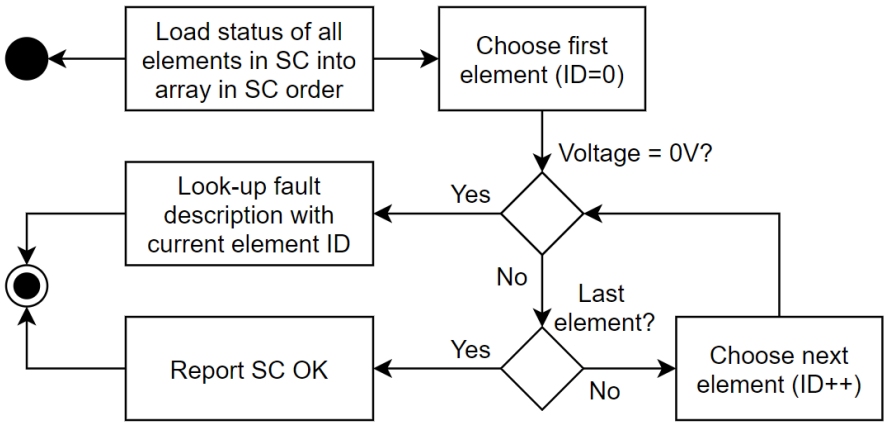


Figure 13: Flowchart of the shutdown circuit control.

The SC elements are first arranged in an array ordered after how they appear in the shutdown circuit with the first element being first in the array and last being last. Then the array is cycled through. If an element is not OK meaning an open circuit the cycling is ended and an error is reported describing what element was broken which can be done with the index of the element. If all elements are okay the cycling reaches the end and the shutdown circuit is reported OK.
The shutdown circuit also produces a parameter called *SC_ok_not_enabled* which means that all elements before the enable part is OK. This variable is used in the central state machine discussed in section 8 to transition to *STATE_ERROR* from *STATE_IDLE* and *STATE_ENABLING_SC*.

# 11 Actuators - Output Signals

The actuators / output signals are digital outputs coming directly from the Master Controller. The actuators consist of the brake-light and the Ready-To-Drive-Sound. It also contains the cooling system pumps and fans but those are discussed separately in section 12.

## 11.1 Brake-Light

The brake-light is controlled electrically and is used to show if the car is hydraulically braking which is required by rule *T6.3.1*.
The light is controlled with the *soft_braking* parameter outputtet from the *brake_pedal* subroutine in task *SENSORS*. The signal goes high when the brake pressure on either the front or rear brake system is higher than a predefined level. The level is chosen by testing how much the brake pedal needs to be pressed down for the hydraulic brakes to engage and the sensor reading at which this happens is noted down and used as a limit.
When the car is in *STATE_INITIALIZE* the brake light is ON no matter the brake pressure and the brake-light is only controlled by the brake pressure when the car is not in this state.

## 11.2 Ready-To-Drive-Sound (RTDS)

When the car enters *STATE_DRIVE* the RTDS timer is set to 2$s$ and started as is required by rule *EV 4.12.1*. While the timer is counting down the digital output going to the RTDS is set high. When the timer finishes, the output is set low again. The timer used is the TON timer described in section 23.2.

```
(* If the car is switched into state DRIVE, start RTDS *)
IF state = STATE_DRIVE AND state_old <> STATE_DRIVE THEN
  cmd_enable_rtds := TRUE;     (* Set command RTDS = true *)
END_IF;

(* Check for command to turn on timer *)
IF cmd_enable_rtds THEN
  TON_rtds_timer.IN    := TRUE; (* Enable timer *)
  rtds_output          := TRUE; (* Set output high *)
END_IF

rtds_timer();      (* Handle timer *)

(* If timer runs out *)
IF TON_rtds_timer.Q THEN
  cmd_enable_rtds      := FALSE; (* Disable command *)
  rtds_output          := FALSE; (* Set output low *)
  TON_rtds_timer.IN    := FALSE; (* Disable timer *)
END_IF
```

Code snippet 4: Code implemented to control the Ready-To-Drive-Sound buzzer.

### 11.2.1 Disconnection Detection

A warning feature is implemented to detect if the RTDS is connected. This is done by measuring the current drawn by the RTDS when it is supposed to be turned on. If it does not draw the currect amount of current, it is reported disconnected and an alarm is send to the alarm management system.

# 12 Cooling System

The cooling system is responsible for keeping inverters and motors within the allowed temperature range during operation. The system contains 2 cooling pumps and 1 cooling fan.

## 12.1 Temperature Monitoring

In the system setup as of this documents hand-in the cooling system task itself does not measure any temperatures itself and therefore the monitoring part does not have a task for itself. Temperatures are measured by other devices such as the AMS and the inverters. All of these temperature measurements are collected on a shared visualization page which can be seen in appendix I in section I.3.

**Future work**
In the future there might come temperature sensors that are connected directly to the cooling lines and these will either be monitored by the cooling system task or the sensor network discussed in section 18.

## 12.2 Cooling Fan

The cooling fan output are set high when the controller turns on. There is no control or error detection associated with the cooling fan.

## 12.3 Cooling Pump Control

The cooling pumps are the main way that the motors and inverters are cooled down. Without the cooling pumps being active the system has a high risk of overheating and destroying expensive equipment. The cooling pumps are therefore supposed to be on at all times when the car is on. If an error occurs in the cooling system the team is notified by an error in the alarm management system. Because it takes the car some time to overheat the alarm is not made critical but only implemented as a warning so the team can take the decision of continuing or stopping the car.
A flowchart of the cooling pump control can be seen in figure 14.

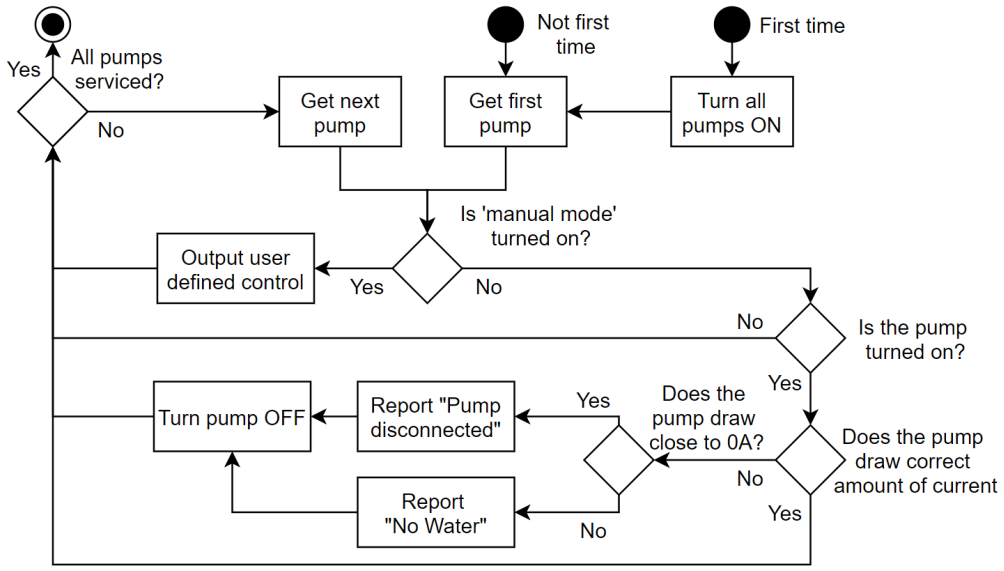


Figure 14: Flowchart of the cooling pump control.

The control starts with turning on all pumps and then constantly monitors the current. If any of them are not drawing the correct amount of current that means there is either no water in the cooling system or the pump is disconnected. The error is found by looking at how much current is drawn and an error reported to the alarm management system and the pump is turned off. The pumps can also be put into manual mode where the error handling is turned off and the user can through the liveview manually control the pumps.
If a pump is turned off by the control it will not be turned back on again. There are two ways to turn a pump back on after it has been disabled by the control.

1. Power cycle the system.
2. Set the pump to manual mode, turn it on and turn manual mode back off. If the pump error has been removed it should stay on, otherwise it will turn back off.

### 12.3.1 No Water Detection

To make sure the cooling pump does not run without water which could damage it. The build in current measurement of digital outputs of the master controller is used to measure how much current the cooling pump is drawing while being on. The cooling pump draws different amount of current depending on whether it is pushing water through the cooling system or not. The cooling pump is programmed to turn off if it is on but not enough current is drawn.

### 12.3.2 Disconnection Detection

The same feature as for detecting if no water is in the system can also be used to detect if the cooling pump is not connected. If the current is $0A$ when the cooling pump is supposed to be on, that means the pump is not connected. If the cooling pump is not connected an error will be reported because motors or inverters might overheat and get damaged if the cooling system is not operating properly.

# 13 Communication

The Master Controller uses different communication protocols to communicate with various devices in the car. The following section first shows the different ports on the X90 controller. Then it goes through the CAN address space division. CAN is the most used communication protocol in the car and for this reason a generic CAN module has been made. A description of this can be found in section 23.3.

## 13.1 Hardware Ports

On PLC's from B&R every hardware interface such as those used for communication each have an address assosiated with them. Ports can be children of other ports and in those cases each step of the hierarchical tree structure is separated by a dot.

| Port Name | Port Address | Usage |
|---|---|---|
| Ethernet | IF2 | Wifi antenna |
| Powerlink | IF3 | Reserved |
| USB | IF4 | Datalogging |
| CAN | IF7 | Inverters 1&2 |
| | IF8 | Inverters 3&4 |
| | IF9 | AMS & Shunt |
| | IF6.ST1.IF1 | Dashboard |
| | IF6.ST1.IF2 | Sensor Network |
| | IF6.ST1.IF3 | FS Datalogger |
| RS-232 | IF6.ST1.IF4 | INS |

Table 5: Overview of the hardware ports, their addresses and uses.

On the X90 controller used for the Viking X car most communication ports are directly connected to the main board and thus have short addresses. The only exception is the interfaces located on the expansion board *X90IF720.04* which adds 3 CAN ports and a RS-232 interface. The option board has the address *ST1* which is added internally to the X90 through a X2X communication bus with the address *IF6*. The three CAN ports have the addresses *IF1*, *IF2* and *IF3*. Their port addresses can be seen as the last 3 addresses under CAN in the table below as well as the RS-232 port.

## 13.2 CAN Communication

Most of the node communication is done through CAN. Therefore a generic CAN module is made that is used as a base for the communication with the nodes. Some nodes have special requirements and therefore use slight variation of the base module. Most nodes are connected to the Master Controller each with a separater CAN bus. To minimize confusion and add the possiblity of combining buses in teh fiture the address space has been divided up to be completely sure no nodes are using the same addresses. This allows any combination of the CAN buses to be merged because no messages have the same ID's. The addresses has been divided up into chunks of 128 addresses as can be seen in table 6.

| | **Address in Decimal** | | **Address in Hexadecimal** | | |
|---|---|---|---|---|---|
| **ID** | **From** | **To** | **From** | **To** | **Reserved For** |
| 0 | 0 | 127 | 0x000 | 0x07F | |
| 1 | 128 | 255 | 0x080 | 0x0FF | |
| 2 | 256 | 383 | 0x100 | 0x17F | |
| 3 | 384 | 511 | 0x180 | 0x1FF | Inverters TX |
| 4 | 512 | 639 | 0x200 | 0x27F | |
| 5 | 640 | 767 | 0x280 | 0x2FF | Inverters RX |
| 6 | 768 | 895 | 0x300 | 0x37F | AMS |
| 7 | 896 | 1023 | 0x380 | 0x3FF | |
| 8 | 1024 | 1151 | 0x400 | 0x47F | FS Datalogger |
| 9 | 1152 | 1279 | 0x480 | 0x4FF | |
| 10 | 1280 | 1407 | 0x500 | 0x57F | Shunt |
| 11 | 1408 | 1535 | 0x580 | 0x5FF | Dashboard |
| 12 | 1536 | 1663 | 0x600 | 0x67F | Sensor Network |
| 13 | 1664 | 1791 | 0x680 | 0x6FF | |
| 14 | 1792 | 1919 | 0x700 | 0x77F | |
| 15 | 1920 | 2047 | 0x780 | 0x7FF | |

Table 6: An overview of the CAN bus message ID address space.

# 14 Dashboard

The dashboard is the driver's main source of information about the vehicle's current status. The dashboard contains 8 LED's which each have a special meaning described below as well as an LCD display that is used to display important information such as vehicle velocity, cooling system temperature, accumulator state of charge, errors, warnings and instructions. The driver can interact with the vehicle throught a set of buttons and a rotary encoder.
The dashboard node is made by SDU-Vikings.

## 14.1 LED Control

The Dashboard LED's are controlled by the master controller. The status of each LED is sent cyclically through CAN to the dashboard.

| LED Name | Description | Rule |
|---|---|---|
| *TS_OFF* | Light up with the Tractive System (TS) is deactivated. | *EV4.10.9* |
| *AMS* | Light up with the Accumulator Management System (AMS) has opened the Shutdown Circuit (SC). | *EV5.8.8* |
| *IMD* | Light up when the Insulation Monitoring Device (IMD) has opened the Shutdown Circuit (SC). | *EV6.3.7* |
| *EBS* | Light up when the Emergency-Brake-System (EBS) detects a failure. | *DV3.2.7* |
| *SC* | Light up with *SC_enable* part of Shutdown Circuit is enabled, blink when *SC_enable* is the only part of the SC that is not enabled. | - |
| *RDY* | Blink when pressing the brake pedal and pressing the *Drive* button will change the car to *STATE_DRIVE*. When the brake pedal is pressed from here the LED stays ON and it will also be ON while in *STATE_DRIVE*. At all other times the LED is off. | - |
| *LV* | Light up when the low voltage system is turned on. Blink when LV battery is low on charge. | - |
| *DV* | Light up when the car is in active self-driving mode. | - |

## 14.2 Buttons

The dashboard has 7 buttons. Three of these being the *Safety Button*, *Tractive Button* and *Drive Button* which are used to set the car into *STATE_DRIVE*. See section 8 for more information about the vehicle's main state machine. The other 4 buttons are there in reserve in case it is necessary to enable or disable special features.

## 14.3 Display

The display is used to convey important information about the current state of the car to the driver. The information is sent from the Master to the Dashboard using CAN. The information transmitted is the following:

| Variable Name | Type | Description |
|---|---|---|
| INSTR_INDEX | UINT | Index that can be used to look up instruction telling what to do next to get the car in drive. |
| CAR_STATE | UINT | The current state of the vehicle. |
| ERROR_CODE | UINT | Index that can be used to look up error. |
| ACT_SPEED | REAL | Current vehicle velocity in $km/h$. |
| SOC | REAL | Current state of charge on accumulator in %. |
| MAX_TEMP | REAL | Current highest motor temperature in $^oC$. |
| LV_BAT_VOL | REAL | Current voltage of low voltage battery in $V$. |
| MISSION_INDEX | UINT | An index telling what mission the car is in.* |
| STATUS_TV | BOOL | Status if torque vectoring is enabled. (0 = OFF, 1 = ON) |
| STATUS_TC | BOOL | Status if traction control is enabled. (0 = OFF, 1 = ON) |
| STATUS_PL | BOOL | Status if power limiting is enabled. (0 = OFF, 1 = ON) |
| STATUS_TL | BOOL | Status if torque limiting is enabled. (0 = OFF, 1 = ON) |
| STATUS_LP | BOOL | Status if launch profile is enabled. (0 = OFF, 1 = ON) |
| STATUS_RB | BOOL | Status if regenerative braking is enabled. (0 = OFF, 1 = ON) |
| RB_AVAILABLE | BOOL | Status if regenerative braking is available. (0 = NO, 1 = YES) |
| TORQUE_LIMIT | REAL | The torque available for the current mission in $Nm$. |
| POWER_LIMIT | REAL | The power available for the current mission in $kW$. |

* Read more about driving mission in section 15.3.10.
Read more about how the exact format the data is transmitted with in the Dashboard CAN protocol in appendix G.

## 14.4 Change Driving Mission

If the car is not in the current driving mission this can be changed from the dashboard. (Read more driving mission in section 15.3.10). The driver can through interaction with the dashboard get it to send asynchronous messages to the master controller requesting a change in the current driving mission.
It can be done in two ways.

- Change between missions
- Modify the current mission

By changing to another mission the driving mnission can fx be changed from Acceleration to Skidpad which will change the car configuration. It is also possible to modify the current mission which fx could mean disabling a prevously enabled feature such as TV or PL.

The code for each can be seen in code snippet 5. A separate CAN message has been made for each of the two ways. The dashboard chooses which to use for any given situation.
The current mission is changed using the message with CAN ID offset of 0x052. This change is detected by the mission feature from section 15.3.10 and it handles loading in the new mission configuration.
The vehicle dynamics controllers are enabled/disabled directly using the message with CAN ID offset of 0x053.

```
ACTION msg_0x052:
  (* Change driving mission *)
  (* Convert the first byte of the message to USINT and compare it with the mission
     *)
  CASE BYTE_TO_USINT(ArCanReceive_0.ReceivedFrame.Data[0]) OF
    MISSION_NONE:          mission_index := MISSION_NONE;
    MISSION_ACCELERATION:  mission_index := MISSION_ACCELERATION;
    MISSION_SKIDPAD:       mission_index := MISSION_SKIDPAD;
    MISSION_SPRINT:        mission_index := MISSION_SPRINT;
    MISSION_ENDURANCE:     mission_index := MISSION_ENDURANCE;
    MISSION_TESTING:       mission_index := MISSION_TESTING;
    ELSE                   mission_index := MISSION_NONE;
  END_CASE;
END_ACTION

ACTION msg_0x053:
  (* Modify the current driving mission *)
  enable_tv := ArCanReceive_0.ReceivedFrame.Data[0].0; (* Torque Vectoring     *)
  enable_tc := ArCanReceive_0.ReceivedFrame.Data[0].1; (* Traction Control     *)
  enable_pl := ArCanReceive_0.ReceivedFrame.Data[0].2; (* Power Limiting       *)
  enable_tl := ArCanReceive_0.ReceivedFrame.Data[0].3; (* Torque Limiting      *)
  enable_lp := ArCanReceive_0.ReceivedFrame.Data[0].4; (* Launch Profile       *)
  enable_rb := ArCanReceive_0.ReceivedFrame.Data[0].5; (* Regenerative Braking *)
END_ACTION
```

Code snippet 5: The code implemented to change driving mission or modify the current driving mission.

Message 0x052 reads in the first byte of the message received and using a switch-case writes the new mission to the variable *mission_index*.
In message 0x053 the first byte contains a bit for the state of each controller. These are then read in separately using the dot-operator into the enable parameters for each of the controllers.

## 14.5 Communication

The standard CAN module described in section 23.3 is used without any changes for the CAN communication between the Master Controller and the Dashboard Node. The CAN port is setup in the following way.

| | |
|---|---|
| **Baud Rate** | 1 MBit/s |
| **CAN ID Size** | 11 bit |
| **Receive Queue Size** | 100 Messages |
| **CAN Port** | IF6.ST1.IF1* |

* For more information about port addressing see section 13.1.
For a list of the CAN messages use between the Master and Dashboard Node see appendix G.

# 15 Inverters

The Viking X has 4 AMK motor / inverter pairs that gives the car four-wheel drive. The motor-inverter pairs are bought and build to work together. The inverters are further more coupled together two-and-two. The two inverters controlling the front wheel motors are a pair and same goes for the rear wheel inverters.

## 15.1 Inverter State Machine

Figure 15 shows the inverter control state machine. The state machine consists of 7 states. Eah inverter has its own state machine, which means that each inverter can at any time be in a different state than the others. The reason for this is to make the system more robust and allowing for an inverter to have a warning or error and shutting itself down without shutting down the rest of the inverters. The system is then able to reset the inverter and get it back online while the car keeps driving.

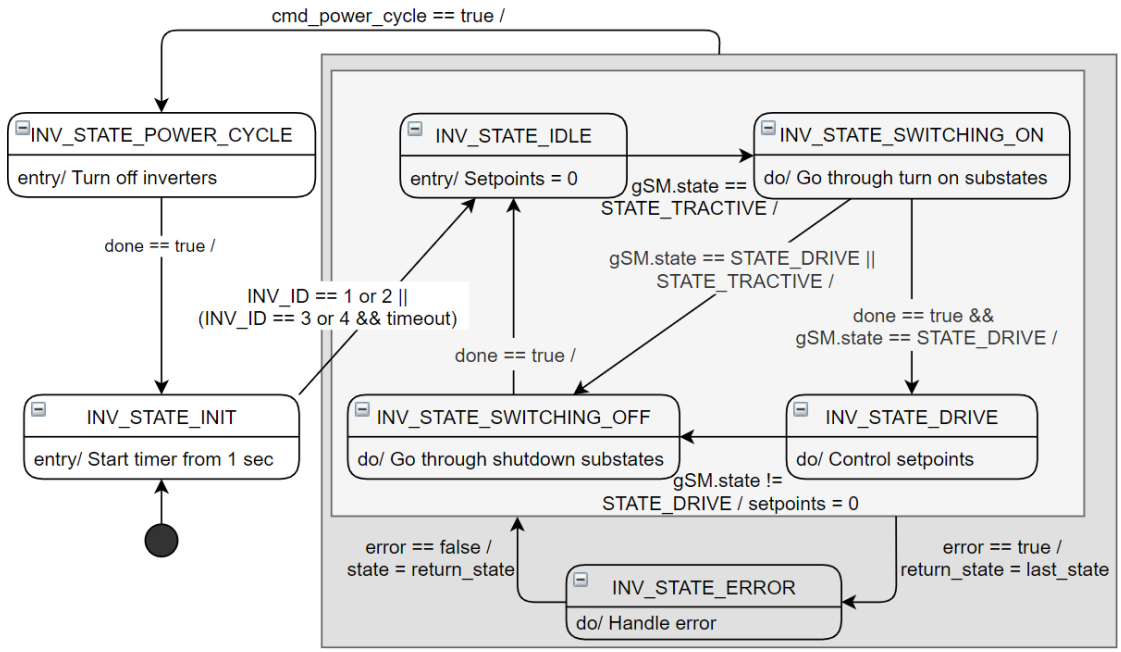


Figure 15: The state machine implemented for each of the inverters.

When the car is turned on all 4 inverters start in *INV_STATE_INIT* which is an initialization state in which the inverters are turned on. The inverters have a large in-rush current so a timer on 1 second is used to delay the turning on of inverters 3 & 4. When an inverter is turned on it transisions to *INV_STATE_IDLE* which is the inverter's safe state.
The inverter state machine is locked to the state of the global car state machine. The inverter state machine will stay in *INV_STATE_IDLE* until the global state machine enters *STATE_TRACTIVE* at which the inverters are switched on.
State *INV_STATE_SWITCHING_ON* consists of a set of substates that are used to switch on each inverter following the flowchart seen on figure 38 in appendix C. [**p.88**, 1] All actions on the left side of the flowchart in appendix C (except

the bottom action) are used for turning on an inverter. The bottom left action is the state in which the car is driving. When the car should be turned off it transistions to the right side and performs all the actions there to safely turn off the inverter. The left side of the flowchart has been packaged as substates inside state *INV_STATE_SWITCHING_ON* and the right side has been packaged as substates inside state *INV_STATE_SWITCHING_OFF*.
When the inverters are turned on the inverter state machine will wait for the global state machine to transistion to *STATE_DRIVE* by which the inverter state machine will transistion to its own drive state. In all other states than *INV_STATE_DRIVE* the motor setpoints are kept at 0 to make sure that they cannot run. When the inverters state machine enters state *INV_STATE_DRIVE* the control of the setpoints are given to the vehicle dynamics control algorithms discussed in section 15.3.
If an error happens to an inverter the current state will be saved and the state machine transistions to state *INV_STATE_ERROR* in which the inverter error handling is implemented. Removing an error is a matter of stepping through the flowchart given by AMK which can be found in appendix D and when the error disappers, *INV_STATE_ERROR* will then step back through the same steps and undo the changes. If an inverter gets an error it is removed and the inverter returns to the same state as it was in before to link up with the other inverters.
In some cases the error on an inverter is not possible to fix without turning off the inverter completely and this can also be necessary in other instances. To make the process easier a power cycle command can be send to an inverter and the inverter will power cycle itself using state *INV_STATE_POWER_CYCLE*. The power cycle command affect both inverters in the inverter pair.
When an inverter enters its *INV_STATE_ERROR* a timer is started. If this timer has a timeout that usually means that the automated error handling of
*INV_STATE_ERROR* is not able to fix the error. The motor pair is then both shutdown if the *pair shutdown feature* is enabled. Read more about the feature in section 15.3.9.
An inverter entering its *INV_STATE_ERROR* will report an alarm to the Alarm Management System. If an inverter has an error it will send an error code to the master controller through CAN and this error code will be part of the alarm message shown to the team. There is also alarms attached to inverter error timeout and inverter pair shutdown.

## 15.2 Communication

The inverter set 1 & 2 and 3 & 4 are each connected to their own CAN bus recommended in their datasheet. [1]
A version of generic CAN module described in section 23.3 is used for each of the buses.
The CAN ports are setup in the following way.

| | |
|---|---|
| **Baud Rate** | 1 MBit/s |
| **CAN ID Size** | 11 bit |
| **Receive Queue Size** | 100 Messages |
| **CAN Ports** | IF7 - Inverters 1 & 2<br>IF8 - Inverters 3 & 4* |

* For more information about port addressing see section 13.1.
For a list of the CAN messages use between the Master and the Inverters see appendix H.

### 15.2.1 Disconnection Detection

The generic CAN module is not able to detect if a signle device on a CAN bus with multiple devices stops communicating. Therefore an additional part is added.
Instead of having a single timer that is always reset every time a new message is received the module is equipped with a timer for each device connected to the bus. In this case that means two timers for each bus. Every time a new message is received it is looked up from which device the message came and the timer associated with that specific device is reset.
If a timeout occurs for one of the inverters an alarm is reported to the alarm management system telling what inverter has stopped communicating. Read more about the alarm management system in section 7.
Each inverter also have its own internal timer and if the inverter does not receive a new message for $50ms$ the inverter will shut itself down.

## 15.3 Vehicle Dynamics Control

The AMK motor-inverters can be be either torque or speed controlled. Speed control is mostly used for motion planning and is thus often used in advanced vehicle dynamics control algorithms. For the time being the motors are torque controlled because it is an easy starting point. This allows for the torque pedal to directly or through a gain control the motor torque setpoints.
Having four-wheel drive allows each wheel to be controlled individually which opens the door for more advanced control algorithms to helping the driver achieve better driving performance.
The torque setpoint is only allowed to the positive due to rule *EV2.2.3* stating that the wheels are not allowed to be spun in reverse. When the torque pedal is not pressed the torque output to all motors should be 0Nm which is states in rule *T11.8.12*.
One of the most influencial rules for vehicle dynamics control is rule *T11.8.14* which states that electronics control units are allowed to move torque around between motors but the total demanded torque may never exceed the demanded torque from the driver. It is only allowed to maintain or lower it. It means if the torque pedal is pushed to 50% the total combined torque requested by all 4 motors may not exceed 50% of their max torque. It means that if the torque pedal is completely lifted all electronics control units loose their ability to control the car. This is done to ensure safe control of the car.

**Future work**
The only control algorithm that can function when the torque pedal is lifted is electronic stabilization control (ESC) which uses individual wheel regenerative braking to maintain vehicle stability. ESC is not implemented in the current version of the car but it might come at a later stage.

### 15.3.1 Vehicle Dynamic Control Overview

**(New feature)**
The vehicle dynamics control of the SDU-Viking X car is still under development at the time of handing in this document so the specific implementation of each controller will not be shown. The following section will give an overview of which controllers that will be implemented and what they are each responsible for. Each controller is separately packaged into their own function block with unique interfaces. Please note that the interfaces and function block designs shown might change over time. For up to date information about the vehicle dyncamics controllers, please consult the teams Gitlab repository.

The overall control structure can be seen in figure 16. The vehicle consists of 6 parallel controllers that each have a specific purpose in controlling the car.

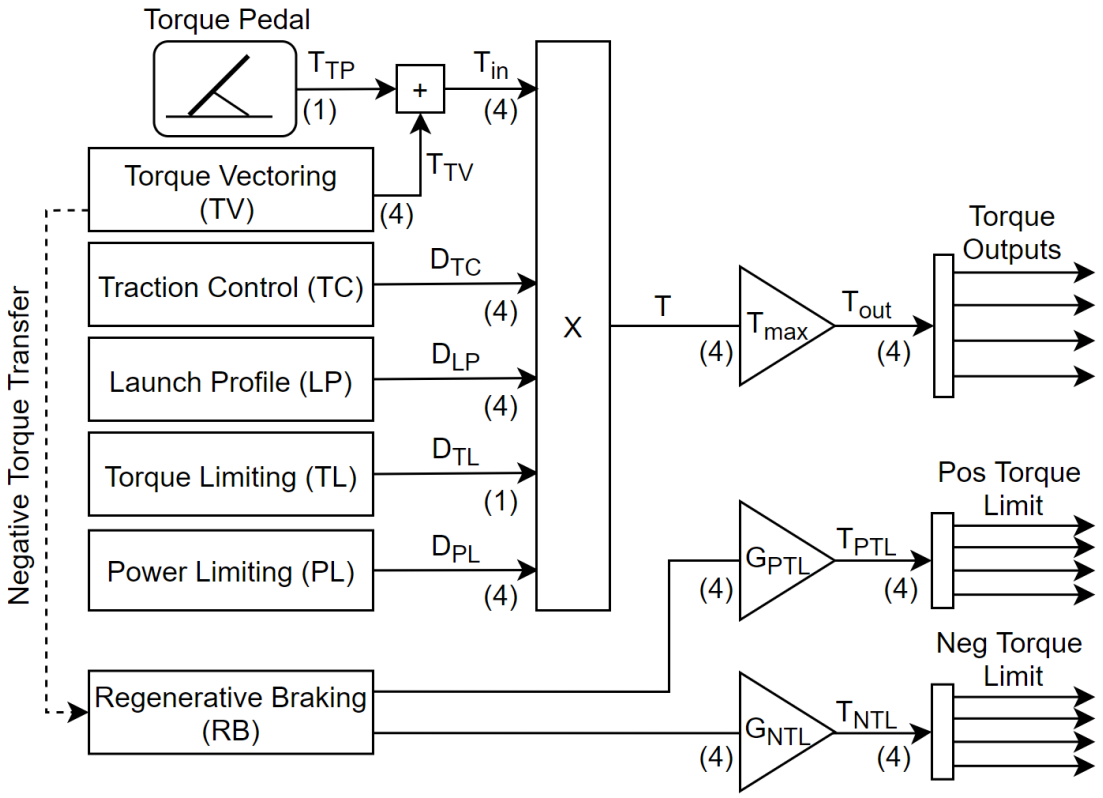


Figure 16: An overview of the vehicle dynamics control.

The controllers will each be explained in further detail in the following sections.

Each controller produces 4 outputs, one for each inverter. Traction Control (TC), Launch Profile (LP), Torque Limiting (LP) and Power Limiting (PL) are all implemented as scaling controllers that scale the torque output. Torque Vectoring (TV) is an additive/substractive controller which adds or removes torque on each motor. Lastly the Regenerative Braking (RB) controller controls the torque limits which tells the inverters how fast or slow it is allowed to change torque setpoints in its internal control loop. Having a slowly rate of change on the setpoint results in a low regenerative braking with a low regenerative current being fed back to the accumulator. Having a fast rate of change will result in higher regenerative braking and a high current being fed back.

The overall control follows equation 4. All the values in the equation are signals 4 values wide besides the $D_{TL}$ which is only 1 because it is the same for all wheels, to denote that that equation uses different variables for each individual motor they are subscripted with $i$ which is a placeholder for the specific motor, fx Front-Left (FL) or Rear-Right (RR).

$$T_i = T_{in,i} \cdot D_{TC,i} \cdot D_{LP,i} \cdot D_{TL} \cdot D_{PL,i} \quad , i \in \{FL, FR, RL, RR\} \tag{4}$$

Where $T_{in}$ is torque level coming from the torque pedal after being modified by the TV controller which is done with equation 5, and each of the $D$ parameters are torque derating output from the scaling controllers.

$$T_{in,i} = T_{TP} + T_{TV,i} \quad , i \in \{FL, FR, RL, RR\} \tag{5}$$

With $T_{TP}$ being the signal from the torque pedal which will be in interval $[0, 1]$ with 0 being no torque and 1 being full torque.
The derating outputs out of the 4 scaling controllers have to obey $D_i \in [0, 1] \quad , i \in \{TC, LP, TL, PL\}$ with 1 being the output if the controller is inactive.
The torque output $T_i$ lays in interval $[0, 1]$. The inverters require torque setpoints in Newton-Meter so $T$ is scaled up using $T_{max}$ to produce a torque setpoint in interval $[0, T_{max}]$.

$$T_{out,i} = T_i \cdot T_{max} \quad , i \in \{FL, FR, RL, RR\} \tag{6}$$

When the torque setpoints are found they are transmitted to each inverter through CAN.

### 15.3.2 Torque Vectoring (TV)

Torque vectoring is used to improve the vehicle's cornering performance by moving torque from the inner wheels to the outer wheels during cornering. It can massively improve the vehicle performance in dynamic events such as Skidpad where the time is roughly proportional to the maximum lateral force that the vehicle is able to maintain while cornering. The maximum lateral force can be improved with the used of TV.
A simple way to implemented a TV controller is to calculate the drivers desired yaw rate using the steering wheel angle and move motor torque from the inner wheels to the outer wheels. This type of controller is what the function block interface seen in

figure 17 is made for. The controller used is a generic PID controller as is described in section 23.1 where the D part is not used.

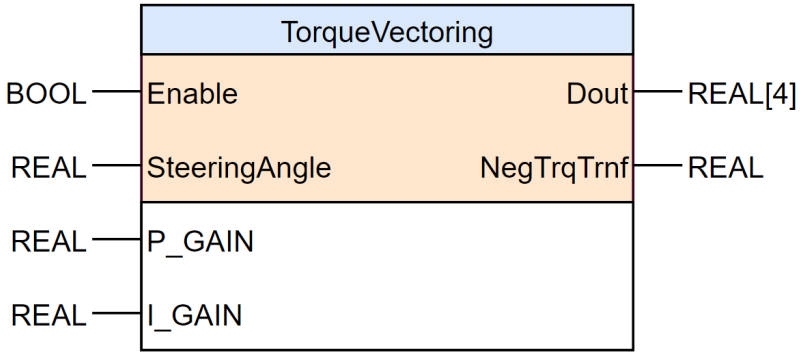


Figure 17: Overview of the Torque Vectoring function block.

TV is the controller mostly affected by rule *T11.8.14* because the torque pedal often is lifted when the car reaches a corner to slow it down by which the controller looses its ability to work.
If the torque vectoring controller is not able to move any torque either because the toruqe pedal is completely lifted, so no torque is available to be moved or because all torque has been moved away from the inner wheels, the torque vectoring controller is able to utilize what has been named *Negative Torque Transfer*. The negative torque transfer is essentially a braking signal send to the regenerative braking controller to tell it to start doing regenerative braking on the inner wheels proportional to the negative torque transfer. Braking on the inner wheels will further increase the vehicle yaw rate and move it closer to the desired yaw rate.
The $T_{TV}$ output is at all times in interval $[-T_{TP}, T_{TP}]$ which limits $T_{TV} \in [-1, 1]$.

### 15.3.3 Traction Control (TC)

The traction controller is responsible for making sure none of the wheels experience wheelspin when the car is accelerating. Wheel spin is undisirable because that does not propel the car forward. The traction control algorithm works to reduce the torque output if the longitudinal wheel slip exceeds the longitidunal slip maximum. The controller is only active when the actual wheels slip is above the maximum because that scenario requires a reduction of the output torque. It does not work in the other direction because that would mean an increase in output torque, making the car dangerous to drive and break rule *T11.8.14*.
The traction controller has two down sides. One being that the controller only becomes active when the wheels start slipping which means potential acceleration is already being lost and also the way wheel slip is calculated which does not work at low speeds see equation 7. Because of these two reasons a launch profile is also implemented which is described in section 15.3.4.

$$\sigma_{x,i} = \frac{R_\omega \omega_{w,i} - V_x}{R_\omega \omega_{w,i}} \quad , i \in \{FL, FR, RL, RR\} \tag{7}$$

Where $\sigma_{x,i}$ is the wheel longitudinal slip, $R_\omega$ is the wheel radius, $\omega_{w,i}$ is the wheel rotational velocity and $V_x$ is the vehicle longitudinal velocity.
When the vehicle is traveling at speeds close to 0km/h.

$$\lim_{V_x \to 0} \omega_{w,i} = 0 \tag{8}$$

which results in

$$\lim_{\omega_{w,i} \to 0} \sigma_{x,i} = \infty \tag{9}$$

Therefore at low speeds this equation do not work.
The TC function block can be seen in figure 18. The controller is a generic PID controller as is described in section 23.1 where the D part is not used.

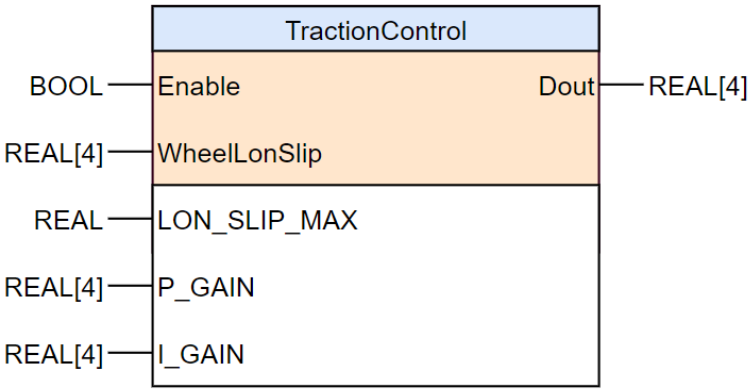


Figure 18: Overview of the Traction Control function block.

The function block outputs a derating signal in interval $[0, 1]$. It will be 1 when the algorithm is not active and reduce towards 0 depending on how much the traction controller needs to affect the output torque in order to achieve optimal longitudinal wheel slip.

### 15.3.4 Launch Profile (LP)

Due to the short comings of the traction controller a launch profile is needed. The launch profile has one main purpose which is to replace the traction controller at low speeds where actual the longitudinal wheel slip cannot be calculated. Depending on the efficiency of the launch profile it might also completely replace the traction controller for the Acceleration event but that is left for the future to decide.
The launch profile is a simple controller controlling the torque output depending on the vehicle velocity. The function block interface shown in figure 19 is made for a simple version of a launch profile that linearly derates the torque depending on the vehicle velocity as can be seen in figure 20. The launch profile can then be configured with the *OFFSET* and *SLOPE* inputs.

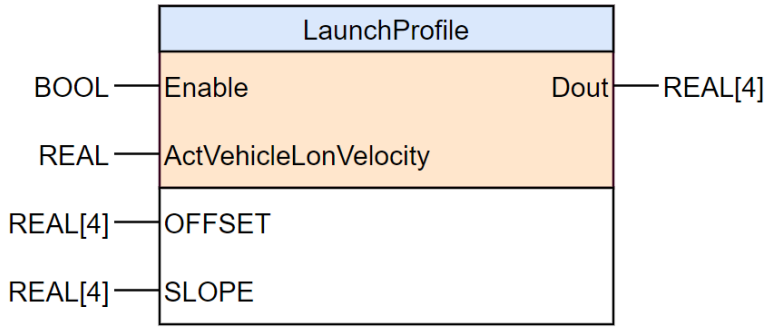


Figure 19: Overview of the Launch Profile function block.

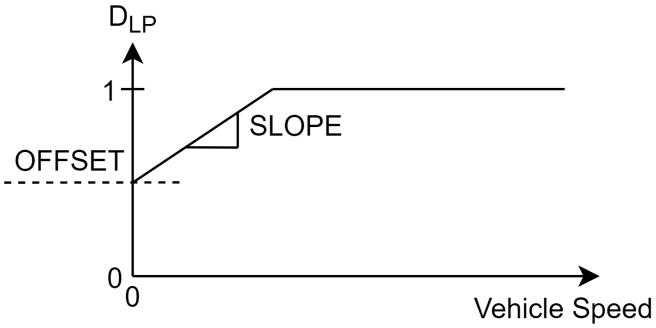


Figure 20: A simple version of the launch profile.

The function block outputs a derating signal in interval $[OFFSET, 1]$. It will have the value $OFFSET$ when the car is at standstill and when it starts moving the output will increase until 1 is reached at which the algorithm will switch to inactive. The $OFFSET$ should be in interval $[0, 1]$.

### 15.3.5 Power Limiting (PL)

The power limiting controller is implemented to limit the power consumption of the vehicle. Rule *EV2.2.1* states that a maximum of 80kW is allowed to be drawn from the accumulator. It is allowed to draw more for a short period because the measurement equipment used at the competition measures in terms of a moving average of 500ms which is specified in rule *D9.4.1*. So the power is allowed to be excited if it is equally lowered afterwards to not exceed 80kW after the moving average filter is applied.
The PL controller works simular to the TC controller in which it will only start being active when the power consumption comes near the maximum power allowed. When it activates it slowly starts derating the output torque which in turn should reduce the power consumption.
The function block can be seen in figure 21. The controller is used a generic PID controller as is described in section 23.1 where the D part is not used.

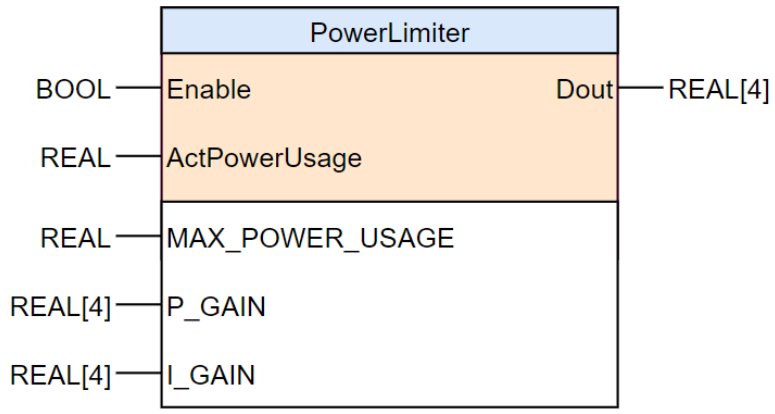


Figure 21: Overview of the Power Limiter function block.

The function block outputs a derating signal in interval $[0, 1]$. It will be 1 when the algorithm is not active and reduce towards 0 the higher the power consumption becomes.

### 15.3.6 Torque Limiting

**(New feature)**

The possibility of limiting the motor torque is implemented to make it possible for the car to be driven at a lower torque levels. This can be useful for tests where the full torque is not needed, or to extend the driving distance by not allowing / tempting the driver to utilize the full torque. Lowering the available torque can also make the car easier to drive which is useful for less experienced drivers.
The feature is implemented by scaling down the available torque. From the liveview a drop-down menu can be chosen where the torque limit can be chosen in steps of 20% going from 20% to 100%. It is possible to control the derating with higher precision but to make it more user friendly is has been limited to these steps.
The function block used can be seen in figure 22. It has an enable input. Setting this to *False* will clamp the output, *Dout*, to 1, if the enable is set to *True*, equation 10 is used to find *Dout*.

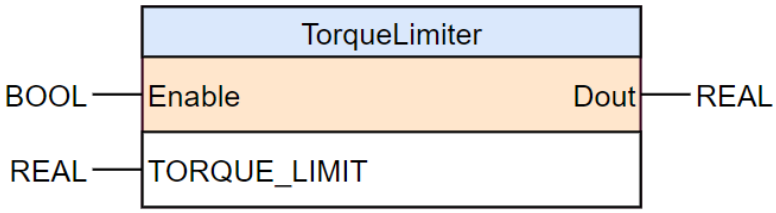


Figure 22: Overview of the Torque Limiter function block.

$$D_{out} = \frac{TORQUE\_LIMIT}{100} \tag{10}$$

Where $TORQUE\_LIMIT$ is a percentage in interval $[0, 100]$, where 0 will result in no output torque and 100 results in full output torque. *Dout* is in interval $[0, 1]$.

### 15.3.7 Regenerative Braking

The regenerative braking functionality is made to have multiple functions in the system. The main functionality is using the electric motors for braking and thereby using them as generators that convert the vehicle's kinetic energy to electric energy that can recharge the battery which is permitted and unrestricted as per rule *EV2.2.2*. The regenerative braking is also used to simulate motor braking seen in combustion engines as well as allowing the torque vectoring algorithm to not only reduce and increase torque on a wheel but also brake on wheels resulting in what is practically a negative torque.

There are some safety aspects to take into account when working with regenerative braking. The accumulator is made of Lithium-Polymer battery cells and they have a tendency to explode or catch fire if care is not taken. During regenerative braking it is important to make sure the cells are not being recharged harder than their maximum rating. It is also important to not try and recharge if the accumulator is fully charged.
The regenerative brake controller takes in the negative torque transfer produced by the TV controller and adds it to the current controller output.
The positive torque limit determine how much the motors can accelerate and this output should be at maximum at all times. It is not part of the regenerative braking algorithm but for convinience it has been assigned to the same controller so both inverter torque limits are managed by the same controller.
An overview of the current regenerative braking function block can be seen in figure 23.

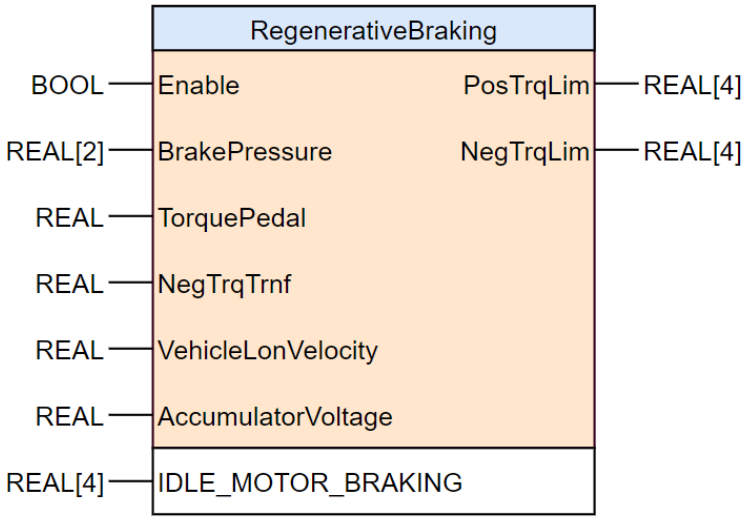


Figure 23: Overview of the Regenerative Braking function block.

Where the positive torque limit $PosTrqLim \in [0, 1]$ and the negative torque limit $NegTrqLim \in [0, 1]$. They are converted to the actual values needed by the inverters by multiplying them with constants $C_{PTL}$ and $C_{NTL}$.

**Future work**
Future versions of the regenerative braking controller could also involve closed loop control of the regenerative braking current instead of the open loop control suggested in this section.

### 15.3.8 Configuration During Drive

**(New feature)**

The liveview functionality makes it possible to configure the car's driving setpoints and configuration parameters at any time. This also includes when the car is driving which can be dangerous if care is not taken. Therefore all configuration of the car is disabled when the car is put into *STATE_DRIVE*. This is done by greying out all input boxes in the liveview and AND'ing all command inputs with the *enable_configuration_during_drive* variable. This does not prevent users from changing values directly through Automation Studio because this cannot be disabled but it prevents it from being done through liveview.
For testing purposes the feature can be disabled allowing for the parameters to be changed when the car is in *STATE_DRIVE* but not moving. Which can significantly improve the tune process of vehicle dynamics controllers.

**Warning**
Changing configuration parameters while the car is in its driving mode can be of great danger to the driver, team members, spectators, the car and other property, so used this feature with caution. The feature should only ever be used during testing by authorized personal and should always be disabled when attending official events.

### 15.3.9 Pair Shutdown

To improve the car's reliability and avoid an error on one inverter from potentially stopping the car right in front of the finish line the feature of pair shutdown has been implemented. The inverters are grouped into two pairs, the front wheels and the rear wheels. If an error occurs on any of the inverters and the automated error handling is not able to fix the error, the pair in which the inverter with the error is located is shuts down by removing the torque output. The feature will half the performance and make the car either front or rear wheel driven but if that is what saves the endurance race then that might be worth it.
The feature can be enabled or disabled and should only be enabled for the endurance event. If the feature activates an alarm will be registered to the alarm management system.

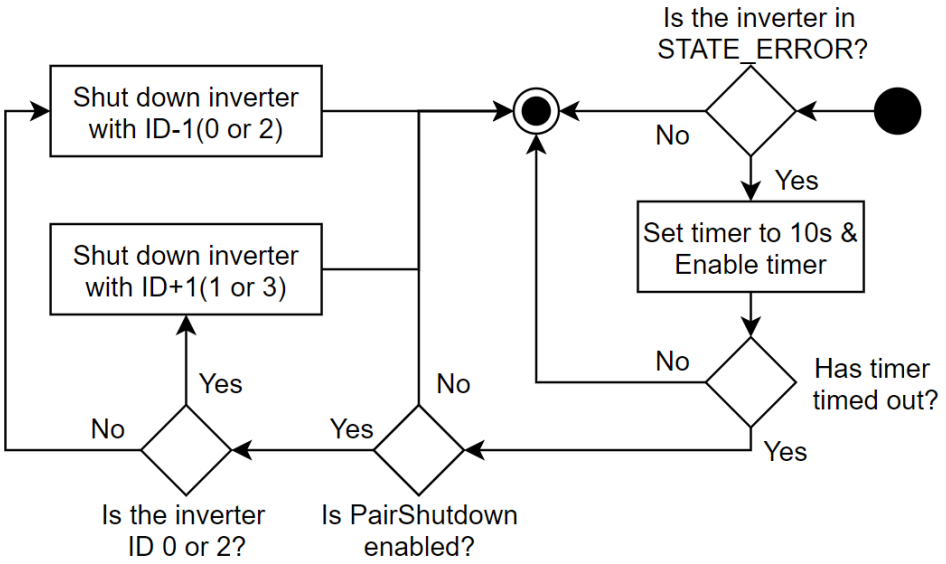


Figure 24: The flowchart of the pair shutdown feature. The flowchart is performed once per cycle per inverter.

```
(* Disable error timer if the inverter exists error state *)
FOR i := 0 TO INVERTER_COUNT DO
  (* enable timer if inverter is in STATE_ERROR and set timeout duration *)
  TON_error_timer[i](IN := (inverter[i].state = STATE_ERROR), PT := T#10s);
  (* If timer runs out trigger errorTimeout *)
  inverter[i].error_timeout    := TON_error_timer[i].Q;
END_FOR

(* If the inverters are allowed to shut each other down parse through each pair
    and check if it should shut its partner down *)
IF enable_pair_shutdown THEN
  (* Front *)
  inverter[1].shutdown_by_partner := inverter[0].error_timeout;
  inverter[0].shutdown_by_partner := inverter[1].error_timeout;
  (* Rear *)
  inverter[3].shutdown_by_partner := inverter[2].error_timeout;
  inverter[2].shutdown_by_partner := inverter[3].error_timeout;
ELSE
  (* Else let all inverters run for themselves *)
  FOR i := 0 TO INVERTER_COUNT DO
    inverter[i].shutdown_by_partner := FALSE;
  END_FOR;
END_IF;
```

Code snippet 6: Code to handle the pair shutdown feature

The timer used is the TON timer described in section 23.2.

**Warning**
Enable only the pair shutdown feature after careful consideration. If in doubt disable it.

### 15.3.10 Race modes / Driving missions

**(New feature)**

To improve the configuration of the car depending on the type of driving that the vehicle is about to do the concept of driving missions is used. A mission is defined as a dynamic event and the following missions are implemented.

- Acceleration
- Skidpad
- Sprint
- Endurance
- Testing

Each mission might have a range of specific parameter value configurations that result in the best performance of the car in that specific event. An example could be enabling the use of a launch profile during the Acceleration event but disabling it for all other events or reducing the available torque with the torque limiter for Endurance to make sure the driver is keeping better control over the car.
To avoid having to configure the car every time it is turned on and reduce the risk of inserting wrong values or losing a mission configuration the driving mission feature is implemented directly in the vehicle software.
To save and load driving mission configurations the software library called *MpRecipe* made by B&R Automation is used. The library provides a function block called *MpRecipeXml* which handles the saving and loading actions to xml-files located in internal memory. A saved set of parameters is called a *recipe* because it is intented to be used for production machines where the parameters could specify the machine configuration to produce a specific product and different products require difference machine configurations.

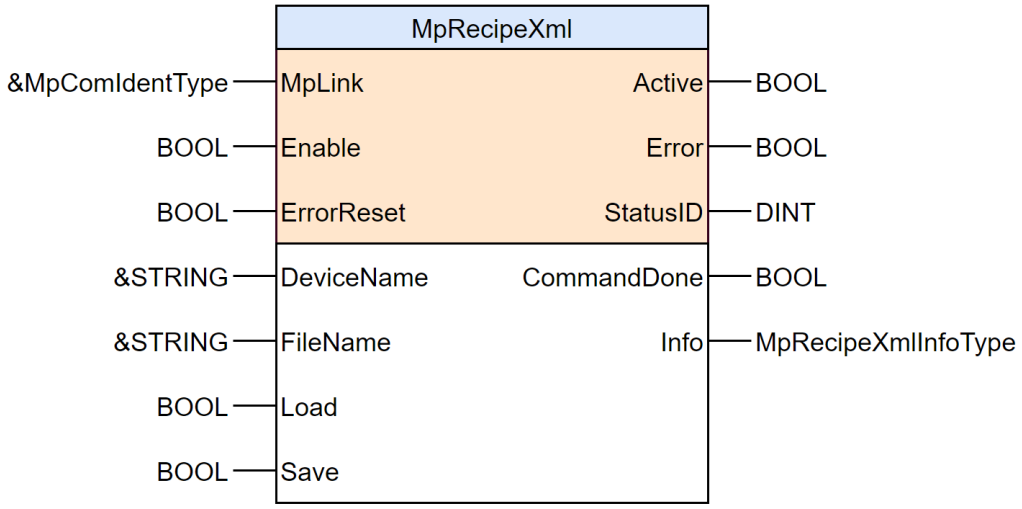


Figure 25: Overview of the *MpRecipeXml* function block.

The block links to the overall library through the *MpLink* interface. It links to the file device to where the xml files should be saved to and loaded from through the *DeviceName* interface and the file name is specified with *FileName*. When these things are setup and the block is enabled it is ready to be used.
The *Load* and *Save* functionalities can be used but does not do anything yet because no parameters has been assigned to be saved and loaded. For this the function block *MpRecipeRegPar* is used. It is used to register parameters to the current recipe. One *MpRecipeRegPar* function block is required for each parameter that should be part of the recipe.

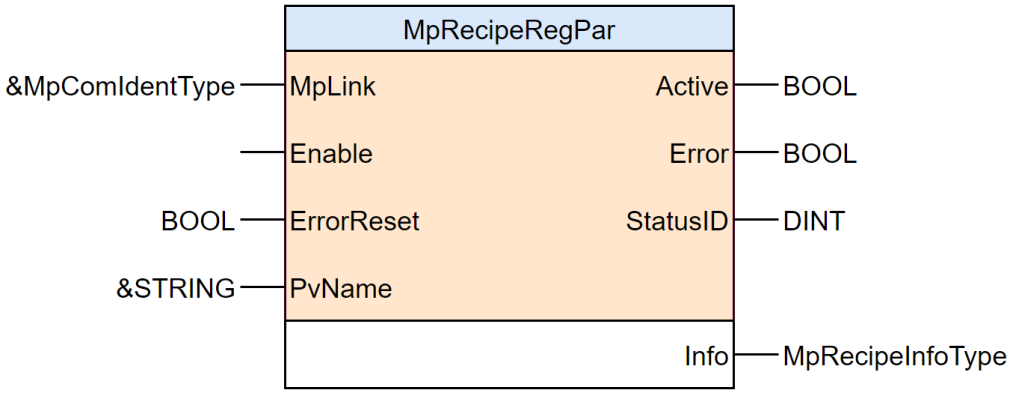


Figure 26: Overview of the *MpRecipeRegPar* function block.

The *MpRecipeRegPar* block is also linked to the overall library through its *MpLink*. It is linked to a specific parameter through its *PVName* interface. Parameters can have any type and can even be structures of many parameters.
The connection between the *MpRecipeXml* function block and the list of *MpRecipeRegPar* function blocks can be seen below.

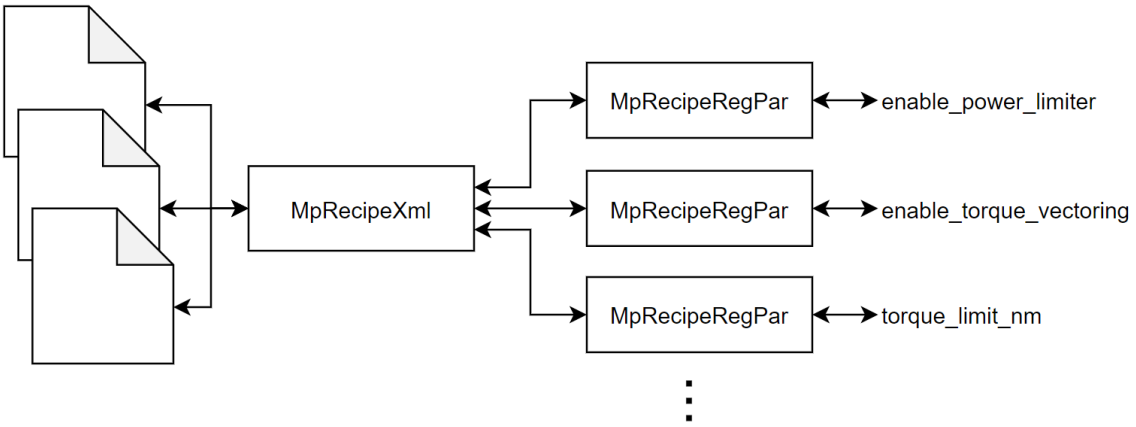


Figure 27: The principle behind the use of the *MpRecipe* library.

The parameters that are being saved as of this documents hand in can be seen below.

| Parameter | Type | Description |
|---|---|---|
| Power Limit [KW] | REAL | The power limit in $kW$s. |
| Torque Limit [Nm] | REAL | The torque limit in $Nm$. |
| Enable Torque Vectoring | BOOL | Status whether torque vectoring is turned on or off. Read more in section 15.3.2. |
| Enable Traction Control | BOOL | Status whether traction control is turned on or off. Read more in section 15.3.3. |
| Enable Launch Profile | BOOL | Status whether launch profile is turned on or off. Read more in section 15.3.4. |
| Enable Power Limiter | BOOL | Status whether power limiting is turned on or off. Read more in section 15.3.5. |
| Torque Limiting Percent | REAL | Torque Limiter percentage. Read more about the torque limiter in section 15.3.6 |
| Enable Regenerative Braking | BOOL | Status whether regenerative braking is turned on or off. Read more in section 15.3.7. |
| Enable Configuration During Drive | BOOL | Status whether configuration during drive is turned on or off. Read more in seciton 15.3.8 |

Table 7: List of the parameters which are saved for each mission.

The actions that can be envoked are a save of the current parameters to the currently selected mission and loading of another mission or resetting potential changes to the current mission. Which can be seen on figure 28.

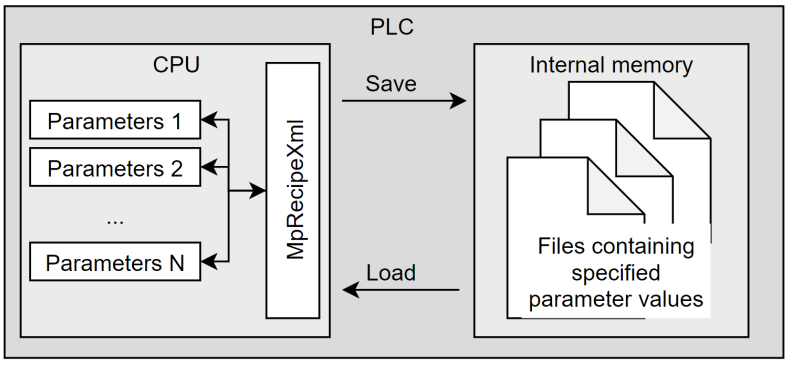


Figure 28: Principle behind saving and loading a mission.

The *save action* can only be envoked by a user pressing a *Save* button on the live-view. This will overwrite the selected mission with the values currently configured in the vehicle.
The *load action* can be envoked in two ways. If the mission parameter is changed the load action is called to load the configuration of the mission being changed to. It can also be called by pressing the *Load* button on the liveview. This will load the current mission values.

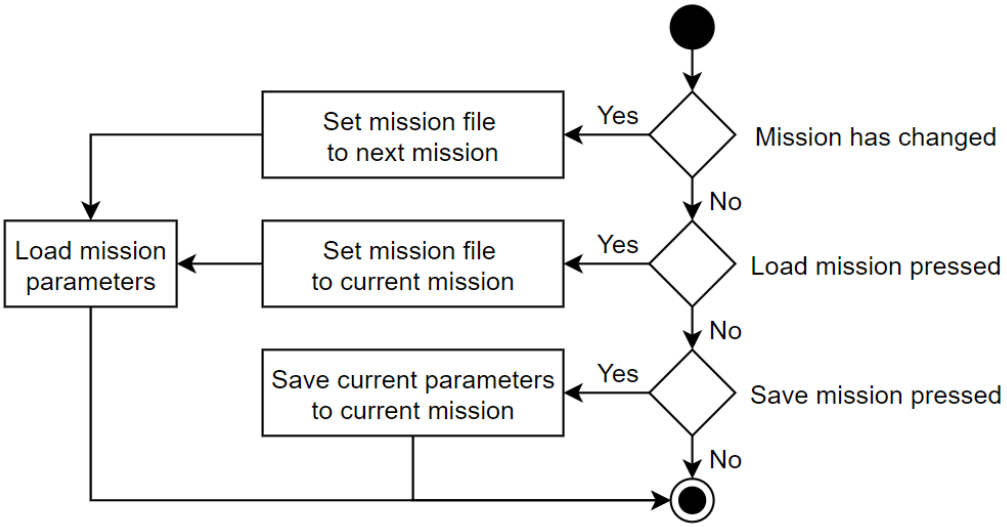


Figure 29: A flowchart showing the driving mission feature.

### 15.3.11 Acceleration Measurements

**(New feature)**

To make it easier to test vehicle performance without having to set up measurement equipment a timer designed to measure the vehicle acceleration is implemented. The timer is meant to give rough performance measurement. The timer block used has a resolution of $10ms$ and the task handling the timer has a cycle time of $10ms$ which gives the acceleration measurement a worst case response time of $20ms$. How the feature works can be seen on figure 30.

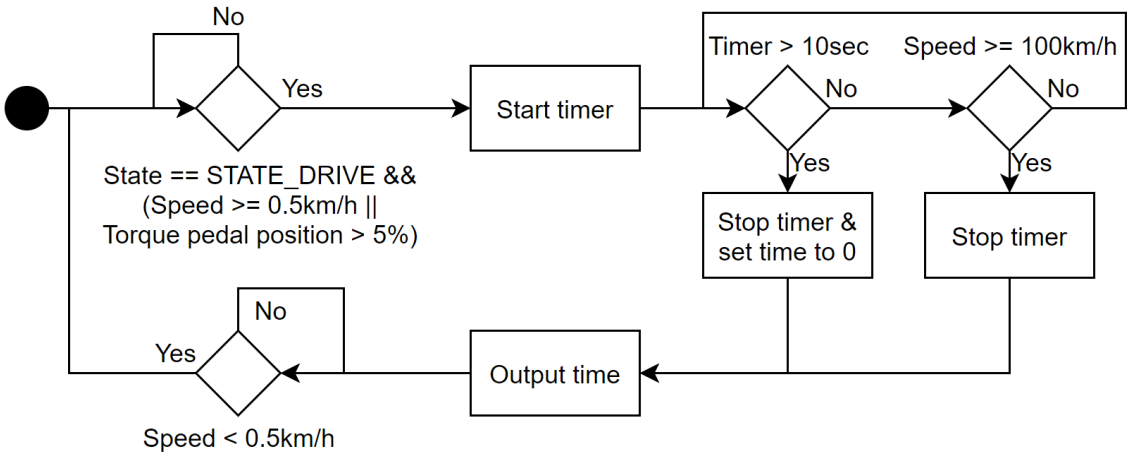


Figure 30: Flowchart over the acceleration measurement.

While the vehicle is not in state drive and the vehicles speed is below $0.5km/h$ or the torque pedal is not being pressed the function will loop around in the upper left corner. Once the boolean expression is true a timer is started. The timer will then run till the vehicle either surpasses $100km/h$ or till it has run for more than $10sec$ at which the function will timeout. In both cases the timer will be stopped and if a

timeout happened the elapsed time in the timer will be overwritten with $0s$ to avoid confusion. The elapsed time is then outputtet. The function will then wait for the vehicle to return to a hold by which it will reset and get ready to run again.

### 15.3.12 Range Estimation

**(New feature)**

To get a rough estimate about how long the vehicle can continue to drive the *range estimation* feature is implemented. The algorithm draws a straight line one a state-of-charge vs distance-traveled coordinate system between the initial values and the current values and then extends the line to estimate the maximum range. The estimated range is the x-coordinate where the line intersection with the distance-traveled axis.
Range estimation is an action managed by the task INV_SM. The action is called every cycle but has a timer set to 1 sec so the content is only executed once per second.
The concept can be seen on figure 31 where the two graphs on the left shows the potential state of charge over distance.

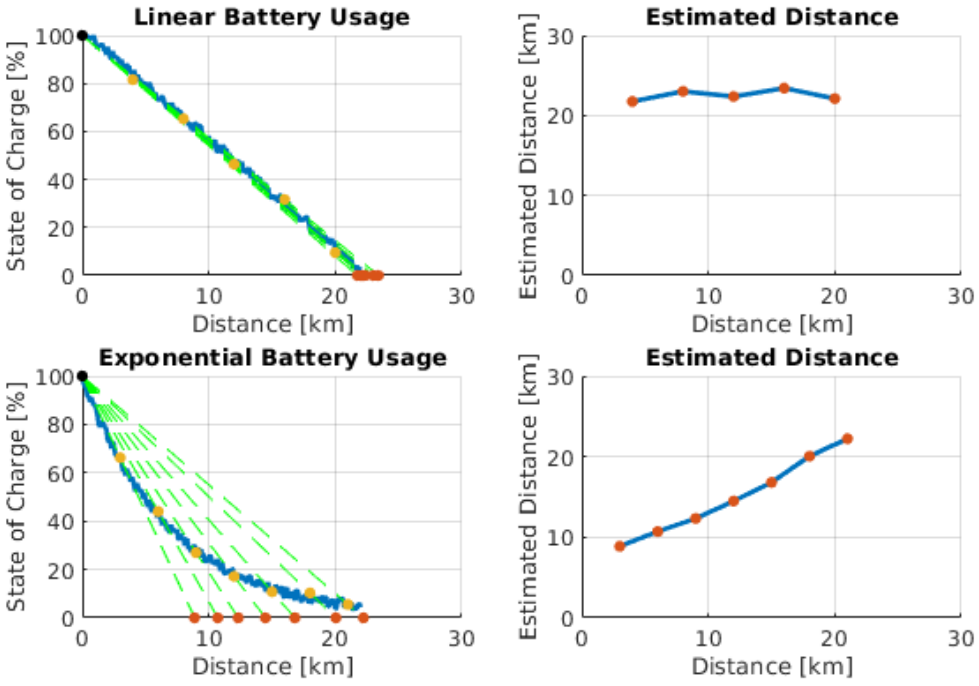


Figure 31: Examples of the way the range estimation feature works. Note that the values in the graphs are **not** measurements or simulations but generated arbitrarily to show the discussed scenario.

The blue line on the left graphs are the state of charge, the yellow points are points where the algorithm is run. The algorithm then estimates the red dots which are then shown over distance on the right plots. The distance estimation works most

consistently when the state of charge decreases relatively linearly which can be seen by the estimate being more constant over distance on the top graph.
The estimated range is calculated with equation 11.

$$d_{estimation} = d_{start} - \frac{c_{start} \cdot (d_{current} - d_{start})}{c_{current} - c_{start}} \tag{11}$$

Where $d_{estimation}$ is the estimated distance, $d_{start}$ is the starting distance which for the most cases will be $0km$, $c_{start}$ is the start state-of-charge, $d_{current}$ is the distance traveled at the time where the algorithm is run and $c_{current}$ is the state-of-charge at the time where the algorithm is run.
The implemented code for the range estimation feature can be seen in code snippet 7.

```
ACTION rangeEstimation:
  (* Handle timer *)
  TON_rangEstTimer.IN := TRUE;
  TON_rangEstTimer();

  (* If timeout occurs *)
  IF TON_rangEstTimer.Q THEN
    (* Reset Timer *)
    TON_rangEstTimer(IN := FALSE);

    (* Calculate next value in range estimation *)
    MTFilterMovingAverage_0(In := distStart - (socStart*(distActual-distStart))/(
        socActual-socStart));
    distEstimated := MovingAverage.Out;
  END_IF;
END_ACTION
```

Code snippet 7: The implementation of the range estimation feature.

The action has a timer called *TON_rangEstTimer* of type TON (read more about the TON timer in section 23.2) and this timer is used to only execute the algorithm once per second. When the timer has a timeout it is reset and equation 11 is calculated. The result is fed into a function block of type *MTFilterMovingAverage* which performs a moving average filter on the result. It is done to make the estimated range vary less over time. The filter width has been set to 10 values.

# 16 Accumulator Management System (AMS)

*The specific implementation of the AMS handling will change shortly after the making of this document, so for up to date information please consult the code repository on Gitlab. The version described here is the version as of document hand-in.*
The AMS task has two responsibilities. Send control signals to the AMS and perform information processing of the data coming from and AMS about the current state of the battery.
The AMS is made by SDU-Vikings.

## 16.1 Control

The AMS requires a set of control commands all of which can be seen in table 8. The commands are transmitted continuously depending on the current state of the master controller. The commands are held high as long as they should be active. Read more about the vehicle state machine in section 8.

| Command Name | Description |
|---|---|
| cmd_idle | Tells the AMS to go to a safe state which is when an error occurs or the car should be in STATE_IDLE. |
| cmd_sc_enable | Tells the AMS to enable the last element of the shutdown circuit. |
| cmd_start_precharge | Tells the AMS to start precharging. |
| cmd_start_drive | Tells the AMS to allow current to be drawn from the battery. |

Table 8: List of AMS commands.

The code for controlling the commands can be seen in code snippet 8. The global state has been renamed to *gSM.state* in the following example to make the code more readable. In the actual code this variable is called *gStateMachine.status.state*.

```
(* Calculate the IDle command *)
cmd\_idle          :=  gSM.state = STATE_IDLE OR gSM.state = STATE_ERROR;

(* Calculate the SC Enable command *)
cmd\_sc\_enable    :=  gSM.state = STATE_ENABLING_SC OR gSM.state = STATE_SC_ENABLED
                      OR gSM.state = STATE_PRECHARGING OR gSM.state = STATE_TRACTIVE
                      OR gSM.state = STATE_DRIVE;

(* Calculate the Precharge command *)
cmd\_precharge     :=  gSM.state = STATE_PRECHARGING OR gSM.state = STATE_TRACTIVE
                      OR gSM.state = STATE_DRIVE;

(* Calculate the Drive command *)
cmd\_start\_drive  :=  gSM.state = STATE_DRIVE;
```

Code snippet 8: Code to handle AMS command signals.

## 16.2 Monitoring

The AMS sends back its current state which can be used to know if the transmitted commands have been received and succesfully performed.

The battery consists of 5 banks each with 28 cells and 10 temperature sensors. The AMS monitors all of these and transmits them all over CAN to the master controller to be visualized on the liveview to give the team an overview of the current battery status. It can also be used for diagnostics purposes in case of errors.
All control of the battery such a voltage or tempererature leaving its allowed range is handled by the AMS and not the master.
The AMS transmits all cell voltages and temperatures but not in one long chunk of data because that large amount of data would fill up the CAN bus for a relatively long period of time which in worst case can compromise the vehicle's real-time performance. Instead the voltages and temperatures are packaged into smaller messages containing either 8 voltages or 4 temperatures that are then spaced out over time to limit the CAN bus load. The cycle time for transmitting all messages has been set to 1 second so all values are refreshed with 1Hz. This solution comes with the downside of the voltages and temperatures not being synced and instantaneous changes will not look instantaneous.
To be able to pack the messages more efficiently each voltage is send as an 8 bit value with 2 decimal point precision starting from 2V. That means every voltage needs to be run through equation 12 in order to convert the received value to the actual cell voltage.

$$V_{cell} = \begin{cases} 0 & V_{received} = 0 \\ 2 + \left(\frac{V_{received}}{100}\right) & V_{received} > 0 \end{cases} \quad [V] \tag{12}$$

The method means that voltages under 2V cannot be shown which for the most part should be fine because cells are not able to be below 2V without taking damage. To show defective cells and disconnected cells it has by convention been decided that if $V_{received} == 0$ then $V_{cell}$ is shown as $0V$.
For a better overview of the battery health the maximum, average and minimum cell voltages and temperature are continuously found and displayed to the team via the liveview.
The AMS is in charge of measuring the status of the *IMD*, *AMS* and *HVDC Interlock* which all are part of the shutdown circuit discussed in section 10.
The AMS has a set of error signals that it can sent over to the master controller which will be reported to the alarm management system discussed in section 7.
The liveview pages made to show the status of the AMS can be seen in appendix I.5.

## 16.3 Communication

The standard CAN module described in section 23.3 is used without any changes for the CAN communication between the Master Controller and the AMS Node.
The CAN port is setup in the following way.

| | |
|---|---|
| **Baud Rate** | 1 MBit/s |
| **CAN ID Size** | 11 bit |
| **Receive Queue Size** | 100 Messages |
| **CAN Port** | IF9* |

* For more information about port addressing see section 13.1.

# 17 Shunt

**(New feature)**
The shunt is of the type IVT-S[9] by Isabellenhuette and is a component monitoring the battery voltage, current and actual power consumption.
The power consumption is a crucial part of the power limiting control discussed in section 15.3.5 and to be able to comply with the strict timing requirements the data is not allowed to experience too much delay. Therefore the shunt is connected to both the master controller and the AMS. The AMS handles the general configuration of the shunt and also uses its data while the master controller only sniffs packages coming from the shunt to use the data. Two alarms in the Alarm Management System have been bound to the value of the power consumption. One of them is a warning triggered when the power consumption reach 90% of the allowed maximum power and one is an alarm triggered when the power exceeds the maximum power. The data does not need any special data handling besides being scaled to the right units which for the most part includes being multiplied by 10.

## 17.1 Communication

The standard CAN module described in section 23.3 is used without any changes for the CAN communication between the Master Controller and the Shunt.
The CAN port is setup in the following way.

| | |
|---|---|
| **Baud Rate** | 1 MBit/s |
| **CAN ID Size** | 11 bit |
| **Receive Queue Size** | 100 Messages |
| **CAN Port** | IF9* |

* For more information about port addressing see section 13.1.
For a list of the CAN messages use between the Master and Shunt see appendix F.

# 18 Sensor Network

**(New feature)**
The sensor network is a soon to come project for the car which contains a collection of small sensor nodes connected to the master controller through a shared CAN bus. The network has not been fully developed and adopted by the time of this documents hand-in which is why some sensors meantioned in this section are also directly measured by the master controller.
*The sensor network is a semester project done in the spring semester of 2020 and thus the master interface will be updated shortly after hand-in. For updated information please consult the master controller code repository on Gitlab. Of this same reason only a short overview of the possible features will be given.*

## 18.1 Data handling

Sensor measurements done by a sensor node are converted to the correct units before being sent over CAN to the master controller. Data is send as 32 bit floats and therefore do not need any further data processing before it can be used by tasks on the master controller.
Each node has to be setup when the car is turned on which is done through a set of CAN messages to each node. When a node is fully setup it will start cyclically transmitting its measured and converted data to the master controller. Each node has a fixed number of sensors and each sensor has a polynomial associated with it to convert the measured value to a correct unit. When the car is turned on the polynomial coefficients need to be received by the sensor node before that particular sensor is setup.
The coefficients are all placed on the master controller to make the network dynamic and easily configured even during runtime.
The planned sensors for the sensor network are the following.

- Torque pedal sensors
- Brake pressure sensors
- Steering wheel angle sensor
- Suspension travel sensors
- Temperature sensors

The current version of the master controller still measure the torque pedal position, brake pressure and steering wheel angle directly which is why these modules are described in section 9.1, 9.2 and 9.3.

## 18.2 Communication

The used for the sensor network will be a combination of asynchronous and synchronous communication. When the car is turned on a lot of asynchronous messages needs to be transmitted to each sensor network node. After the initial setup the

master controller do not need to send any more messages unless runtime changes are performed by the team.
The standard CAN communication module described in section 23.3 is used with a slight modification of the TX module. It does not use a timer to transmit messages but instead cyclically checks if any modifications have been made to the variables related to the polynomial coefficients. If a change occurs an asynchronous message is sent off to the node to update it.

The CAN port is setup in the following way.

| | |
|---|---|
| **Baud Rate** | 1 MBit/s |
| **CAN ID Size** | 11 bit |
| **Receive Queue Size** | 100 Messages |
| **CAN Port** | IF6.ST1.IF2* |

* For more information about port addressing see section 13.1.
The specific CAN messages used between the Master and Sensor Network are not specified by the time of hand-in.

# 19 Inertial Nagivation System (INS)

The Inertial Nagivation System (INS)[12] is a device that combines GPS, 3-axis accelerometer, 3-axis gyro, 3-axis magnetometer and barometric pressure sensor together in one package. Due to the university closing the sensor interface has not been implemented. Thus only its expected implementation and future usage will be explained in this section.
*For up-to-date information about the implementation please consult the Master Controller Gitlab repository.*

## 19.1 Usage

The sensor will mainly be used for accurate longitidunal and lateral velocity and acceleration measurements as well as yaw rate measurements used by the vehicle dynamics controllers. The information will also be viewable through the liveview and the longitudinal speed will also be displayed on the dashboard.

## 19.2 Communication

The communication is done via RS-232 which means that on the master controller a library called *DVFrame* should be used. This library contains the following relevant function blocks.

- *FRM_xopen*: Used to configure and enable the RS-232 port.
- *FRM_read*: Used to receive information from the receive buffer.
- *FRM_write*: Used to write information to the send buffer.

# 20 FS Datalogger

At the competition a datalogger is inserted into the car to measure the power consumed by the motors to ensure rule *EV2.2.1* of maximum power consumption is obeyed. Due to the event cancellations of the summer of 2020 the FS Datalogger CAN protocol described in rule *DE7.3.5* has not been released by the time of handing in this document. Therefore the interface has not been fully implemented.

## 20.1 Data handling

The specific data and data format are not known. The data and format sent through CAN in 2019 is shown in table 9.

| **Parameter** | **Value** |
|---|---|
| CAN ID | 0x430 |
| Direction | FS Datalogger → Master |
| Transmission Rate | Periodically: Cycle time unknown. |
| Message Length (DLL) | 64bit / 8byte |

| **Variable** | **Start [bit]** | **Length [bit]** | **Unit** | **Type** | **Description** |
|---|---|---|---|---|---|
| MSG_CNTR | 0 | 8 | - | uint8 | Message number |
| STATUS | 8 | 8 | - | uint8 | Data logger status |
| POWER | 32 | 16 | $3 \cdot W$ | sint16 | Actual Power |
| VOLTAGE | 48 | 16 | $0.04 \cdot V$ | sint16 | Actual Voltage |
| CURRENT | 48 | 16 | $0.05 \cdot A$ | sint16 | Actual Current |

Table 9: The data send by the FS Datalogger in 2019 through CAN.

## 20.2 Communication

The receive part of the generic CAN module is used because the datalogger is cyclically transmitting data. Read more about the generic CAN module in section 23.3. Rule *DE7.3.4* states that the CAN bus operates with a baud rate of 1 Mbit/s. The CAN port is setup in the following way.

| | |
|---|---|
| **Baud Rate** | 1 MBit/s |
| **CAN ID Size** | 11 bit |
| **Receive Queue Size** | 100 Messages |
| **CAN Port** | IF6.ST1.IF3* |

# 21 Liveview - Mapp View

The main purpose of the liveview functionality is to let the team monitor the car live from the sidelines of the track. The B&R Automation framework provides a visualization feature called *Mapp View* that can be used to create and configure liveview visualizations. A visualization can both display live data but also send commands back to the controller which provides the possibility of tuning or configuring the system parameters at runtime without reprogramming the PLC.

*Mapp View* is a drag and drop style editor made to create custom visualization pages. The visualization pages can then be accessed during runtime through the browser if the PLC is connected via Ethernet cable or WiFi. The visualization itself is hosted by the PLC but handled during idle time. Idle time is defined as the time between the handling of processes which means the hosting of the visualization does not affect the other tasks running on the PLC. If the PLC is running at high CPU loads the visualization will get slow and unresponsive but not affect the real-time performance of the car. The majority of the visualization is client based so beyond the initial loading of each page there is not much work associated with hosting the visualization.

To transfer parameter values to the visualization the communication protocol called *OPC-UA* is used which is the defaulted protocol used by B&R. The specific details about the protocol is not relevant for the master controller or SDU-Vikings. OPC-UA can simply be seen as a variable hosting service offered by the master controller to the liveview clients. The communication is configured via the file "OpcUaMap.uad" found in Automation Studio under "Configuration View > Connectivity > OpcUA". In this file each parameter that should be available for showing on the liveview is enabled and next time the project is transferred to the target the variable will be accessable. For more information about *Mapp View* and OPC-UA see section 11 in the Automation Studio Starter Guide in appendix J.

To limit the amount of work needed for the PLC only a specific amount of clients are allowed to connected to the visualization at any one point in time. To allow the liveview PC to show more than one page of the visualization and potentially allow for additional members to also access the visualization at the same time, the maximum number of clients have been set to 5.
In appendix I all the different liveview pages can be seen.

# 22 Datalogging

The datalogging is handled by the task *DATALOG*. Every time the task runs a new value of each variable is save to an external file. The task runs with 10Hz giving a resolution of 100ms.
The datalogging structure used is one developed by B&R Automation Denmark and it has not been made specifically for SDU-Vikings. It has a range of features to make it easy to plug into a new project and get to work easily but because of its wide range of uses it has a high complexity. Therefore a simplified state machine has been shown below and instead of state names, the states contain a short description

of what it does. Note that some of the states are actually multiple states in the real implementation but has been simplified to give a better overview. The simplified state machine can be seen in figure 32.

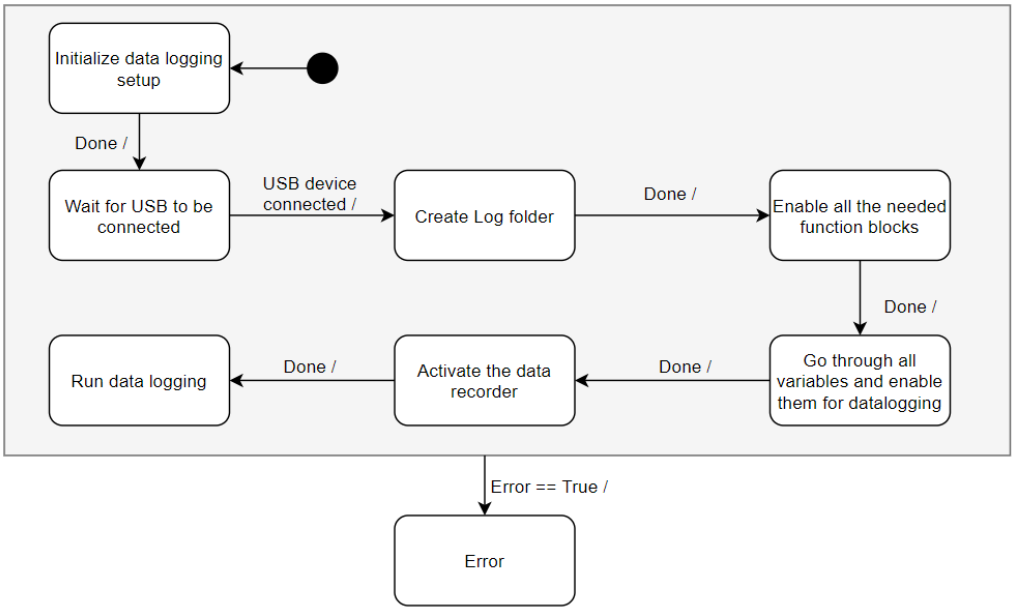


Figure 32: Simplified state machine for the datalogger task.

When the controller is turned on it first runs the initialization of the data logging setup. In this state it registers all the variables that should be datalogged. In Automation Studio this is done in file *Datalog/initSetup.st* and each variable that should be datalogged is written into the array *gDataLogging.data.csvData[]*. The array has been made 300 variables long but it can be extended if needed.
When the initialization is over the state machine will continue to the wait state. In this state the controller will wait for an USB to be connected. If a USB is already connected it will only stay in this state for a single cycle. The data logging task is made in a way that a USB can be connected to any USB port on the device and it will start data logging there so the state cycles through all USB ports and the first port where it finds a USB it will start the datalogging. On the X90 controller there is only one USB port so this cyclic checking is not strictly needed.
When a USB is detected the task creates a new folder for the datalogging with the current date. When this is done all the function blocks used throughout are enabled. The configuration of these function blocks where done in the initialization phase. When all blocks are enabled and running correctly each variable is registered in the function block *MpDataRegPar*. The *MpDataRegPar* function block works very simular to the *MpRecipeRegPar* described in section 15.3.10 but *MpDataRegPar* is used together with *MpDataRecorder* which is the function block that does the actual data logging. When all variables have been registered the *MpDataRecorder* function block is activated and the data logging starts.
The task has per default no error handling implemented so if an error occurs it will go to the error state and stop the data logging. The state disabled all the function blocks and wait for either a manual reset or a system reboot.

# 23 Generic modules

Some modules are used many times throughout the project and therefore generic modules are made or included. In the following section the build in PID controller and timer are explained and the custom SDU-Viking CAN base module is described.

## 23.1 PID Module (*MTBasicsPID*)

The following section contains a description of the PID function block called *MTBasicsPID*.

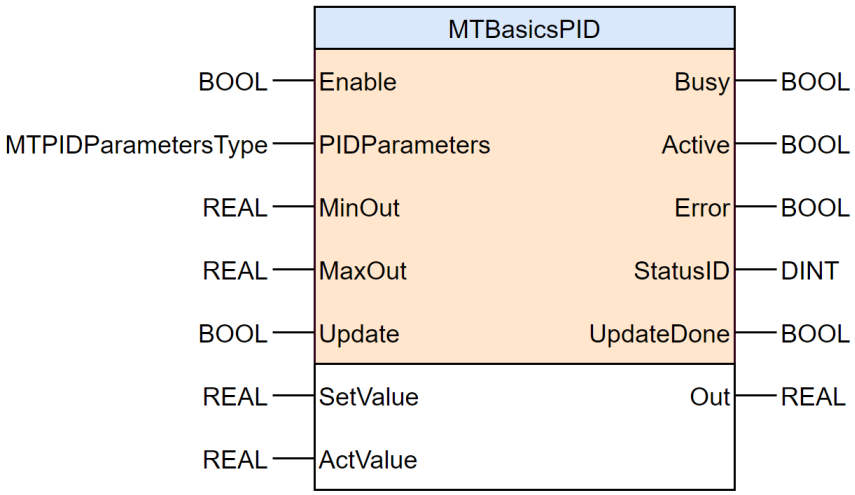


Figure 33: Overview of the *MTBasicsPID* function block.

*Enable* turns on the function block, *PIDParameters* are discussed below, *MinOut* is a saturation point at which the output cannot go lower, *MaxOut* is a saturation point at which the output cannot go higher, *Update* updates the input parameters to the block at the rising edge, *SetValue* is the setpoint for the block and *ActValue* is the system feedback. The output *Out* is the control output of the block.
The PID works based on the following equations.

$$e(t) = \text{SetValue}(t) - \text{ActValue}(t) \tag{13}$$

$$P(t) = K_p e(t) \tag{14}$$

$$I(t) = \frac{K_p}{T_i} \int e(t)dt \tag{15}$$

$$D(t) = K_p T_d \frac{d}{dt} e(t) \tag{16}$$

$$\text{Out}(t) = P(t) + I(t) + D(t) \Rightarrow \tag{17}$$

$$\text{Out}(t) = K_p e(t) + \frac{K_p}{T_i} \int e(t)dt + K_p T_d \frac{d}{dt} e(t) \tag{18}$$

Where $K_p$ is the gain, $T_i$ is the integration time and $T_d$ is the derivative time. These are setup through the *MTPIDParametersType*. If the normal PID control structure on the form seen in equation 19 is used to tune a PID controller it can be converted with equations 20-22.

$$u(t) = K'_p e(t) + K_i \int e(t)dt + K_d \frac{d}{dt} e(t) \tag{19}$$

The gains can be converted in the following way

$$K_p = K'_p \tag{20}$$

$$T_i = \frac{K_p}{K_i} \tag{21}$$

$$T_d = \frac{K_d}{K_p} \tag{22}$$

## 23.2 Timer (*TON*)

The TON timer function block is in simple terms a switch on a delay. The function block can be seen on figure 34.

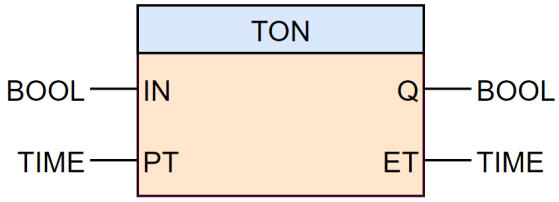


Figure 34: Overview of the TON timer function block

The block has 2 inputs. *IN* which enables the function block and *PT* which describes how long the timer should run. When the timer runs out the output *Q* goes high and while the timer is running the elapsed time can be monitored via the output *ET*.
If *IN* is FALSE then *Q* is FALSE and the *ET* output is 0. When *IN* is set to TRUE then *ET* will start to count up until $ET = PT$ by which *ET* will stop counting up. *Q* is TRUE if *IN* is TRUE and $ET = PT$. Otherwise *Q* will be FALSE.
The timer has a resolution of $10ms$ and cannot be used to monitor anything below that.

## 23.3 CAN Module

The Master Controller communicates with a lot of other modules through CAN and the handling of the CAN communication is done almost in the same way for all modules. The generic module is described here and if a task has modified it, why and how can be found in the section about that module.
The CAN module operated on top of the state machine shown on figure 35.

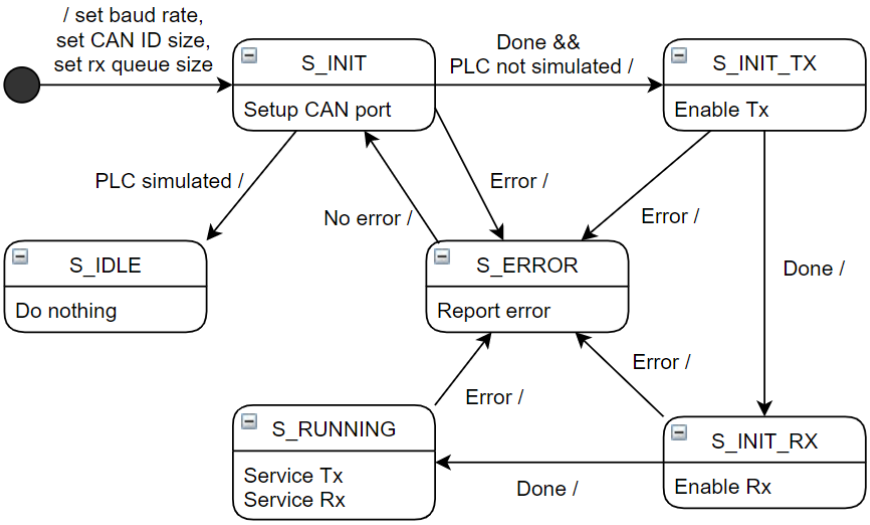


Figure 35: State machine for the CAN task.

When the PLC boots up it will enter the initialization state, *S_INIT*. If the PLC is simulated on a PC and no CAN ports are present the task will change to *S_IDLE*, where the task is dormant. If this is not the case, the initialization state sets up the CAN communication which involves, baudrate, CAN ID size, receive queue size and CAN port.
If the port initialization is successful the state machine initializes the CAN transmit function block. When this is successful the CAN receive function block is enabled. If both function blocks are enabled and active the state machine will go to the state where the CAN port is serviced cyclically and the state will remain here unless an error occurs.
In the running state the transmit action and receive action are called cyclically with the frequency at which the CAN task is running.
The flow of events for the transmit action can be seen on figure 36.

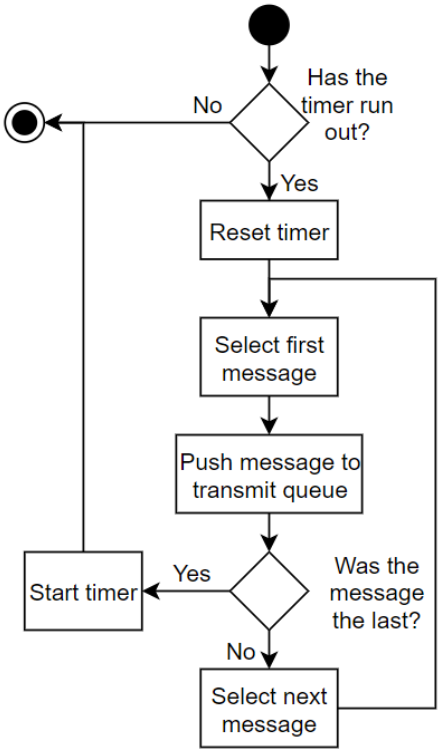


Figure 36: Flowchart showing the flow of events performed cyclically to transmit CAN messages.

The action has an internal timer which is used to keep track of when to transmit the messages which are all send one after another in a block of messages. Using a timer makes it possible to transmit slower than the frequency at which the task is executed.
If the timer runs out the action has a list of CAN messages that should be send, and it will cycle through all of them, insert the current PLC data in a predefined way and add each of them to the transmit queue. When all messages has been pushed to the queue the timer will be reset and the action ends.
The receive action can be seen on figure 37.

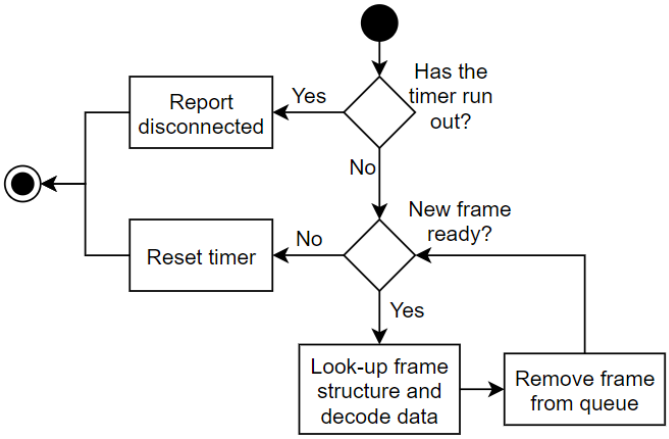


Figure 37: Flowchart showing the flow of events performed to receive CAN messages.

The receive action also has a build in timer but this is used to keep track of how much time has passed between received messages. If more time elapses than is allowed, the CAN port is assumed disconnected. This simple form of disconnection detection has two limitations. It requires the other module communicating to cyclically transmit and it does not work for purely event based communication. It also only works for CAN connections where only one module besides the master controller is communicating. If multiple modules are using the same port one module could be communicating while the other is not and it would go by undetected.
See 15.2.1 for a disconnection detection version that works with multiple devices on the same CAN bus.
All messages being received is put into a recevie queue. The receive action goes through all CAN messages in the queue and extracts the data from them and writes them their corresponding variables. When this is done the timeout timer is restarted.

### 23.3.1 Message encoding

All messages are encoded using the same two building blocks. One for encoding individual bits and one for encoding one or more bytes.
For encoding single bits the dot-operator is used. The dot operator allows a single bit in a variable to be accessed. Boolean variables take up a whole byte in memory but only the 0th bit is used. When this value is 0 the boolean value is *FALSE* and when it is 1 the value is *TRUE*. Therefore when the value needs to be mapped to a single bit in a message it is the value of the 0th bit that should be mapped. An example of the dot-operator can be seen in the top of code snippet 9 under *Bit encoding*.
If the variable that should be encoded into the message is instead multiple bytes fx if a 16bit/2bytes integer should be send another method is used. The method is to copy the memory containing the variable into the message using the *memcpy()* function. To do this the memory address of the first byte of the destination is needed, the address of the first byte of the source is needed and lastly the number of bytes to copy is needed. The memory address of a variable can be found with the *ADR()* function. An example of the usage of the *memcpy()* function can be seen below in listing 9 under *Byte encoding*.
Direct memory copying is possible because data in the PLC is saved in little-endian format and the data going into a CAN message should also be in little-endian format. If there had been a mismatch, the data could not be directly copied.

```
(***** Bit encoding *****)
(* 'boolean_variable1' is written to the 1st bit of byte 0 of the message. *)
ArCanSend_0.Frame.Data[0].0 := boolean_variable1;

(* 'boolean_variable2' is written to the 2nd bit of byte 0 of the message. *)
ArCanSend_0.Frame.Data[0].1 := boolean_variable2;

(***** Byte encoding *****)
(* Two bytes of 'integer_variable1' is written to bytes 1-2 of the message. *)
memcpy(ADR(ArCanSend_0.Frame.Data[1]), ADR(integer_variable1), 2);

(* Two bytes of 'integer_variable2' is written to bytes 3-4 of the message. *)
memcpy(ADR(ArCanSend_0.Frame.Data[3]), ADR(integer_variable2), 2);
```

Code snippet 9: CAN message encoding example.

### 23.3.2 Message decoding

To decode a message the same two methods as for encoding is used but the sources and destination are just switched around. Examples of the decoding process can be seen below in listing 10.

```
(***** Bit decoding *****)
(* The 1st bit of byte 0 of the message is written to 'boolean_variable1'. *)
boolean_variable1 := ArCanSend_0.Frame.Data[0].0;

(* The 2nd bit of byte 0 of the message is written to 'boolean_variable2'. *)
boolean_variable2 := ArCanSend_0.Frame.Data[0].1;


(***** Byte decoding *****)
(* Bytes 1-2 of the messag is written to 'integer_variable1'. *)
memcpy(ADR(integer_variable1), ADR(ArCanSend_0.Frame.Data[1]), 2);

(* Bytes 3-4 of the messag is written to 'integer_variable2'. *)
memcpy(ADR(integer_variable2), ADR(ArCanSend_0.Frame.Data[3]), 2);
```

Code snippet 10: CAN message decoding example.

### 23.3.3 CAN ID's

All devices in the car that communicates through CAN has more than 1 message that they either transmit or receive. To make it easier to maintain, debug and to avoid unnecessary work all messages from the same device are placed in the same CAN ID range. Fx could a range be *0x300-0x380*. Instead of hardcoding in all message ID's they are instead made relative to a base-address. This makes is possible to move the range to another location in CAN ID space without having to recalculate all message ID's. The range *0x300-0x380* would therefore instead be defined as *BASE_ADDRESS = 0x300, MAX_OFFSET = 0x080*.
This would make addresses *0x305* and *0x310* to *BASE_ADDRESS + 5* and *BASE_ADDRESS + 10*.

SDU

# Part V
# Abbreviations

| Abbreviation | Description |
|---|---|
| AIR | Accumulator Isolation Relay |
| AMS | Accumulator Management System |
| BOTS | Brake Over-Travel Switch |
| CAN | Controller Area Network |
| DLL | Data Link Layer |
| DV | Driverless Vehicle |
| EBS | Emergency Brake System |
| FL | Front-Left |
| FMEA | Failure Mode and Effects Analysis |
| FR | Front-Right |
| FS | Formula Student |
| FSG | Formula Student Germany |
| FSUK | Formula Student United Kingdom |
| GPS | Global Positioning System |
| HVD | High Voltage Disconnect |
| HVDC | High Voltage DC |
| IMD | Insulation Monitoring Device |
| INS | Inerital Navigation System |
| INV | Inverter |
| I/O | Inputs and Outputs |
| LCD | Liquid Crystal Display |
| LED | Light Emitting Diode |
| LP | Launch Profile |
| LV | Low Voltage |
| LVMS | Low Voltage Master Switch |
| PL | Power Limiting |
| PLC | Programmable Logic Cirsuit |
| PV | Parameter Value |
| RB | Regenerative Braking |
| RDY | Ready |
| RL | Rear-Left |
| RR | Rear-Right |
| RTDS | Ready-To-Drive-Sound |
| RX | Receive |
| SC | Shutdown Circuit |
| SM | State Machine |
| SOC | State-Of-Charge |
| TC | Traction Control |
| TL | Torque Limiting |
| TS | Tractive System |
| TSMS | Tractive System Master Switch |
| TV | Torque Vectoring |
| TX | Transmit |

Table 10: Abbreviations

# Part VI
# Bibliography


[1] AMK. *AMK RACING KIT 4 wheel drive "Formula Student Electric"*.

[2] BR Automation. *X90CP174.48-00*. URL: https://www.br-automation.com/da/produkter/plc-systems/x90-mobile-control-system/x90-mobile-control/x90cp17448-00/ (visited on 05/16/2020).

[3] BR Automation. *X90DI110.10-00*. URL: https://www.br-automation.com/da/produkter/plc-systems/x90-mobile-control-system/x90-option-boards/digital-inputs/x90di11010-00/ (visited on 05/16/2020).

[4] BR Automation. *X90IF720.04-00*. URL: https://www.br-automation.com/da/produkter/plc-systems/x90-mobile-control-system/x90-option-boards/communication-modules/x90if72004-00/ (visited on 05/16/2020).

[5] BR Automation. *X90PO210.08-00*. URL: https://www.br-automation.com/da/produkter/plc-systems/x90-mobile-control-system/x90-option-boards/digital-outputs/x90po21008-00/ (visited on 05/16/2020).

[6] Formula Student Germany. *Formula Student Germany Electric Inspection Sheet 2019*.

[7] Formula Student Germany. *Formula Student Rules 2020 V1.0*. URL: https://www.formulastudent.de/fileadmin/user_upload/all/2020/rules/FS-Rules_2020_V1.0.pdf (visited on 05/16/2020).

[8] Formula Student Germany. *FSG Competition Handbook 2020 V1.0*. URL: https://www.formulastudent.de/fileadmin/user_upload/all/2020/rules/FSG20_Competition_Handbook_v1.0.pdf (visited on 05/16/2020).

[9] *IVT-S, HIGH PRECISION CURRENT MEASUREMENT*. Version Datasheet, Version 1.01. Isabellenhuette.

[10] SDU-Vikings. *FS 2019 FMEA - SDU-Vikings - Car E86*.

[11] SDU-Vikings. *FS 2020 FMEA - SDU-Vikings - Car E33*.

[12] VECTORNAV Embedded Navigation Solutions. *VN-200 GPS-Aided INS*. URL: https://www.vectornav.com/products/vn-200 (visited on 05/16/2020).

[13] *TM213 Automation Runtime*. Version TM213TRE.461-ENG. BR Automation.

[14] Formula Student UK. *Formula Student UK 2020 Supplementary Rules 2020*. URL: https://www.imeche.org/docs/default-source/1-oscar/formula-student/fs2020/rules/formula-student-uk-2020-supplementary-rules---final.pdf?sfvrsn=2 (visited on 05/16/2020).

# Part VII
# Appendix

## A Rules

This appendix contains a compiled list of all the rules relevant to the master controller.

| Rule | Description |
|---|---|
| EV2.2.1 | The TS power at the outlet of the TS accumulator container must not exceed 80 kW. |
| EV2.2.2 | Regenerating energy is allowed and unrestricted . |
| EV2.2.3 | Wheels must not be spun in reverse. |
| EV2.3.2 | The commanded motor torque must remain at 0Nm until the APPS signals less than 5pedal travel and 0Nm desired motor torque, regardless of whether the brakes are still actuated or not. |
| EV4.10.9 | A green indicator light in the cockpit that is easily visible even in bright sunlight and clearly marked with “TS off” must light up if the TS is deactivated, see EV4.10.3. |
| EV4.11.1 | The driver must be able to activate and deactivate the TS, see EV4.10.2 and EV4.10.3, from within the cockpit without the assistance of any other person. |
| EV4.11.6 | After the TS has been activated, additional actions must be required by the driver to set the vehicle to ready-to-drive mode (e.g. pressing a dedicated start button). The transition to ready-to-drive mode must only be possible during the actuation of the mechanical brakes and a simultaneous dedicated additional action. |
| EV4.11.7 | The ready-to-drive mode must be left immediately when the shutdown circuit is opened. |
| EV4.12.1 | The vehicle must make a characteristic sound, continuously for at least one second and a maximum of three seconds when it enters ready-to-drive mode. |
| EV5.8.8 | red indicator light in the cockpit that is easily visible from inside and outside the cockpit even in bright sunlight and clearly marked with the lettering “AMS” must light up if the AMS opens the shutdown circuit. It must stay illuminated until the error state has been manually reset, see EV6.1.6. Signals controlling this indicator are SCS, see T11.9. |
| EV6.1 | Shutdown Circuit Section |
| EV6.3.7 | A red indicator light in the cockpit that is easily visible from inside and outside the cockpit even in bright sunlight and clearly marked with the lettering “IMD” must light up if the IMD opens the shutdown circuit. It must stay illuminated until the error state has been manually reset, see EV6.1.6. Signals controlling this indicator are SCS, see T11.9. |
| D9.4.1 | A violation is defined as using more than the maximum power, see EV2.2, or exceeding the specified voltage, see EV4.1.1, after a moving average over 500 ms is applied to the respective data logger signal, see EV4.6. |
| DE7.3.4 | The CAN bus interface operates on a data rate of 1 Mbit7s and is not terminated internally. |
| DE7.3.5 | The CAN message layout can be obtained from a dbc-file provided on the competition website. |
| DV3.2.7 | A red indicator light in the cockpit that is easily visible even in bright sunlight and clearly marked with the lettering “EBS” must light up if the EBS detects a failure. |
| T6.3.1 | The vehicle must be equipped with one brake light that is illuminated if and only if -the hydraulic brake system is actuated or the electric brake system is actuated, see EV2.2.2 |
| T11.8.3 | Pedal travel is defined as percentage of travel from fully released position to a fully applied position where 0% is fully released and 100% is fully applied. |
| T11.8.6 | If analog sensors are used, they must have different, non-intersecting transfer functions, . A short circuit between the signal lines must always result in an implausibility according to T11.8.9. |
| T11.8.7 | The APPS signals are SCSs, see T11.9. |

| | |
|---|---|
| T11.8.8 | If an implausibility occurs between the values of the APPSs and persists for more than 100 ms. - The power to the motor(s) must be immediately shut down completely. It is not necessary to completely deactivate the tractive system, the motor controller(s) shutting down the power to the motor(s) is sufficient |
| T11.8.9 | Implausibility is defined as a deviation of more than ten percentage points pedal travel between any of the used APPSs or any failure according to T11.9. |
| T11.8.12 | A fully released accelerator pedal must result in: A wheel torque of 0Nm. |
| T11.8.14 | Any algorithm or electronic control unit that can manipulate the APPS signal, for example for vehicle dynamic functions such as traction control, may only lower the total driver requested torque and must never increase torque unless it is exceeded during a gearshift. Thus the drive torque which is requested by the driver may never be exceeded. |
| T11.9 | System Critical Signals (SCSs) Section |
| T11.9.2 | Any of the following SCS single failures must result in a safe state of all connected systems: (a) Failures of signals transmitted by cable: - Open circuit - Short circuit to ground (b) Failures of analog sensor signals transmitted by cable: - Short circuit to supply voltage (c) Failures of sensor signals used in programmable devices: - Implausibility due to out of range signals, e.g. mechanically impossible angle of an angle sensor. (d) Failures of digitally transmitted signals by cable or wireless: - Data corruption (e.g. checked by a checksum) - Loss and delay of messages (e.g. checked by transmission time outs) Signals might be a member of multiple signal classes, e.g. analog signals transmitted by cable might be a member of T11.9.2.a, T11.9.2.b and T11.9.2.c. If a signal failure is correctable, e.g. due to redundancy or worst case values, the safe state must be entered as soon as an additional non correctable failure occurs. |
| T11.9.5 | Indicators according to T11.9.1 with safe state "illuminated" (e.g. absence of failures is not actively indicated) must be illuminated for 1 s to 3 s for visible check after power cycling the LVMS. |

Table 11: Rules used in the development of the Master Controller.

| **Steps** | **Description** |
|---|---|
| ▷ | Set vehicle to ready to drive state. Press accelerator pedal $> 25\%$. Push brake pedal |
| 274 | Motors stop turning |
| ▷ | Release brake, while accelerator pedal still activated |
| 275 | Motors do not turn |
| ▷ | Release accelerator pedal slowly |
| 276 | Motors turn again when APPS position is $< 5\%$ |
| ▷ | Get motors turning, disconnect $\geq 50\%$ of APPS while motors turn. |
| 277 | Motors stop turning |
| ▷ | Disconnect all APPS |
| 278 | Motors do not turn |

Table 12: Inspection sheet steps. Steps marked with ▷ are actions done by the scrutineer. Steps marked with a number are the required behaviors by the car.

# B Alarms

This appendix contains a compiled list of all the alarms implemented in the alarm management system discussed in section 7.

| **Alarm Name** | **Alarm Description** |
|---|---|
| steeringAngleOutOfRange | The steering angle sensor has left the operating range. |
| brakePressureRearOutOfRange | The rear brake pressure sensor has left the operating range. |
| brakePressureFrontOutOfRange | The front brake pressure sensor has left the operating range. |
| torquePedalError1 | The Torque pedal sensor 1 has left the operating range. |
| torquePedalError2 | The torque pedal sensor 2 has left the operating range. |
| RTDSDisconnection | RTDS is disconnected. |
| LowVoltageBatteryLow | Low voltage battery has under voltage error. |
| coolingSystemManual | The cooling system is set to manual control. |
| pump1Disconnection | Pump1 is disconnected. |
| pump2Disconnection | Pump2 is disconnected. |
| stateChangeTimeout | The general state machine had a state timeout. |
| powerLimitWarning | The power concumption is close to the maximum allowed power. |
| powerLimitError | The power concumption is above the maximum allowed power. |
| torqueVectoringDisabled | Torque Vectoring is disabled. |
| launchProfileDisabled | Launch Profile is disabled. |
| tractionControlDisabled | Traction Control is disabled. |
| powerLimiterDisabled | Power Limiting is disabled. |
| regenBrakingDisabled | Regenerative Braking is disabled. |
| torqueLimiting | Torque Limiting is enabled. |
| inverter1disabled | Inverter 1 is disabled. |
| inverter2disabled | Inverter 2 is disabled. |
| inverter3disabled | Inverter 3 is disabled. |
| inverter4disabled | Inverter 4 is disabled. |
| inverter1ErrorDiagnostics | Report and display if inverter 1 transmits an error code. |
| inverter2ErrorDiagnostics | Report and display if inverter 2 transmits an error code. |
| inverter3ErrorDiagnostics | Report and display if inverter 3 transmits an error code. |
| inverter4ErrorDiagnostics | Report and display if inverter 4 transmits an error code. |
| inverter1Disconnected | Inverter 1 disconnected. |
| inverter2Disconnected | Inverter 2 disconnected. |
| inverter3Disconnected | Inverter 3 disconnected. |
| inverter4Disconnected | Inverter 4 disconnected. |
| amsDisconnection | AMS is disconnected. |
| amsUnderVoltage | AMS reports under voltage error. |
| amsOverVoltage | AMS reports over voltage error. |
| amsUnderTemperature | AMS reports under temperature error. |
| amsOverTemperature | AMS reports over temperature error. |
| amsCommunicationErrorLTC | AMS reports communication with cell monitoring chip LTC6813. |
| amsCommunicationErrorShunt | AMS reports communcation error with shunt. |
| amsPrechargeTimeout | AMS reports pre-charge timeout. |
| amsBankCountMismatch | AMS reports a bank count mismatch. |
| amsClosePreError | AMS reports unable to close precharge relay. |
| amsOpenPreError | AMS reports unable to open precharge relay. |
| amsCloseAirPlusError | AMS reports unable to close AIR+ relay. |
| amsOpenAirPlusError | AMS reports unable to open AIR+ relay. |
| amsCloseAirMinusError | AMS reports unable to close AIR- relay. |
| amsOpenAirMinusError | AMS reports unable to open AIR- relay. |
| amsScCloseError | AMS reports unable to close shutdown circuit. |
| amsScOpenError | AMS reports unable to open shutdown circuit. |

| | |
|---|---|
| amsUnexpectedChargingRate | AMS reports experiencing unexpected charging rates. |
| chargingCurrentWarning | Charging current warning. |
| dashboardDisconnection | Dashboard is disconnected. |
| dataloggingInactive | Datalogging is inactive. |

Table 13: List of the alarms handles by the alarm management system.

# C AMK Inverter State Machine

In figure 38 a recreation of the AMK inverter state machine for turning on and turning off a single inverter. The original state machine can be found in the AMK datasheet [**p.88**, 1].

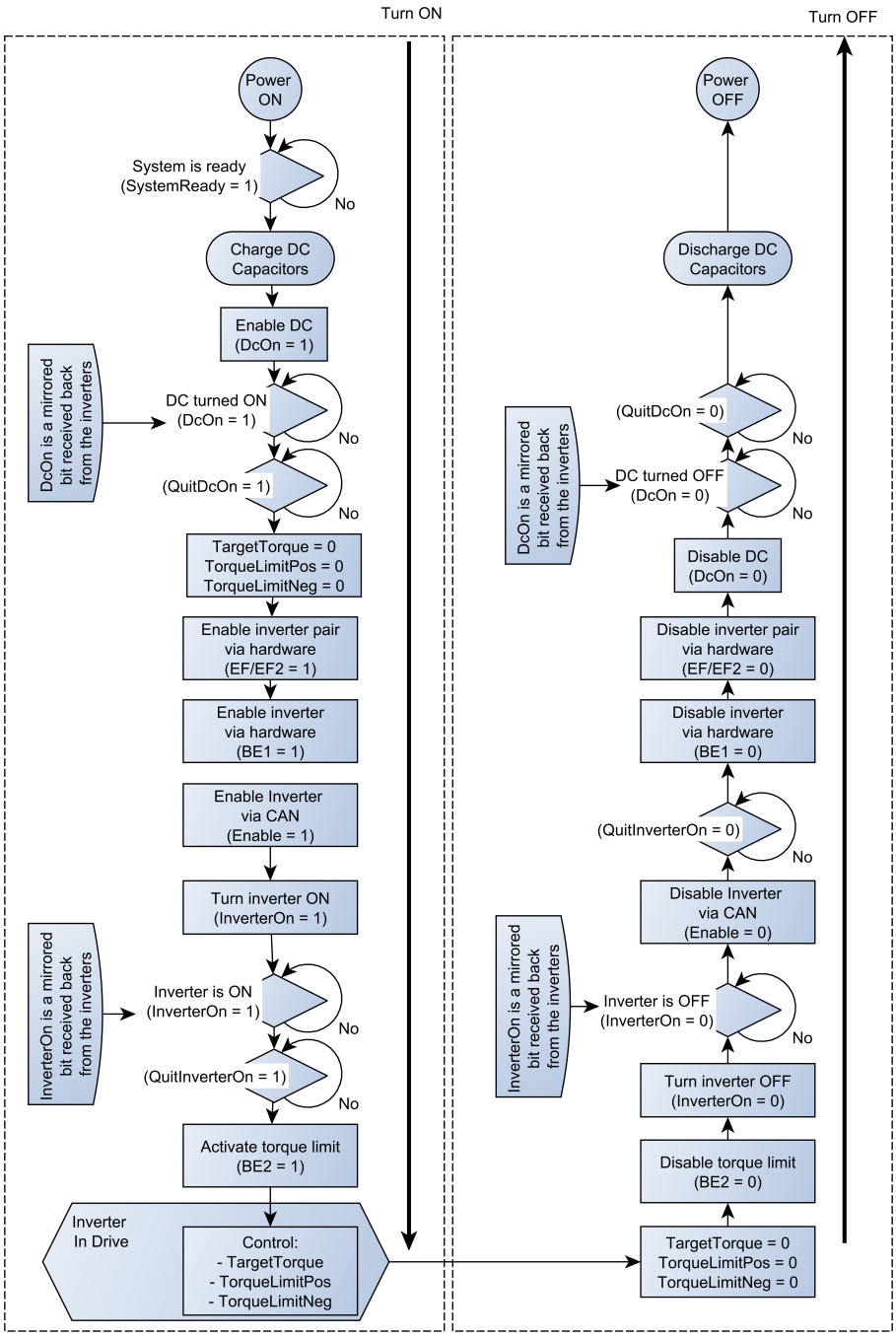


Figure 38: AMK Inverter state machine for turning on and turning off an inverter.

# D AMK Inverter Error Removal

The AMK inverter error removal precedure given in the AMK inverter datasheet [1].

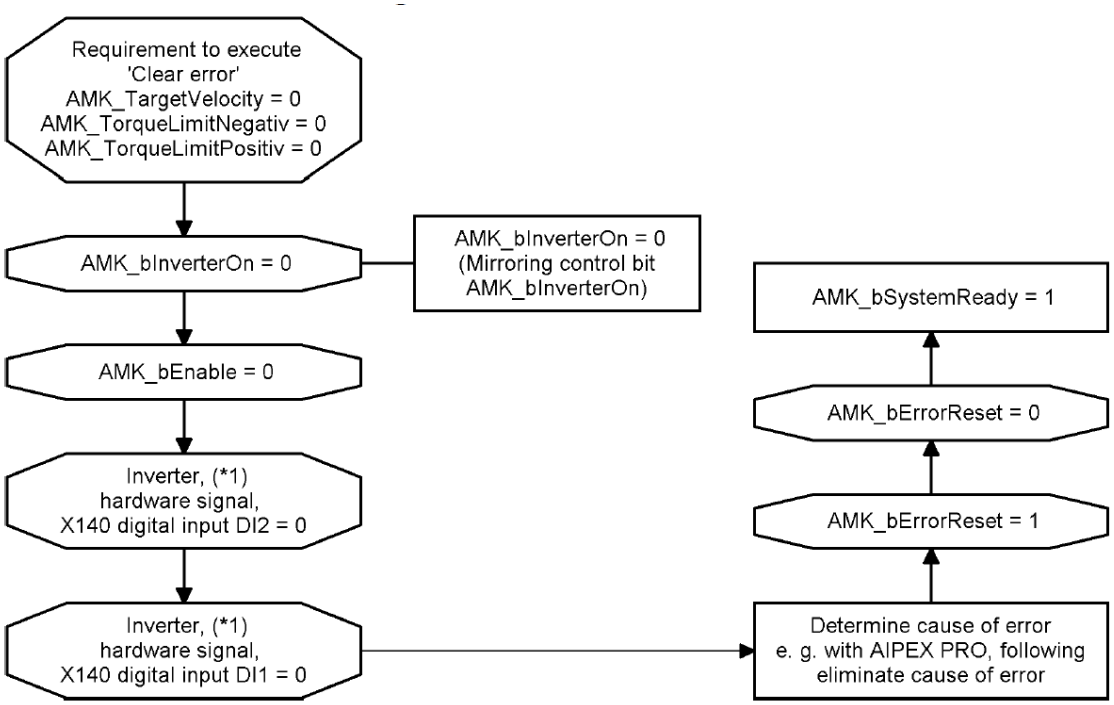


Figure 39: Flowchart for removing an error on an AMK inverter.

# E Shutdown Circuit Circuit

Overview of the shutdown circuit implemented in the car.

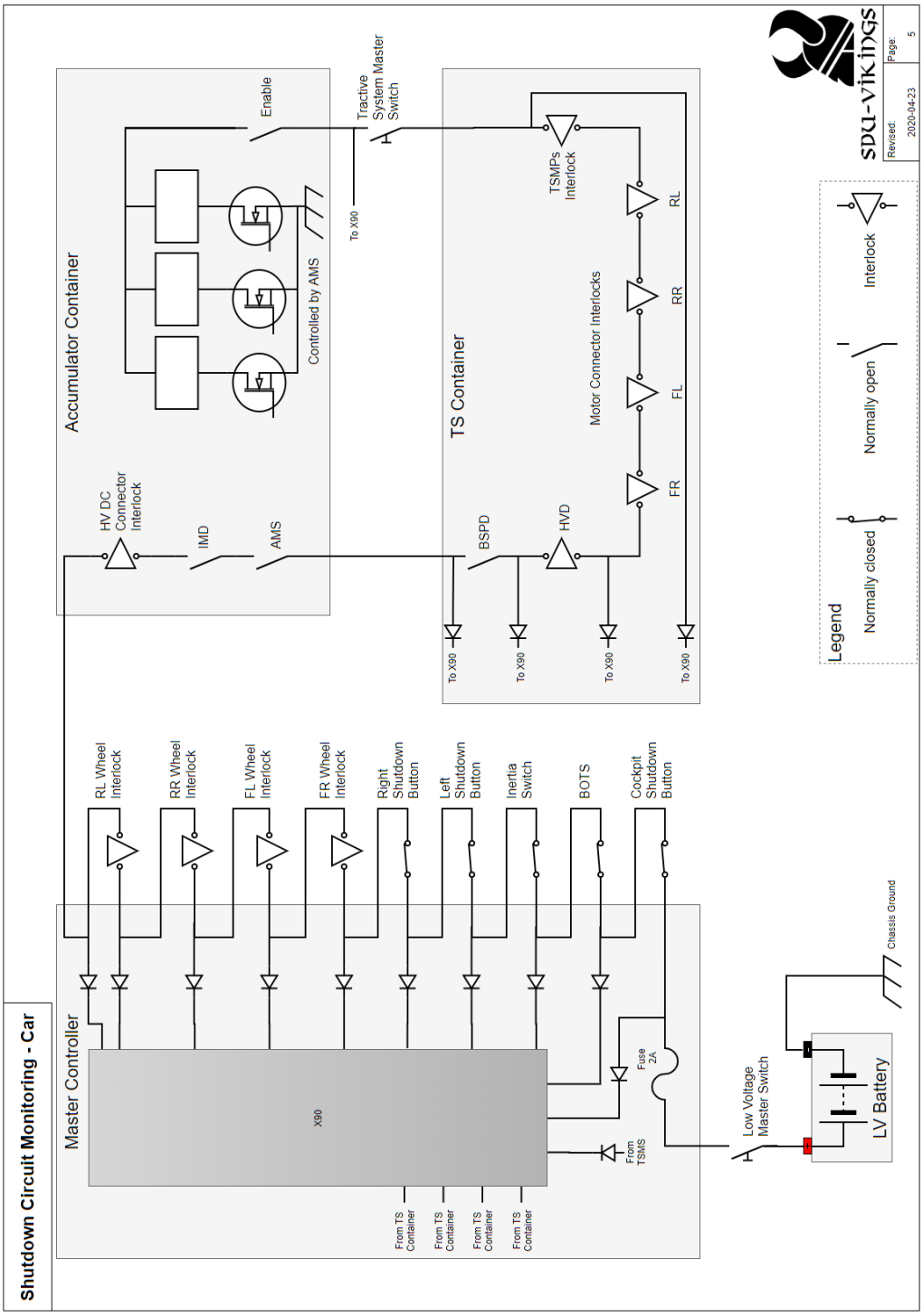


Figure 40: Overview of the shutdown circuit.

# F CAN Protocol Used Between Master Controller and Shunt

The following is the CAN protocol implemented between the master controller and the shunt. It might no longer be up-to-date. Visit `https://gitlab.com/sdu-vikings/X/master-controller/-/wikis/AMS-CAN-Protocol` for the newest information. Base Address for Shunt Messages: 0x500.

| **Parameter** | **Value** |
|---|---|
| Name | IVT_Msg_Result_I |
| Base Address Offset | 0x021 |
| Direction | Shunt → Master & AMS |
| Transmission Rate | Periodically: 20ms |
| Message Length (DLL) | 48 bit / 6 byte |

| **Variable** | **Start [bit]** | **Length [bit]** | **Unit** | **Type** | **Description** |
|---|---|---|---|---|---|
| MUX_ID | 0 | 8 | - | - | Multiplexer. See more in datasheet. |
| MSG_CNT | 8 | 4 | - | - | Cyclic counter. See more in datasheet. |
| ERROR | 12 | 4 | - | - | Error. See more in datasheet. |
| ACT_CURRENT | 16 | 32 | mA | sint32 | Actual current consumption. |

| **Parameter** | **Value** |
|---|---|
| Name | IVT_Msg_Result_U1 |
| Base Address Offset | 0x022 |
| Direction | Shunt → Master & AMS |
| Transmission Rate | Periodically: 60ms |
| Message Length (DLL) | 48 bit / 6 byte |

| **Variable** | **Start [bit]** | **Length [bit]** | **Unit** | **Type** | **Description** |
|---|---|---|---|---|---|
| MUX_ID | 0 | 8 | - | - | Multiplexer. See more in datasheet. |
| MSG_CNT | 8 | 4 | - | - | Cyclic counter. See more in datasheet. |
| ERROR | 12 | 4 | - | - | Error. See more in datasheet. |
| ACT_VOLTAGE_1 | 16 | 32 | mV | sint32 | Actual voltage 1. |

| Parameter | Value |
|---|---|
| Name | IVT_Msg_Result_U2 |
| Base Address Offset | 0x023 |
| Direction | Shunt → Master & AMS |
| Transmission Rate | Periodically: 60ms |
| Message Length (DLL) | 48 bit / 6 byte |

| Variable | Start [bit] | Length [bit] | Unit | Type | Description |
|---|---|---|---|---|---|
| MUX_ID | 0 | 8 | - | - | Multiplexer. See more in datasheet. |
| MSG_CNT | 8 | 4 | - | - | Cyclic counter. See more in datasheet. |
| ERROR | 12 | 4 | - | - | Error. See more in datasheet. |
| ACT_VOLTAGE_2 | 16 | 32 | mV | sint32 | Actual voltage 2. |

| Parameter | Value |
|---|---|
| Name | IVT_Msg_Result_T |
| Base Address Offset | 0x025 |
| Direction | Shunt → Master & AMS |
| Transmission Rate | Periodically: 100ms |
| Message Length (DLL) | 48 bit / 6 byte |

| Variable | Start [bit] | Length [bit] | Unit | Type | Description |
|---|---|---|---|---|---|
| MUX_ID | 0 | 8 | - | - | Multiplexer. See more in datasheet. |
| MSG_CNT | 8 | 4 | - | - | Cyclic counter. See more in datasheet. |
| ERROR | 12 | 4 | - | - | Error. See more in datasheet. |
| ACT_TEMP | 16 | 32 | $0.1^oC$ | sint32 | Actual temperatuer. |

| Parameter | Value |
|---|---|
| Name | IVT_Msg_Result_W |
| Base Address Offset | 0x026 |
| Direction | Shunt → Master & AMS |
| Transmission Rate | Periodically: 30ms |
| Message Length (DLL) | 48 bit / 6 byte |

| Variable | Start [bit] | Length [bit] | Unit | Type | Description |
|---|---|---|---|---|---|
| MUX_ID | 0 | 8 | - | - | Multiplexer. See more in datasheet. |
| MSG_CNT | 8 | 4 | - | - | Cyclic counter. See more in datasheet. |
| ERROR | 12 | 4 | - | - | Error. See more in datasheet. |
| ACT_POWER_W | 16 | 32 | W | sint32 | Actual power consumption in [watt]. |

| Parameter | Value |
|---|---|
| Name | IVT_Msg_Result_As |
| Base Address Offset | 0x027 |
| Direction | Shunt → Master & AMS |
| Transmission Rate | Periodically: 30ms |
| Message Length (DLL) | 48 bit / 6 byte |

| Variable | Start [bit] | Length [bit] | Unit | Type | Description |
|---|---|---|---|---|---|
| MUX_ID | 0 | 8 | - | - | Multiplexer. See more in datasheet. |
| MSG_CNT | 8 | 4 | - | - | Cyclic counter. See more in datasheet. |
| ERROR | 12 | 4 | - | - | Error. See more in datasheet. |
| ACT_AMP_HOUR | 16 | 32 | As | sint32 | Actual ampere hour counter in [amp*sec]. |

| Parameter | Value |
|---|---|
| Name | IVT_Msg_Result_Wh |
| Base Address Offset | 0x028 |
| Direction | Shunt → Master & AMS |
| Transmission Rate | Periodically: 30ms |
| Message Length (DLL) | 48 bit / 6 byte |

| Variable | Start [bit] | Length [bit] | Unit | Type | Description |
|---|---|---|---|---|---|
| MUX_ID | 0 | 8 | - | - | Multiplexer. See more in datasheet. |
| MSG_CNT | 8 | 4 | - | - | Cyclic counter. See more in datasheet. |
| ERROR | 12 | 4 | - | - | Error. See more in datasheet. |
| ACT_POWER_WH | 16 | 32 | Wh | sint32 | Actual power consumption in [watt*hour]. |

# G CAN Protocol Used Between Master Controller and Dashboard Node

The following is the CAN protocol implemented between the master controller and the dashboard node. It might no longer be up-to-date. Visit `https://gitlab.com/sdu-vikings/X/master-controller/-/wikis/Dashboard-CAN-Protocol` for the newest information.
Base Address for Dashboard Messages: 0x580.

| **Parameter** | **Value** |
|---|---|
| Base Address Offset | 0x000 |
| Direction | Master → Dashboard |
| Transmission Rate | Periodically: 100$ms$ |
| Message Length (DLL) | 64bit / 8byte |

| **Variable** | **Start [bit]** | **Length [bit]** | **Unit** | **Type** | **Description** |
|---|---|---|---|---|---|
| SPEED | 0 | 16 | 0.1km/h | sint16 | Current vehicle velocity |
| SOC | 16 | 16 | 0.1% | uint16 | Current state of charge on accumulator |
| MAX_TEMP | 32 | 16 | 0.1 $^oC$ | sint16 | Current highest motor temperature |
| LV_BAT_VOL | 48 | 16 | 0.1V | uint16 | Current voltage of low voltage battery |

| **Parameter** | **Value** |
|---|---|
| Base Address Offset | 0x001 |
| Direction | Master → Dashboard |
| Transmission Rate | Periodically: 100$ms$ |
| Message Length (DLL) | 16bit / 2byte |

| **Variable** | **Start [bit]** | **Length [bit]** | **Unit** | **Type** | **Description** |
|---|---|---|---|---|---|
| INSTR_IDX | 0 | 8 | - | uint8 | Index that can be used to look up instruction telling what to do next to get the car in drive. See table below instructions indexes. |
| CAR_STATE | 8 | 8 | - | uint8 | The current state of the vehicle. See table below for states. |
| ERR_CODE | 16 | 8 | - | uint8 | Index that can be used to look up error. See table below for error codes. |

| Parameter | Value |
|---|---|
| Base Address Offset | 0x002 |
| Direction | Master → Dashboard |
| Transmission Rate | Periodically: 100$ms$ |
| Message Length (DLL) | 64bit / 8byte |

| Variable | Start [bit] | Length [bit] | Unit | Type | Description |
|---|---|---|---|---|---|
| LED_IMD | 0 | 1 | - | bool | LED State. (0 = OFF, 1 = ON) |
| LED_AMS | 1 | 1 | - | bool | LED State. (0 = OFF, 1 = ON) |
| LED_SC | 2 | 1 | - | bool | LED State. (0 = OFF, 1 = ON) |
| LED_EBS | 3 | 1 | - | bool | LED State. (0 = OFF, 1 = ON) |
| LED_DV | 4 | 1 | - | bool | LED State. (0 = OFF, 1 = ON) |
| LED_RDY | 5 | 2 | - | uint | LED State. LED State. (00 = OFF, 01 = ON, 10 = Slow Blink, 11 = Fast Blink) |
| LED_LV | 7 | 1 | - | bool | LED State. (0 = ON, 1 = Fast Blink) |

| Parameter | Value |
|---|---|
| Base Address Offset | 0x003 |
| Direction | Master → Dashboard |
| Transmission Rate | Periodically: 100$ms$ |
| Message Length (DLL) | 48bit / 6byte |

| Variable | Start [bit] | Length [bit] | Unit | Type | Description |
|---|---|---|---|---|---|
| MISSION_IDX | 0 | 8 | - | uint | An index telling what mission the car is in. See table below for mission indexes. |
| STATUS_TV | 8 | 1 | - | bool | Status if torque vectoring is enabled. (0 = OFF, 1 = ON) |
| STATUS_TC | 9 | 1 | - | bool | Status if traction control is enabled. (0 = OFF, 1 = ON) |
| STATUS_RB | 10 | 1 | - | bool | Status if regenerative braking is enabled. (0 = OFF, 1 = ON) |
| STATUS_PL | 11 | 1 | - | bool | Status if power limitation is enabled. (0 = OFF, 1 = ON) |
| STATUS_SD | 12 | 1 | - | bool | Status if static derating is enabled. (0 = OFF, 1 = ON) |
| STATUS_LP | 13 | 1 | - | bool | Status if launch profile is enabled. (0 = OFF, 1 = ON) |
| RB_AVAILABLE | 14 | 1 | - | bool | Status if regenerative braking is available. (0 = NO, 1 = YES) |
| TRQ_LIMIT | 16 | 16 | 0.1Nm | uint | The torque available for the current mission |
| PWR_LIMIT | 32 | 16 | 0.1kW | uint | The power available for the current mission |

| Parameter | Value |
|---|---|
| Base Address Offset | 0x050 |
| Direction | Dashboard → Master |
| Transmission Rate | Periodically: 100$ms$ |
| Message Length (DLL) | 8bit / 1byte |

| Variable | Start [bit] | Length [bit] | Unit | Type | Description |
|---|---|---|---|---|---|
| BTN_SAFETY | 0 | 1 | - | bool | Status of the safety button. (0 = OFF, 1 = ON) |
| BTN_TRACT | 1 | 1 | - | bool | Status of the tractive button. (0 = OFF, 1 = ON) |
| BTN_DRIVE | 2 | 1 | - | bool | Status of the drive button. (0 = OFF, 1 = ON) |
| BTN_1 | 3 | 1 | - | bool | Status of the extra button 1. (0 = OFF, 1 = ON) |
| BTN_2 | 4 | 1 | - | bool | Status of the extra button 2. (0 = OFF, 1 = ON) |
| BTN_3 | 5 | 1 | - | bool | Status of the extra button 3. (0 = OFF, 1 = ON) |
| BTN_4 | 6 | 1 | - | bool | Status of the extra button 4. (0 = OFF, 1 = ON) |

| Parameter | Value |
|---|---|
| Base Address Offset | 0x051 |
| Direction | Dashboard → Master |
| Transmission Rate | Periodically: 100$ms$ |
| Message Length (DLL) | 8bit / 1byte |

| Variable | Start [bit] | Length [bit] | Unit | Type | Description |
|---|---|---|---|---|---|
| DSB_STAT_IDX | 0 | 8 | - | uint | An index telling the status of the dashboard. See table below for status indexes. |

| Parameter | Value |
|---|---|
| Base Address Offset | 0x052 |
| Direction | Dashboard → Master |
| Transmission Rate | Aperiodically |
| Message Length (DLL) | 8bit / 1byte |

| Variable | Start [bit] | Length [bit] | Unit | Type | Description |
|---|---|---|---|---|---|
| SET_MISSION | 0 | 8 | - | uint | Set the current mission. See table below for mission indexes. |

| Parameter | Value |
|---|---|
| Base Address Offset | 0x053 |
| Direction | Dashboard → Master |
| Transmission Rate | Periodically: 100$ms$ |
| Message Length (DLL) | 8bit / 1byte |

| Variable | Start [bit] | Length [bit] | Unit | Type | Description |
|---|---|---|---|---|---|
| ENABLE_TV | 0 | 1 | - | bool | Command to enable/disble torque vectoring. (0 = OFF, 1 = ON) |
| ENABLE_TC | 1 | 1 | - | bool | Command to enable/disble traction control. (0 = OFF, 1 = ON) |
| ENABLE_PL | 2 | 1 | - | bool | Command to enable/disble power limiting. (0 = OFF, 1 = ON) |
| ENABLE_TL | 3 | 1 | - | bool | Command to enable/disble torque limting. (0 = OFF, 1 = ON) |
| ENABLE_LP | 4 | 1 | - | bool | Command to enable/disble launch profile. (0 = OFF, 1 = ON) |
| ENABLE_RB | 5 | 1 | - | bool | Command to enable/disble regenerative braking.. (0 = OFF, 1 = ON) |

# H CAN Protocol Used Between Master Controller and AMK Inverters

The communication protocol between the Master Controller and the 4 AMK Inverters are the standard protocol supplied by AMK.
The following CAN protocol might no longer be up-to-date. Visit `https://gitlab.com/sdu-vikings/X/master-controller/-/wikis/Inverter-CAN-Protocol` for the newest information.
Base address for AMK Actual Values 1 and 2: 0x280
Base address for AMK Setpoints 1: 0x180

| Inverter # | NODE_ID |
|---|---|
| Inverter 1 | 1 |
| Inverter 2 | 2 |
| Inverter 3 | 5 |
| Inverter 4 | 6 |

Table 14: Inverter Node ID's

| **Parameter** | **Value** |
|---|---|
| Name | AMK Actual Values 1 |
| Base Address Offset | 0x002 + NODE_ID |
| Direction | Inverters → Master |
| Transmission Rate | Periodically: $5ms$ |
| Message Length (DLL) | 64bit / 8byte |

| **Variable** | **Start [bit]** | **Length [bit]** | **Unit** | **Type** | **Description** |
|---|---|---|---|---|---|
| AMK_Status | 0 | 16 | - | uint16 | Status word see [**p.60**] for word content. |
| AMK_ActualVelocity | 16 | 16 | RPM | sint16 | Actual speed value |
| AMK_TorqueCurrent | 32 | 16 | - | sint16 | Raw current $I_q$ current. See [**p.80**] how to calculate current in [A] |
| AMK_Magnitizing-Current | 48 | 16 | - | sint16 | Raw current $I_d$ current. See [**p.80**] how to calculate current in [A] |

| Parameter | Value |
|---|---|
| Name | AMK Actual Values 2 |
| Base Address Offset | 0x004 + NODE_ID |
| Direction | Inverters → Master |
| Transmission Rate | Periodically: $5ms$ |
| Message Length (DLL) | 64bit / 8byte |

| Variable | Start [bit] | Length [bit] | Unit | Type | Description |
|---|---|---|---|---|---|
| AMK_TempMotor | 0 | 16 | 0.1 $^oC$ | sint16 | Motor Temperature |
| AMK_TempInverter | 16 | 16 | 0.1 $^oC$ | uint16 | Cold Plate Temperature |
| AMK_ErrorInfo | 32 | 16 | 0.1 $^oC$ | uint16 | Diagnostics Number |
| AMK_TempIGBT | 48 | 16 | 0.1 $^oC$ | sint16 | IGBT Temperature |

| Parameter | Value |
|---|---|
| Name | AMK Setpoints 1 |
| Base Address Offset | 0x003 + NODE_ID |
| Direction | Master → Inverters |
| Transmission Rate | Periodically: $30ms$ |
| Message Length (DLL) | 64bit / 8byte |

| Variable | Start [bit] | Length [bit] | Unit | Type | Description |
|---|---|---|---|---|---|
| AMK_Control | 0 | 16 | | uint16 | Control word see [**p.61**] for word content |
| AMK_TargetTorque* | 16 | 16 | $Nm$ | sint16 | Torque Setpoint |
| AMK_TorqueLimit-Positive | 32 | 16 | 0.1%$M_N$ | sint16 | Positive torque limit (subject to nominal torque) |
| AMK_TorqueLimit-Negative | 48 | 16 | 0.1%$M_N$ | sint16 | Negative torque limit (subject to nominal torque) |

* In the datasheet this is called *AMK_TargetVelocity* because the datasheet describes the message when the inverter is speed controlled but in the Viking X car the inverters are torque controlled and then this changes to be the target torque instead.

# I Liveview Visualization Pages

In the following section the different liveview pages implemented in the system can be seen.
**Note**: The header bar and left navigation bar seen on the Main Page is also present on all the other liveview pages but has been left out on the figures to make them bigger.

## I.1 Main Page

The Main Page is used to give a general overview of the car, and is made to be understandable and relatable to people outside the team.

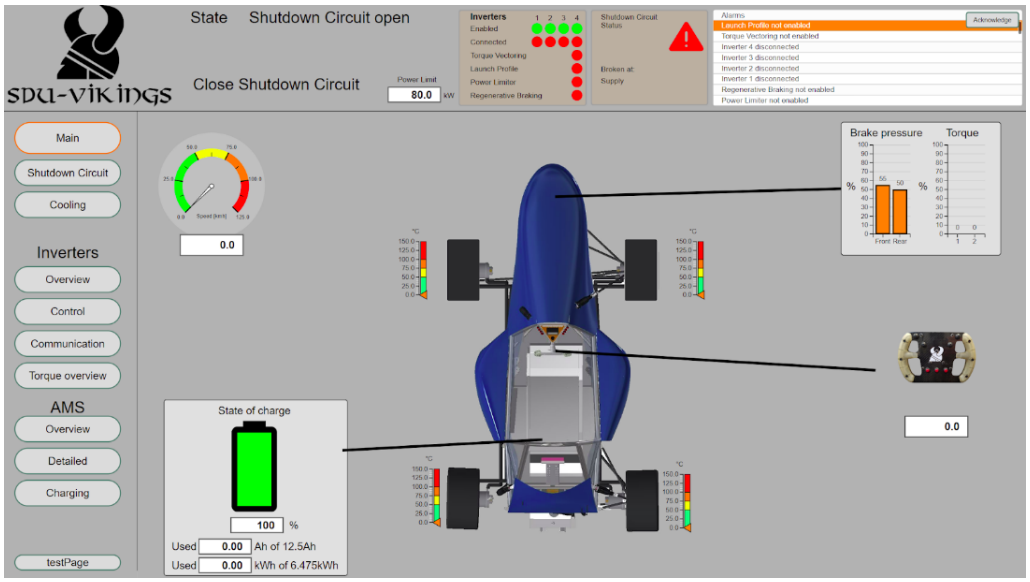


Figure 41: The Main Page.

## I.2 Shutdown Circuit Page

The Shutdown Circuit Page is there to give a better overview of the current shutdown circuit status. Each element in the circuit has an LED associated with it and if the element is good, the LED will be green otherwise it will be grey. For more information about the shutdown circuit see section 10.

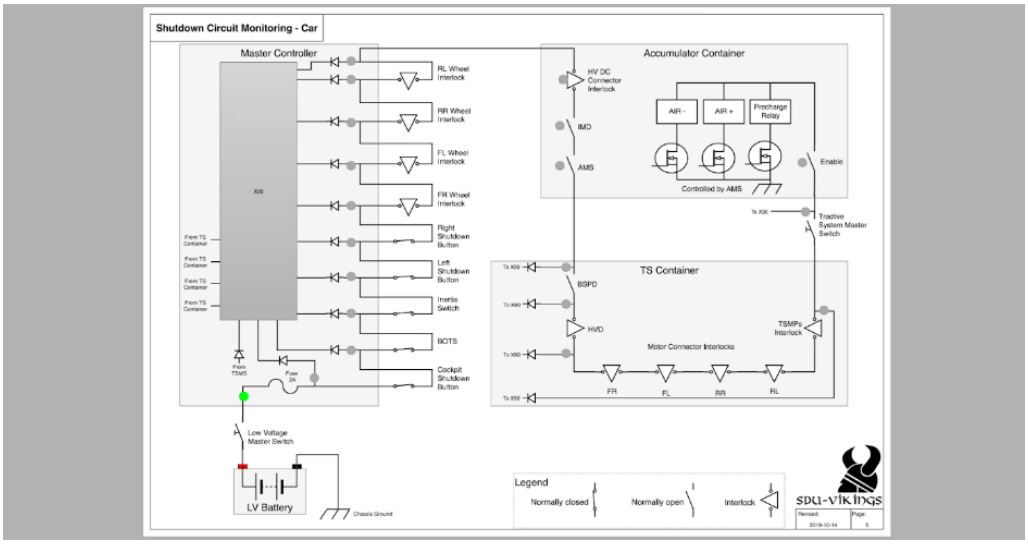


Figure 42: The Shutdown Circuit Page

## I.3 Cooling System Page

The Cooling System Page give an overview of the temperatures measured in the car as well as giving the status of the cooling pumps and fans. It is also here that manual mode can be enabled as was discussed in section 12.

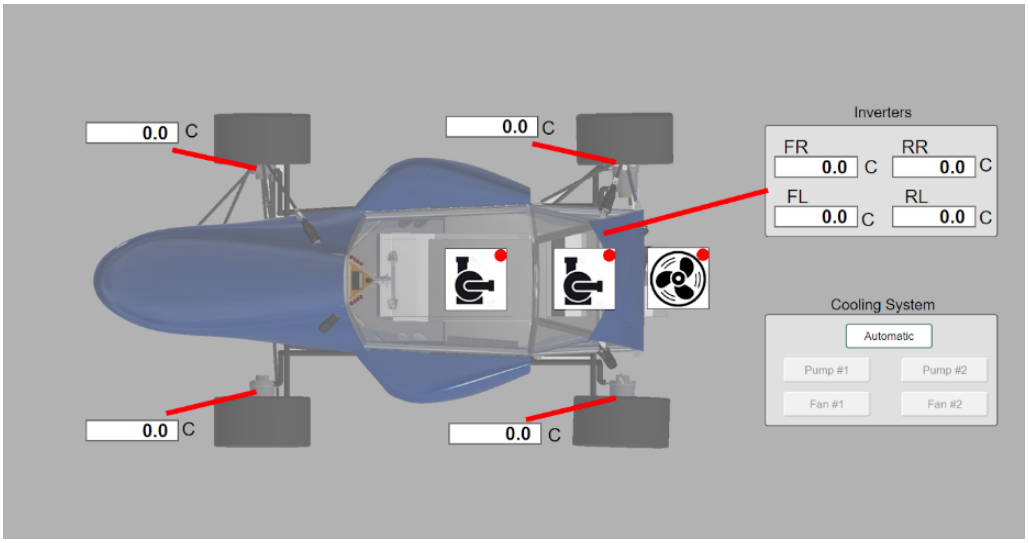


Figure 43: The Cooling System Page.

## I.4 Inverter Pages

The Inverter Overview Page is made to give a general overview of the 4 inverters and motors and their current statuses.

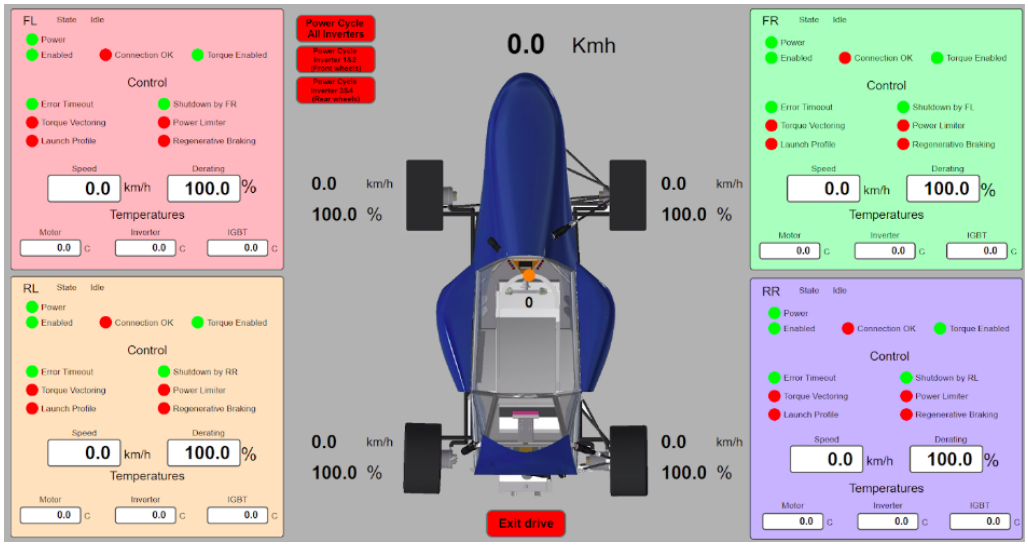


Figure 44: The Inverter Overview Page.

The Inverter Control Page is made to monitor and configure the current vehicle dynamics controller working in the car.

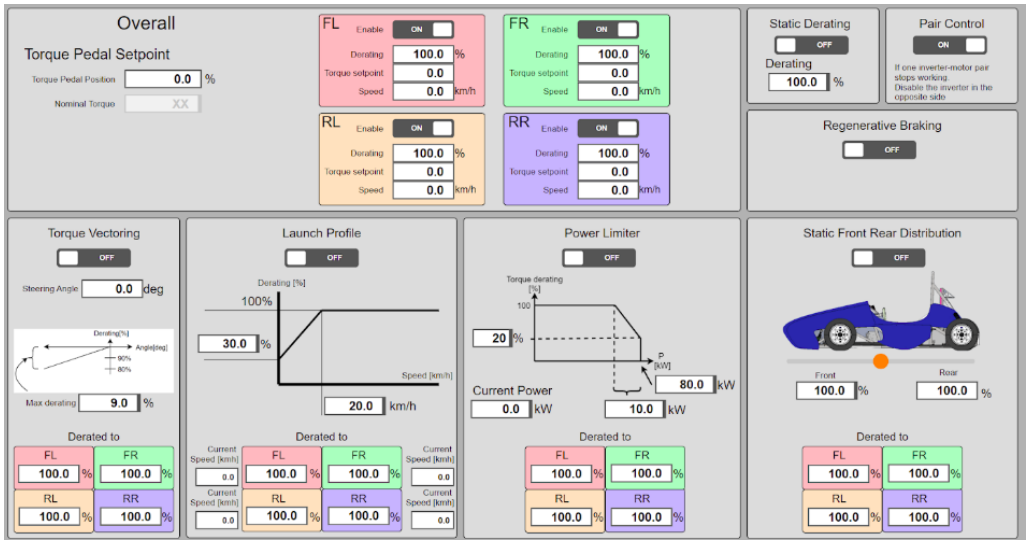


Figure 45: The Inverter Detailed Page.

## I.5 AMS Pages

The AMS Overview Page is made to give a general overview of the 5 banks making up the battery, as well as total voltage, output current, state of charge and different status signals.

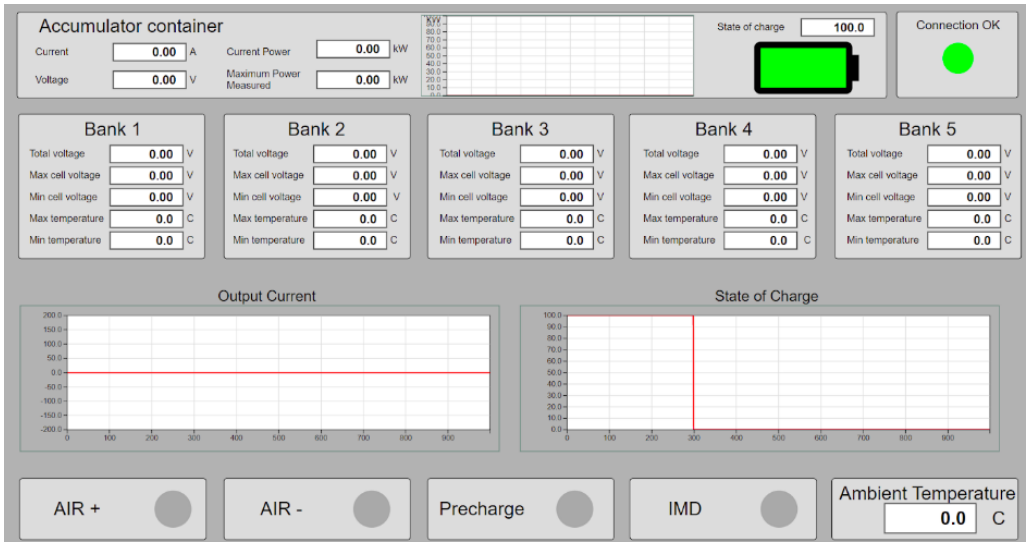


Figure 46: The AMS Overview Page.

The AMS Detailed Page can be used to monitor the battery more closely and gives the voltage of every cell and all temperatures measured in the battery. Note that these graphs are not line-graphs but bar-charts. The lines on the image marks the upper and lower voltage levels allowed for the cells. All cells are $0V$ because the picture is taken from the simulator.


Figure 47: The AMS Detailed Page.

# J Automation Studio Starter Guide

The following pages contains the Automation Studio Starter Guide. The guide is meant to give an overview of the B&R Automation software framework with a focus on the SDU-Viking relevant aspects.

# Automation Studio Starter Guide

By: Jacob Offersen

The following document is the product of an elective Formula Student course taken in the spring of 2020 (F20)
Developed to go along with the Master Controller for the Viking X


PERFECTION IN AUTOMATION
A MEMBER OF THE ABB GROUP


| Done by | CPR | Exam Nr | Signature |
|---|---|---|---|
| Jacob Offersen | 311294 | 84929345 | |

## Advisor

Karsten Holm Andersen, Associate Professor at SDU


Handed in on 31/05/2020

# 1 Introduction

The following document is a Automation Studio starter guide tailored for SDU-Viking team members that wants to develop software systems controlled by B&R Automation hardware.
The document contains short descriptions of the most important aspects and concepts and provides information about where to find more information if needed. B&R Automation provides a wide range of starter guide documents used for education of both B&R staff and B&R customers and this document is not made replace those. The relevant B&R documents for Formula Student can be found in the bibliography.
This starter guide is written as a supplementary guide meant to go along with Master Controller Documentation for the Viking X.

SDU

# Contents

# 2 Installation

Programming and transferring programs to B&R products is done with the software environment *Automation Studio*. Automation Studio can be downloaded from B&R Automation's website. Be aware of downloading version 4.5 which is the version used in the car, download link.

## 2.1 Automation Studio License

To use the software a license needs to be obtained. A 90 days free license can be obtained at `https://www.br-automation.com/da/service/automation-studio-licensing/`. Choose *Evaluation license for Automation Studio* and fill out the form. An email with the license will be send shortly. When the 90 days are over a new free license can be obtained following the same method and this can be done indefinitely.

## 2.2 Modules and packages

For the most part all B&R hardware such as PLC's, PLC expansion boards and so on can be found and included directly in Automation Studio. When new products are released they are not included in old version of Automation Studio but only appear by default in newer version of the Automation Studio. All hardware modules used for the Viking X and Viking X charger are included in Automation Studio 4.5 by default except one X90 option board. The option board in question is the *X90IF720.04-00* communication module which was released after Automation Studio version 4.5 release. The module files have been downloaded separately and placed in the project folder under *Project/as*. The hardware can be installed to Automation Studio by doing the following. Going to the toolbar and choosing "Tools>Upgrades". In the windows that opens choosing the tab "Local" and navigate to the project folder and into the subfolder "as". Then tick off the *X90IF720.04-00* module and pressing "Install Selected Upgrades". After the installation the module is added to Automation Studio and it should work as all other modules.

# 3 Introduction to Automation Studio

The following section is meant to give a broad overview of the Automation Studio program. For a more in-depth introduction to Automation Studio see [1].
Automation Studio is the program through which the whole system is setup, programmed, debugged, tested and diagnosed. The program has a wide range of features and can be used exclusively when programming B&R Automation PLC's.
On figure 1 a picture of the Automation Studio environment can be seen.

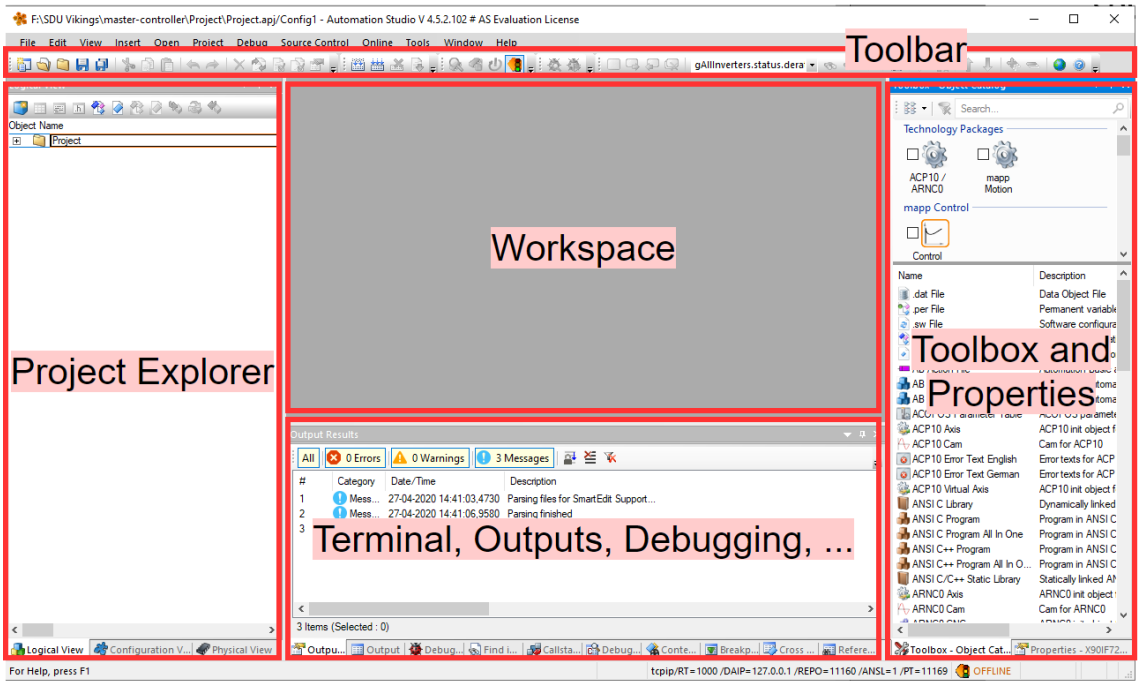


Figure 1: Automation Studio Overview

The program has the following windows:

- **Project Explorer**, which is used to navigate around in the project.
- **Workspace**, which is used to edit whatever has been selected in the Project Explorer. This can be code, variable declarations, I/O mapping, configurations and more.
- **Toolbar**, which is used for saving, opening, building, starting a simulated PLC, starting the debugger and searching the project
- Bottom window, which contains the terminal, error window, debugging window and many more.
- **Toolbox**, which contains items that can be dragged into the project. This is where new task files, variable files, library templates, visualization widgets and more can be found.
- **Properties**, which is where certain properties of a given object can be changed. For visualization widgets this is where the styling and variable binding happens.

## 3.1 Project Explorer

The project explorer is divided into 3 tabs called *Logical View*, *Configuration View* and *Physical View*.

| Icon | View | Description |
|---|---|---|
|  | Logical View | Represents the hardware-independent view which is the software files such as variable files, file containing code as well as the visualization files. |
|  | Configuration View | Represents the hardware-dependent view which contains hardware configurations, software configurations, I/O mapping and more. |
|  | Physical View | Represents the hardware which is a tree structure view of the hardware modules. |

## 3.2 Toolbar

The toolbar contains the following

| Icon | Name | Description |
|---|---|---|
|  | New Project | Create a new project |
|  | Open Project | Open an existing project |
|  | Close Project | Close the current project |
|  | Save | Save the current file open in the workspace |
|  | Save All | Save all files open in the workspace |
|  | Build | A fast way to build the project but this will only build files that has been changed since the last build. If the project has not been build before this button is the same as the "Rebuild" button. |
|  | Rebuild | Do a complete build of the whole project. This is slow and for the most part a normal "Build" is more than enough. The only times a rebuild it needed is if the project has not been build before or if a file included in a file has been changed. Fx a picture has been updated for the visualization. This type of change is not detected by the standard "Build". |
|  | Transfer | If a build has not been performed it will first do a "Build" and then it will open the transfer window used to transfer the build project to the connected PLC or simulated PLC. |
|  | Monitor | Open a watch to live monitor variable values |
|  | Stop Target | Set the target offline |

| | | |
|---|---|---|
|  | Warm Restart | Restart the PLC. There is also a cold restart but the Warm is for the most part enough. A warm restart is the same as a power cycle and the same that happens when a PLC restarts after getting a new code transferred. |
|  | Activate Simulation | Startup a simulated PLC that can be used to test code on a computer before putting it on the actual hardware. |
|  | Help | Opens the Automation Studio Help. Read more about it in section 3.3 |

### 3.3 B&R Help

The Help window can be opened by either pressing F1 anywhere within Automation Studio, or by pressing the Help button in the toolbar. The help is the main way in which coding help, hardware help or any other support can be found. The help is a dictionary containing all documentation about Automation Studio and technical software details about B&R products. It is not possible to find help online so the Help is the main way to get insight.

## 4 Operating System - Automation Runtime

Devices from B&R Automation comes with an operating system called *Automation Runtime* [2, p.7]. Some of the operating system features include

- Makes the application hardware-independent
- Each task class can be configured to be Hard and Soft Real-Time
- Guarantees deterministic behavior with a cyclic runtime system
- Provides up to 8 different task classes and cycle times each with a real-time configuration
- Guarantees a response to timing violations
- Provides configurable timing tolerance limits for each task class

Each cycle class requires a cycle time and a cycle tolerance. The cycle time specifies how often tasks in the class should be scheduled and the tolerance provides the possibility of a task to surpass its deadline with a specified amount making the system soft real-time. If a task passes its deadline + tolerance the system will enter *Service Mode* and the PLC will stop executing application code.

Read more about the operating system in [2] and in the B&R Help under "Real-time operating system".

### 4.1 Tasks

All application code has to be contained in a task. A task is grouped in cycle class's and the task will inherit the cycle class's cycle-time and tolerance.
A task has 3 sections. *INIT*, *CYCLIC* and *EXIT*. The *INIT* routine is called only once when the PLC starts up. The *CYCLIC* part is what is scheduled into the CPU with a static period during run-time. The *EXIT* part is executed only when the PLC is running and a new program is loaded onto the PLC and is therefore rarely used. The *EXIT* routine is mostly used for function blocks that needs to give up their resources before new instances of themselves are initiated.

Read more about the topic in the B&R Help under "Real-time operating system > Method of operation > Runtime performance > Tasks".

### 4.2 Libraries

Automation Studio contains a long list of libraries that can be included in a project to provide additional features. The libraries can be found in Automation Studio in the "Toolbox" window under "B&R Libraries". Choosing this module will open a window contain a list of all libraries. Please note that not all libraries are free. Some require a license and if one of these are included the project will report a license violation every time it is build.
It is also possible to make custom libraries including any number of custom function blocks. Read more about how this is done and about the build in libraries in the B&R Help under "Programming > Libraries".

## 5 Input and Output Handling

The reading of input and writing of outputs are just like the tasks bound to a specific cycle class. Each I/O can be either manually bound to a cycle class or set to *inherit* in which the I/O is bound to the cycle class of the fastest task using the I/O.

## 6 Connect to a device

PLC's can be connected to via a wired Ethernet cable or if the PLC is connected to a Wi-Fi antenna then through Wi-Fi.

### 6.1 Setup computer's Ethernet or Wi-Fi

The PLC will have a IP address associated with its Ethernet port. The IP address can be found in Automation Studio under Physical View. The Ethernet port can then be right clicked and "Configuration" chosen. On the configuration page the IP address is specified under "PORT ADDRESS > Activate interface > Device Parameters > Mode > IP address".
For the X90 Master Controller used for the Viking X car has the following setup:
IP Address: **169.254.143.70**
Subnet Mask **255.255.0.0**
To connect to the PLC make sure the computer used is within the IP range of the PLC. It can sometimes be an advantage to set the computer IP to be static and set it so the PLC is in range. It does come the downside that the computer can then not access the internet through the same Ethernet/WiFi port while this setup

is used. So in the case the same port is used for internet and the PLC, the network port will have to be change back and forth between static and dynamic.
On Windows the IP address can be changed by right clicking the internet icon in the taskbar and choosing 'Open Network & Internet Settings". In the Settings window choose Wi-Fi or Ethernet depending on what port should be used and click "Change adapter options". The Network Connections window will open and here the same port should be right clicked and 'Properties' should be chosen. Choose "Internet Protocol Version 4 (TCP/IPv4)" and click the "Properties" button. In here the port can either be set to dynamic/automatic by clicking "Obtain an IP address automatically" which should be used to access the internet, or click "Use the following IP Address" and a static address in the range of the PLC should be inserted. An example of a valid IP address could be:
IP Address: **169.254.143.77**
Subnet Mask: **255.255.0.0**
When this has been done the computer is ready to connect to the PLC.

## 6.2 Connect via Automation Studio

When the computer has been setup correctly open Automation Studio.
If the Automation Studio project has been setup to automatically use the PLC it should automatically connect and be ready for a transfer of the program.
If this is not the case fx if the project is new. Go to the taskbar and choose "Online>Settings". This should open the "Online Settings" window. In the taskbar of this window make sure the "Browse" button is pressed. If the PLC and computer are both configured correctly the PLC should appear on the right side of the Online Settings window. The X90 PLC would appear as the only device on the list. Right click the device and choose "Connect". This should move the device to the left side of the window and change its letters to bold. If this happens the device connection status in the lower right corner should change to "RUN". By marking the "Use in active config" check-mark for the PLC on the left side of the window the PLC will be automatically connected to if it appears in the network in the future.
The PLC is now ready for a transfer of the program.

Read more about the topic in the B&R Help under "Programming > Build & Transfer > Establishing a connection to the target system".

# 7 Transfer project to device

Once the project has been build with either the "Build" or "Rebuild" buttons in the toolbar the "Transfer" button can be pressed. This will open a window called "Transfer to target" which can be seen in figure 2.

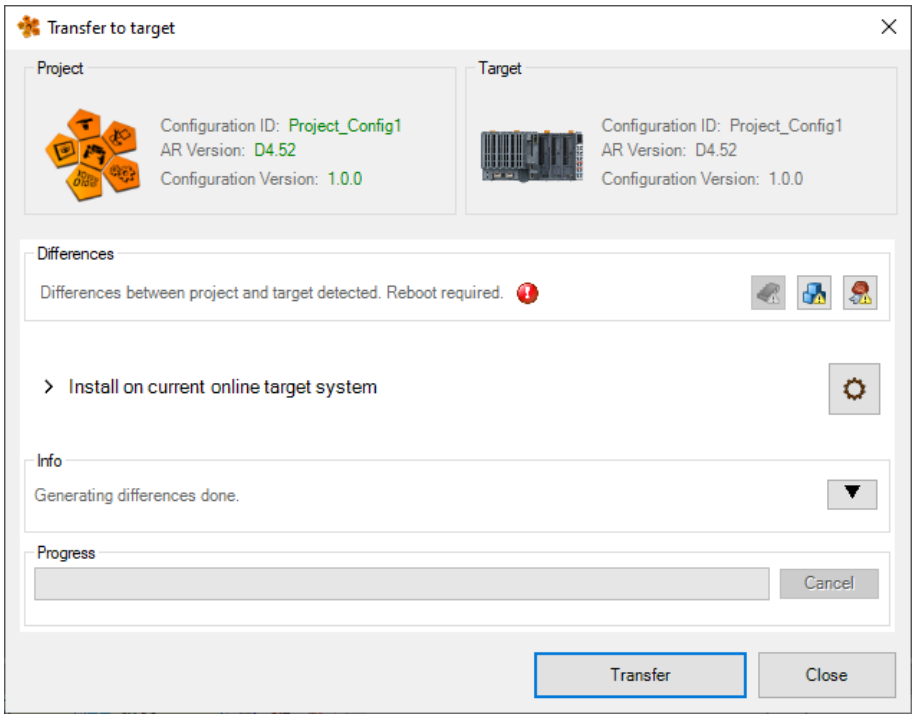


Figure 2: The "Transfer to target" window

Here the project configuration, Automation Runtime version and configuration version can be seen both for the project about to be transferred as well as what is currently running on the PLC. The "Transfer" button can then be pressed after which the Progress bar will start to fill. First the project will be transferred to the PLC and then the PLC will restart.
In the simulator this process is super fast only taking a second or two. This process will take a couple seconds if the PLC is connected via Ethernet cable and even longer if it is connected via WiFi antenna. Give it time, sometimes the antenna solution can take up to 20 sec.

Read more about the topic in the B&R Help under "Programming > Build & Transfer > Transfer to target".

# 8 The Basics of Structured Text

Structured text is the main way that the B&R controllers are programmed. They can also be programmed with other programming languages such as Ladder and C but they are best suited for Structured Text and therefore this is chosen.
The structured text looks a lot like VHDL without being hardware descriptive and the syntax for the most used functions can be seen below.

```
(* A comment can look like this *)
// Or like this
```

Code snippet 1: Structured Text Syntax Examples

```
(* Operations *)
A := B; (* Value assignment *)

A := NOT B; (* Not statement. Only works if A and B are boolean variables *)

(* Math operations *)
A := B + C;      (* Addition *)
A := B - C;      (* Subtraction *)
A := B * C;      (* Multiplication *)
A := B / C;      (* Division *)
A := B MOD C;    (* Modulo *)

(* Comparisons *)
A := B = C;      (* Equal to *)
A := B <> C;     (* Not equal to *)
A := B < C;      (* There are also >, >=, <= *)

(* Bitwise operations*)
A := B AND C;    (* Bitwise AND operation *)
A := B OR C;     (* Bitwise OR operation *)
A := B XOR C;    (* Bitwise XOR operation *)
```

Code snippet 2: Structured Text Syntax Examples

```
(* If statement *)
IF condition1 THEN
    (* Do something *)
ELSIF condition2 THEN
    (* Do something *)
ELSE
    (* Do something *)
END_IF

(* For loop *)
FOR i := x TO y DO
    (* Do something *)
END_FOR

(* Switch case *)
CASE variable OF
    x:
            (* Do something *)
    y:
            (* Do something *)
    ELSE
            (* Else clause *)
END_CASE
```

Code snippet 3: Structured Text Syntax Examples

Read more about Structured Text in the B&R Help under "Programming > Programs > Structured Text (ST)".

## 8.1 Function Blocks

In structured text the concept of functions as known from C and C++ does not directly exist. The closest structured text gets to this is a *Function Block* or a subroutine. A function block is a way to hide code and execute it by calling the function block just like with a normal function.
The differences between functions and function blocks shows up in the way data is parsed to it and the way to execute the code inside.
In a function known from C or C++ the function is called and inside the parenthesis the data is placed and the function is executed.

```
(* How to make a function call in a C like language. *)
someFunction(data1, data2);
```

Code snippet 4: "Example of C code"

A function block is somewhat different. The data can be parsed in two different ways and the data can be parsed without executing the code and the code can also be executed without parsing any data.
The first way looks like the C version with the only difference that the function parameters needs to be included and the data assigned directly in the function block call.

```
(* How to parse and make a function call in Structured Text. *)
someFunction(D1 := data1, D2 := data2);
```

Code snippet 5: "Example 1 of structured text code"

The other way the data is assigned separately from the function call itself. The data can be assigned when it is found and this way of assigning data is useful when it comes on different points in the program or is one data point depends on a condition a new parameter is not needed to keep the value before parsing it to the function.

```
(* How to first parse data and then do the function call in Structured Text. *)
someFunction.D1 := data1;
someFunction.D2 := data2;
someFunction();
```

Code snippet 6: "Example 2 of structured text code"

The code inside function blocks are only run when the last line is executed.

Read more about the topic in the B&R Help under "Programming > Functions and function blocks > Function block".

# 9 Simulation

All code can be simulated in real time on a computer. If a visualization is present the computer will also host it and if data-logging is used it can be configured to use the computers USB port or internal memory.
The simulation is started by pressing the traffic light symbol in the toolbar. This will start up a simulated PLC. When the simulated PLC has started up (shown by the status in the lower right corner saying "RUN") the project can be build with one of the build buttons in the toolbar. See section 3.2 for more details about the build options. Once the project has been build it can be transferred to the simulated

PLC with the Transfer button in the toolbar. This will bring up a window identical to that for a normal PLC which will give the possibility of transferring the project and rebooting the simulated PLC.

If the code does not seem to run on the physical PLC first thing to check is if the simulation is active.

## 9.1 Changing variable values during runtime

Variables can be changed during run-time both on a simulated PLC as well as a normal PLC running on a test bench or in the car. The PLC needs to be connected to the computer and the status in the lower right corner needs to say "RUN". If this is the case the *Monitor* button in the toolbar can be pressed. This will grey out the workspace window and open up a small window called *Watch* on the right side of the workspace. Variables from the code can now be marked and dragged into the watch window or the watch window can be right clicked and any variable can be selected. Once a variable is in the watch it will update live and by clicking on its value field its value can be changed.
If the value is constantly being controlled by the code the change made in the watch will be overwritten when the code is executed on the next cycle.
Values directly controlling pin output will prompt the user asking if they are sure they want to change the value because this can destroy physical equipment. If "Yes" is clicked the value can then be forced. To bring the variable back to be controlled by the code it can be right clicked and "Force Variable" can be disabled.
One variables being used in the current workspace window can be added to the watch.
Read more about the topic in the B&R Help under "Diagnostics and services > Diagnostrics tools > Watch (variable monitoring)".

# 10 System Diagnostics

Automation Studio has a wide range of useful tools to monitor and debug the software on a running PLC. The most important tools are mentioned in this section.

## 10.1 Logger

The Logger is an event log type feature that logs all events happening in the system. This both includes normal events that cause no harm (fx a new user logging into the visualization), it includes warnings (fx license violations) and critical erros (fx invalid pointers).
The logger can be found in the toolbar under "Open > Logger" and in the logger window a selection of categories can be chosen to be viewed. The selected categories can also be chosen to automatically update if new events happen or be manually updated by the user.

Read more about the Logger in the B&R Help under "Diagnostics and service > Diagnostics tools > Logger".

## 10.2 Profiler

The Profiler can be used to get a detailed report of how the CPU is being used. It can be used to show a lot of details about how much CPU each process is taking. It includes CPU usage percentage for each process and each process's minimum, average and maximum processing times both in net time and gross time. Where net time is the pure processing time per cycle for a given process and gross time includes potential interrupts and higher priority process running in the middle of the tasks execution.
The profiler is a hugely powerful tool and is the best tool for determine the systems performance.

Read more about the Profiler in the B&R Help under "Diagnostics and service > Diagnostics tools > Profiler".

## 10.3 Watch

The Watch is a way to monitor variable values during runtime on either a simulated PLC or a physical PLC. The Watch can be access through "Open > Watch" and is a window version of the in-line watch discussed in section 9.1.

Read more about the Watch in the B&R Help under "Diagnostics and service > Diagnostics tools > Watch".

## 10.4 System Diagnostics Manager

The System Diagnostics Manager (SDM) is the first place to look how a PLC is running. The SDM is a webpage hosted by the PLC which shows important runtime information such as software versions, node number, current PLC status, CPU temperatures, IP address, CPU usage, memory usage, cycle class, module connection statuses, and much more. Through the SDM the Logger and Profiler can also be accessed.
The SDM is a very important general diagnostics tool and are often implemented directly in the machine visualizations.
The SDM can be disabled for a PLC to reduce CPU usage but if it enabled is can be accessed through the browser from any computer connected to the PLC with the following URL:
*http://{IP-address}/sdm/index.html*
The IP-address should be the address of the PLC and for the master controller this would for example be:
**http://169.254.143.70/sdm/index.html**

Read more about the System Diagnostics Manager in the B&R Help under "Diagnostics and service > Diagnostics tools > System Diagnostics Manager (SDM)".

# 11 Liveview - Mapp View

There exists two main ways to host visualizations on B&R Automation devices. *Visual Components* and *Mapp View* where the former is being outphased and replaced by the latter. Mapp View is a drag-and-drop style visualization development tool developed by B&R Automation to improve the processes of designing visualizations for system monitoring and control.
The visualization is handled in CPU idle time where no other task is scheduled to run. Which means if the CPU has a high load it stops handling the visualization so the visualization does not affect the system performance. Read more about idle time in the B&R Help under "Real-time operation system > Method of operation > Runtime performance > Scheduling > Idle time".
Figure 3 shows an example of the *Mapp View* editor.

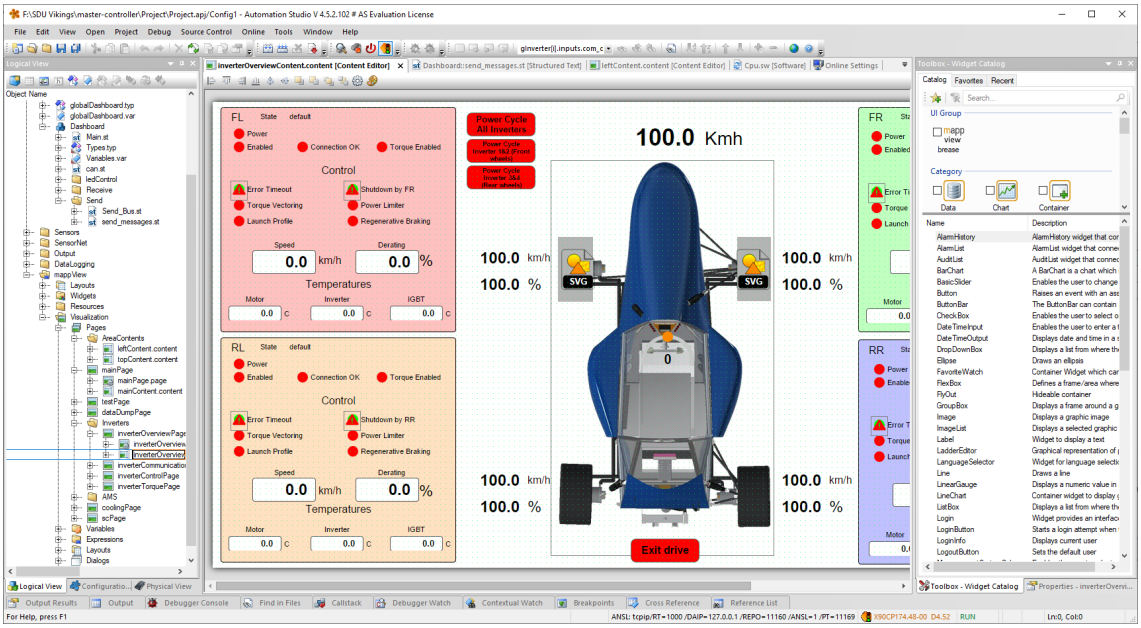


Figure 3: Overview of the mappView editor in Automation Studio. In the middle the visualization page can be seen and on the right the list if widgets that can be used can be seen.

A visualization is divided up into the various pages that it should consist of. Each page can be configured to fulfill its specific purpose by dragging and dropping in the needed software widgets.
A widget can be everything from a line to a button, numeric input, image, table, line chart, keyboard and many others. In Mapp View version 5.2 over 70 different widgets can be used. Each widget has a wide range of configuration possibilities to make it work and look the right way.
Figure 4 shows the Mapp View properties window in which the configuration possibilities of a Numeric Output, which is used to display a single numeric value, can be seen.

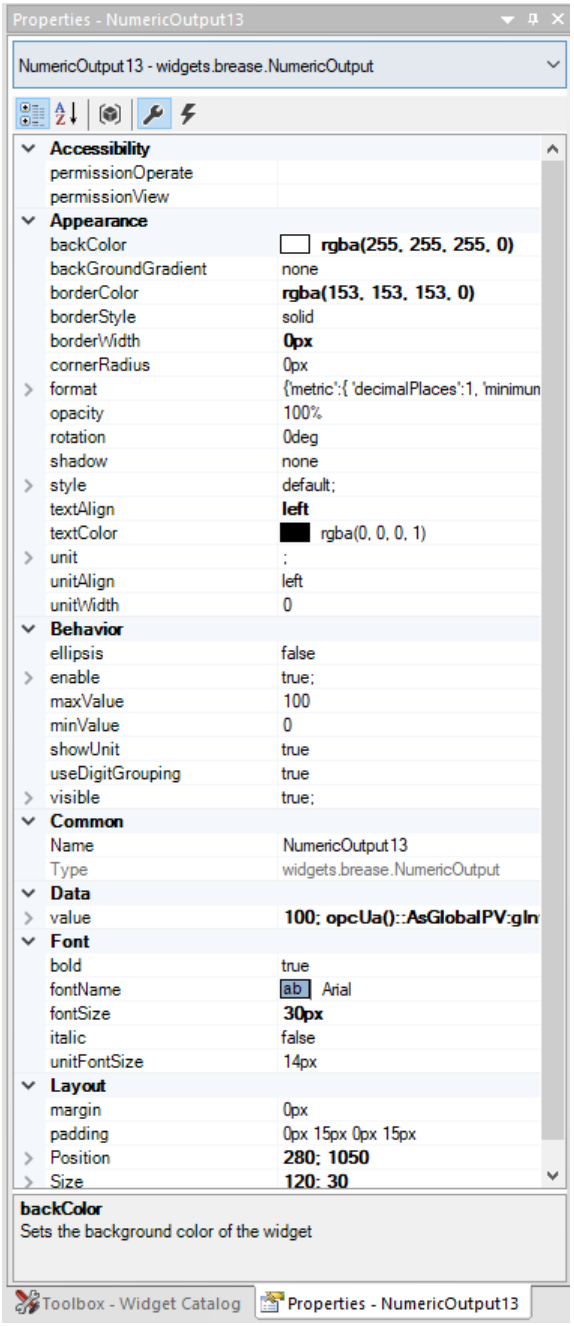


Figure 4: Overview of the widget properties of a numeric output used to display a numeric variable value.

To link a widget to the process variables used in the PLC code the variable first needs to be made available and this is done through the communication protocol OPC-UA. When this is done it can be linked to any number of widgets through their *value* field in the widget's properties window.

Read more about Mapp View in the B&R Help under "Visualization > mapp View".

## 11.1 Communication protocol - OPC-UA

**O**pen **P**latform **C**ommunication **U**nified **A**rchitecture (OPC UA) is an international standard that is developed for and by the automation industry to make communication between hardware from different manufacturers easier. The platform is made to provide safe and reliable end-to-end communication between sensors, actuators and the processing unit. [3, p.4-5]
OPC-UA is used to provide the variable data to the visualizations for being shown to users. When working with visualizations the OPC-UA platform can be seen simply as a variable value hosting service.
For a variable to be hosted it needs to be enabled on the OPC-UA configuration page under "Configuration View > {Config Name} > {PLC Name} > Connectivity > OpcUA > OpcUaMap.uad". Navigate to the wished variable, mark it and click the checkmark button in the OpcUa windows toolbar.
When the variable has been enabled it can be bound to a widget. The variable value will be shown on the visualization page and refreshed with the frequency setup for the binding and the system. For *standard bindings* the value is refreshed every $100ms$ but the standard binding frequency can be changed. A bindings can also be set to *fast binding* or *slow binding* giving the possibility of choosing between two other refreshing frequencies for some bindings. For the majority of cases the standard binding can be used, fast bindings are usually used for animations and slow bindings for slowly changing variables.

Read more about OPC UA in the B&R Help under "Communication > OPC UA".

# 12 Bibliography

Besides the B&R Help build into Automation Studio, B&R Automation also provides a lot of educational documents used for education of both internal B&R staff as well as external staff of B&R customers. The most Formula Student relevant manuals have been listed in the bibliography below.


[1] *TM210 Working with Automation Studio*. Version TM210TRE.461-ENG. B&R Automation.

[2] *TM213 Automation Runtime*. Version TM213TRE.461-ENG. B&R Automation.

[3] *TM980 OPC UA basics and use*. Version TM980TRE.452-ENG. B&R Automation.

[4] *TM223 Automation Studio Diagnostics*. Version TM223TRE.461-ENG. B&R Automation.

[5] *TM230 Structured Software Development*. Version TM230TRE.00-ENG. B&R Automation.

[6] *TM246 Structured Text (ST)*. Version TM246TRE.00-ENG. B&R Automation.

[7] *TM291 Basics of simulation for industrial control technology*. Version TM291TRE.452-ENG. B&R Automation.

[8] *TM600 Introduction to Visualization*. Version TM600TRE.00-ENG. B&R Automation.

[9] *TM611 Working with mapp View*. Version TM611TRE.444-ENG. B&R Automation.

[10] *TM641 Alarms, charts, data in mapp View*. Version TM641TRE.444-ENG. B&R Automation.

[11] *TM671 Creating powerful mapp View visualizations*. Version TM671TRE.444-ENG. B&R Automation.

[12] *TM910 Control and IO system design*. Version TM910TRE.00-ENG. B&R Automation.

[13] *TM920 Diagnostics and Service*. Version TM920TRE.00-ENG. B&R Automation.

[14] *TM923 Diagnostics and Service with Automation Studio*. Version TM923TRE.444-ENG. B&R Automation.