CHALMERS
UNIVERSITY OF TECHNOLOGY

Data-driven Understanding of User Interaction with Human Machine Interface (HMI) in the Automotive Industry

Master's Thesis in Master's Programme Systems, Control and Mechatronics

ERIK HOLMSTEDT

Department of Industrial and Materials Science
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021

Master's thesis 2021

Data-driven Understanding of User Interaction with Human Machine Interface (HMI) in the Automotive Industry

ERIK HOLMSTEDT

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Industrial and Materials Science
Division of Product Development
Chalmers University of Technology
Gothenburg, Sweden 2021

Data-driven Understanding of User Interaction with Human Machine Interface (HMI) in the Automotive Industry
ERIK HOLMSTEDT

Supervisor: Julia Orlovska, Department of Industrial and Materials Science
Supervisor: Jonas Thorsell, Volvo Group Trucks Technology
Examiner: Rikard Söderberg, Department of Industrial and Materials Science

Master's Thesis 2021
Department of Industrial and Materials Science
Division of Product Development
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Zoom in of the infotainment system in a Volvo Truck, including Instrument Cluster, Secondary Information Display and steering wheel switches.

Typeset in LaTeX
Gothenburg, Sweden 2021

Data-driven understanding of user interaction with Human Machine Interface (HMI) in the Automotive Industry
Erik Holmstedt
Department of Industrial and Materials Science
Chalmers University of Technology

Abstract

User behavior evaluation within the automotive field has traditionally been based on qualitative methods like interviews and surveys. However, the improved data availability and a stronger focus on data utilization make this approach change fast towards the data-driven evaluation since decision-making in development needs an evidence-based approach. Numerous studies of user interaction based on quantitative methods like data logging have been performed. Most of them are focused on external ECU communication (CAN, LIN, Ethernet, etc.) or video recordings. To fully understand all user interaction steps, data collected directly from the HMI needs to be studied. Therefore, as a first step, semi-structured interviews with HMI Engineers at Volvo GTT were conducted to determine relevant logging points. Afterward, a user interaction logging setup was developed for an Instrument Cluster in trucks, based on internal ECU data. Data logged includes usage of Instrument Views, Focus Shift, Gauges and a menu application.

In designing the logging, parts of the HMI have been modelled using Discrete Event Systems theory to identify the system behavior. Moreover, the analysis has been made to determine the most suitable level of the software to extract wanted data. Finally, driver card information has proven to be a beneficial way to identify unique users within the truck industry. Logging is done in the form of DOIDs, a type of parameters with static definitions of the data structure. However, for the future, a more dynamic high-frequency logging setup is needed to connect logged user data to driving conditions and system settings.

As the user interaction logging has been verified successfully, a mixed methods approach is proposed once real customer data is available. Logged data standalone will not explain patterns found in the data, but a more complete picture can be given when combining quantitative and qualitative methods.

Keywords: User interaction, Human Machine Interface, Data logging, Automotive, User behavior evaluation, Instrument Cluster, Infotainment, Trucks, Mixed methods

Acknowledgements

First, I would like to thank my supervisor Julia Orlovska from Chalmers for giving me the chance to perform this thesis and being a great support along the way. Thanks for your excellent guidance and patience since our first meeting one and a half year ago.

At Volvo GTT I would like to thank all my colleagues and friends that have helped and supported me with the project. I couldn't have done this without you. Extra thanks to my supervisor Jonas Thorsell for pushing me to the finish line.

Last but not least I want to thank my family, my mother Karin, my brother Oskar and my late father Bertil. I love you!

Erik Holmstedt, Gothenburg, June 2021

Contents

List of Figures

List of Tables

List of Abbreviations

ACC - Adaptive Cruise Control
APX - AUTOSAR Port eXchange
AUTOSAR - Automotive Open System Architecture
CAN - Controller Area Network
CC - Cruise Control
DOID - Diagnostic Object Identifier
ECU - Electronic Control Unit
GDPR - The General Data Protection Regulation
GFX - Graphics
HMI - Human Machine Interface
IC - Instrument Cluster
LHS - Left Hand Side
LIN - Local Interconnect Network
MVC - Model-View-Controller
OBD - Onboard Diagnostic
OEM - Original Equipment Manufacturer
PTI - Pre-Trip Inspection
QML - Qt Modelling Language
RHS - Right Hand Side
RPM - Revolutions Per Minute
SID - Secondary Information Display
SWC - Software Component
SWS - Steering Wheel Switch
UTC - Coordinated Universal Time
VIN - Vehicle Identification Number

1
Introduction

1.1 Background

The driver environment in cars and trucks is getting more and more complex these days. Plenty of new features have been added over the past years, like navigation, more advanced media functionality, etc. With the widespread introduction of digital displays in vehicles, it is now possible to develop more dynamic Human Machine Interfaces (HMI) with a faster development cycle. With higher complexity of automotive systems, it also gets harder and harder to understand how the driver interacts with the system. The driver might interact in different ways depending on, for instance, the driving conditions or the current system settings. With a cleverly designed HMI the driving experience can be enhanced. The trip can be safer if driver interaction with safety-critical functions is clear. Making it simpler to find or adjust functions can add flexibility and comfort and decrease driver distraction.

However, the often cited Standish Group Study presented by Jim Johnson found that 64% of the functions in some of their typical systems were rarely or never used [1]. Although this is not a general truth for all systems, many people can easily reflect on the amount of features never used in their own personal vehicles.

Today, evaluation of the user behavior and its relation to the HMI within the automotive industry is mainly done by inquiry methods like surveys, questionnaires, etc. This approach makes it hard to evaluate all functions as a vehicle typically consists of several hundred functions. Therefore, the evaluation is usually focused on a small number of issues [2]. Another problem could be the difficulty to decide for certain that the driver has experienced the situation explained in a specific question.

One way to get around these problems is to base evaluations on vehicle data. Based on actual vehicle data, it can be precisely determined to what degree certain functions are used and during which conditions. Also, potential differences between different groups of drivers can be identified. With knowledge about what functions drivers use the most, how these functions are used or if drivers do not use a specific function, design decisions can be taken, improving the quality level of the HMI.

Previous user interaction studies in the automotive field based on vehicle data have mostly focused on logging signals between ECUs, like CAN, LIN and Ethernet data. Orlovska et al. have in their research shown that combining vehicle data logging (quantitative approach) and driver interviews (qualitative approach) can give a deeper understanding of how a function is utilized [3]. When only looking at communication between ECUs, activation of functions, etc., can be detected, but not how the user navigated in menus to do the activation. For this kind of evaluation, data needs to be logged directly from the graphical user interface.

This study aims at investigating the possibility to conduct user interaction evaluation based on vehicle data collected from the graphical user interface. A logging setup is developed for the HMI. The logged data aims to answer such questions as what views are used, how the user navigates in menus and what buttons are pushed. Moreover, in this work, the best practices for the evaluation are investigated: what data is relevant, what tools and models are needed, how is the quality of the data secured, etc.

1.2 Purpose

The thesis aims to find simple and effective ways to make detailed user behavior evaluations within the automotive field connected to HMI displays. The hypothesis is that logging data from the HMI itself will benefit this kind of analysis, given that interesting data and smart ways of structuring the data can be found. A better understanding of how a product and its HMI is used can provide a better foundation for understanding how to design a HMI. This is important for automotive companies that want to develop user-friendly HMI and remain competitive.

An additional purpose of this work is to pinpoint any differences found within the truck industry compared to the car industry in the context of user behavior evaluation and data logging.

1.3 Objective

The objective of this thesis is to develop a setup for logging of user interactions with the HMI in an Instrument Cluster developed by Volvo Trucks. Data will be logged and analyzed to draw conclusions about the HMI in question.

In order to achieve this, a number of subproblems will have to be solved:

- Conduct interviews with HMI Engineers to define interesting parts of the HMI to analyze.

- Create a model of the HMI to find efficient ways of logging user interaction data.
- Develop logging setup.
- Find good practice of extracting the data from the Instrument Cluster.
- Verify the implementation of the logging setup.
- Describe how data logging can be used during product development, for instance, by combining qualitative and quantitative approaches to get a deeper understanding of findings from the logged data.

1.4 Scope

The main scope of the thesis is the development of a logging setup to analyze user behaviors within the automotive industry and, in this particular case, with the HMI in an Instrument Cluster.

The scenarios of user interactions with the HMI in the Instrument Cluster that are considered as parts of the thesis are:

- Instrument Views. Which views are used by the driver? How much time does the driver spend in each Instrument View, and how often is it changed?
- Gauges. What gauges does the user choose to display, i.e., what data is of interest to the driver?
- Navigation. How does the user navigate in menus and inside the different applications? Do the user patterns follow the intended behavior by HMI engineers?
- Efficiency. How much time does the user spend in different parts of the HMI or need to complete certain tasks?
- Interaction with Secondary Information Display (SID). On which display does the driver keep the focus? Details on how the user navigates inside the SID cannot be determined when only logging data in the Instrument Cluster, but a complexity analysis can be made based on number of button pushes.

The scenarios to be investigated are based on input from engineers and designers working closely with the HMI and the possibilities of what data that can be logged.

1.5 Limitations

To narrow down the scope of the thesis, a number of limitations have been made. The limitations are the following:

- Anonymization of data will not be the focus of this thesis. The focus is on how to find relevant data and utilize it. Still, logging user data is a sensitive subject. The logging must not be made without the approval of the user. A truck has a VIN number, making it possible to identify the user. Truck drivers can also be identified based on driver card data, which will be described more in-depth in later chapters. It could also be possible to identify specific users solely based on user patterns. In all cases when it is possible to identify individual users, the data has to be anonymized by post-processing.
- Only data that is available internally in the Instrument Cluster will be logged. A wider view of the system could potentially be achieved when combining data from the Instrument Cluster with data from other ECUs or intercommunication between ECUs, but this data will not be considered for now.
- To understand when or what triggers a specific driver behavior, the current driving situation could be considered. It would be interesting to study whether the driver acts differently depending on the traffic situation or the weather conditions, for instance, but no consideration will be taken to these things.
- To model and log the complete HMI is a time-consuming task. Therefore, the logging is limited to a few chosen applications as a proof of concept. Further parts of the HMI will be logged later when a successful logging strategy is set.
- To store data to Diagnostic Objects per user of a truck instead of on truck level adds a lot of complexity to the implementation. Therefore data is logged per user for only a few of the logged attributes as a proof of concept. This can be considered enough to prove that it would be possible to do for all data in a fully developed logging setup.
- Due to the long lead time of collecting vehicle data from customers, data has just been logged from two trucks in the internal test fleet at Volvo GTT for a limited time period. User behavior evaluation should not be based on a small sample of test fleet data, but the data can still be used to visualize how to work with customer data in the future.

1.6 Thesis Structure

This thesis report consists of seven chapters in total.

- The 1st chapter introduces the topic of user behavior evaluation and the purpose of the thesis. Limitations to the thesis are also stated.
- The 2nd chapter provides information about some of the theories used for the thesis and presents the outcome of some related studies.
- The 3rd chapter provides an overview of how the truck is built up with a high focus on the components used for the logging and the Instrument Cluster in particular.
- The 4th chapter presents the methods used for the thesis. All the way from interviews with HMI Engineers to set the scope of the logging, to implementation and collection of data.
- The 5th chapter presents the results and findings of the thesis.
- The 6th chapter includes a discussion on identified possibilities and issues. Additionally, future development of the logging system is suggested.
- The 7th chapter concludes the thesis in a condensed manner.

2
Theory

This chapter will describe the theory used for the thesis and results from some related studies.

2.1 User Behavior Evaluation

There are different ways of performing user behavior evaluations. Most methods can be sorted either under qualitative or quantitative research approaches. Established practices at the particular OEM, strict data protection regulations and poor data availability often make qualitative data the main resource in user-related research [2].

2.1.1 Qualitative Methods

To clearly define what is meant by qualitative research has proven difficult over the years. However, Strauss and Corbin [4] defined it as "By the term 'qualitative research', we mean any type of research that produces findings not arrived at by statistical procedures or other means of quantification. It can refer to research about persons' lives, lived experiences, behaviours, emotions, and feelings as well as about organisational functioning, social movements, cultural phenomena, and interactions between nations.".

Most common methods in qualitative research are unstructured interviews, participant observations and direct observations with describing records [5]. Data collection involves direct interaction with the participants, meaning that the data can be very detailed, but also it is subjective [6]. One advantage of qualitative research design is that it has a flexible structure [7], meaning that the design for how an issue is tackled can be reconstructed to get the appropriate analysis done. According to Denzin and Lincoln [8] "Qualitative research is multi-method in focus, involving an interpretive, naturalistic approach to its subject matter".

One major disadvantage of qualitative data is the small sample size, which makes it difficult to generalize the research findings for the whole population [6].

Apart from the risk of generalization, the cases' analysis can be very time consuming [9].

2.1.2 Quantitative Methods

The ambition of quantitative methods is to answer questions like how many, how much and to what extent [10]. Bryman described it as "A research strategy that emphasises quantification in the collection and analysis of data..." [11]. One typical quantitative method is data logging.

Compared to qualitative methods results from quantitative methods are more suited for generalization as a much larger sample can be included [12]. It is also less time consuming as it is possible to utilize advanced statistical software for data analysis [13].

One major disadvantage of quantitative methods is that it often fails to give deeper underlying meanings and explanations [6]. You can typically see from the data that something has occurred and how often, but you do not necessarily know why. The results found may also be too general or abstract to apply for the whole range of driving situations, different markets and other specific or individual parameters [14].

2.1.3 Combining Qualitative and Quantitative Methods

Combination or mixing of qualitative and quantitative research techniques is referred to as mixed methods research. The idea of mixed methods research is to emphasize the strengths of each method and mitigate the weaknesses [14]. The strength of mixed methods is that words, pictures and narratives can add understanding to numbers and numbers can add precision to the narratives. In this way meanings can be added that might be missed when only using one of the methods. It can also add generalizability of the results.

Weaknesses of mixed methods is that it might be difficult for a researcher to perform both qualitative and quantitative research, it requires learning and understanding of multiple approaches and how to mix those. This means that a team might be needed to perform the research, and it quickly gets more time consuming and expensive [14].

One example of applying mixed methods to research is from Orlovska et al., where Automated Driver Assistance Systems was studied [15]. First, a quantitative approach was applied to collect user data. Based on findings from that data and identified patterns, a qualitative study was performed where a subset of the drivers from the quantitative study was interviewed. As qualitative studies are flexible in its design they could easily be adapted based on the results from the quantitative study and provide a deeper understanding to the identified patterns.

2.1.4 Usability

User interaction is the concept of how a user acts on a system and vice versa. In a user interface the user can perform tasks and based on the interactions from the user the system will interact with the user in a certain way. For a user interacting with a system the usability is key. Usability of human-system interaction is defined as an "extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" by ISO 9241-11 [16]. The ambition of usability engineers therefore is to design a system that people find usable and will use [17].

There is no clear definition of what attributes to use for a usability assessment, but Becerril et al. performed a literature review where a list of 60 attributes was found [18]. Some examples of attributes from the list are efficiency, flexibility, ease of use, understandability and consistency.

How the usability evolves can be described in two steps according to Peham et al., the learning process and the usage process [19]. As a new user, you need to gain knowledge on the system by studying, practicing and improving your skills. This is referred to as the learning process. A user will hopefully improve the usage of the system and at a certain point the user performance will stabilize. This is when the usage process has been reached. Once the user has learned how to use the system the easiness of using the system can be measured.

2.1.5 Related Studies

Numerous studies have been done on user interaction and driver behavior with quantitative methods in the automotive industry. One example is the 100-car naturalistic study where crashes and near crashes were studied [20]. Yet another example is the MIT Advanced Vehicle Technology Study, where data was collected to understand human interaction with vehicle automation technology [21]. As already mentioned, Orlovska et al. have also collected user data to study Adaptive Driver Assistance Systems [2].

These studies use data from CAN, LIN, Ethernet, etc. Some even use video recordings, but in none of these studies data is directly extracted from inside the HMI. As stated by Orlovska et al. in another study "We cannot register all clicks within the graphical user interface, only those that lead to a changing of hardware status. As a result, we are not currently able to measure the number of steps needed to complete the task, and therefore we cannot see if the users learned the optimal path to activate the function." [22]. Hence these studies can not provide a full understanding of the user performance in the HMI. Therefore in this study, we aim to close this gap and extract user interaction data from the Instrument Cluster.

2.2 Discrete Event Systems

One way of modelling a Human Machine Interface is as a discrete event system using automata theory. There are a number of examples where this method has been used, for instance by Adachi et al. where automata theory was used to model both a HMI and its user to identify automation surprises [23].

2.2.1 Discrete States and Events

A system described with discrete states is done so under the assumption that the state only changes at discrete time instances $t_k, k = 1, 2, 3, ...$ [24]. A discrete state x, the input signal u and output signal y can only take values from a discrete countable set

$$x \in \Omega_x = \{x^{(1)}, x^{(2)}, x^{(3)}, ...\}, u \in \Omega_u, y \in \Omega_y \tag{2.1}$$

where Ω_x is called the state space of the system. The state space model describing state x at time t_k is seen below

$$x(t_k^+) = f(x(t_k), u(t_k), t_k) \tag{2.2}$$
$$y(t_k) = g(x(t_k), u(t_k), t_k) \tag{2.3}$$

When a state is changed from one state to another is called a state transition. For discrete events the state transition is assumed to happen instantaneous. Some examples of potential discrete events are:

- *Mode changes.* A typical mode change is that the status of a task is updated. For instance start and completion of an operation.
- *Discrete signal changes.* This event occurs when a discrete signal changes from one value to another.
- *Messages.* Messages sent between different systems is also a typical discrete event.
- *State jumps.* A state jump occurs when a state variable makes a jump over/below a certain threshold, causing the jump.

Relating to the automotive industry and an Instrument Cluster we can see that these examples can be similiar to:

- *Mode change of the ECU.* The Instrument Cluster for instance changes mode at start-up and at some Vehicle Mode transitions.

- *Changes in CAN signals.* Changes of signals can trigger notifications or change the state of an application, enabling/disabling parts of the application.
- *Button pushes.* The driver can navigate in the Human Machine Interface using buttons and manually change the state of the system.

2.2.2 Automata Theory

A finite state automaton A is a theoretic way of describing a discrete event system [24]. A finite deterministic state automaton is mathematically described as a 4–tuple

$$A = \langle Q, \Sigma, \delta, q_i \rangle \tag{2.4}$$

where

- $Q = \{q_1, q_2, \ldots, q_n\}$ is the finite set of states of size n.
- $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$ is the finite set of events of the automaton. This is usually called the alphabet of the automaton.
- $\delta : Q \times \Sigma \to Q$ is the transition function, describing the transition from a state based on the event occuring.
- q_i is the initial state of the automaton.

A quite simple example of an automaton can be seen in Figure 2.1. As seen the automaton has an initial state indicated by an arrow. There is also a marked state indicated by a double circle. Marked states are desired states that are wanted to be reached. The marked automaton A can instead be described by a 5–tuple

$$A = \langle Q, \Sigma, \delta, q_i, Q_m \rangle \tag{2.5}$$

where Q_m is a set of the marked states of the automaton.

The discrete events available in the automaton in Figure 2.1 is a, b, c, d, e, f and g, i.e. the alphabet is $\Sigma = \{a, b, c, d, e, f, g\}$. If some events are assumed to be executed in a specific sequence this can be denoted as a *string* of events. All possible strings of events in an automaton A forms a set of strings called the language $L(A)$. An empty string, i.e. no event has occured is denoted by ϵ. For our example this means that the language is

$$L(A) = \{\epsilon, a, ag^*, ag^*b, ag^*d, ag^*de, ag^*def, c, ce, cef\} \tag{2.6}$$

where g^* means that event g can be performed repeatedly.

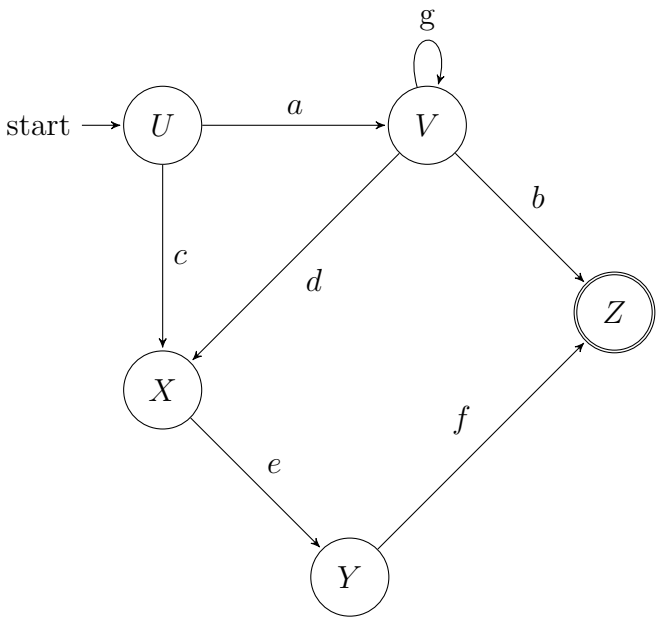

Figure 2.1: An example of a simple automaton. The automaton has an initial state U and a marked state Z. As seen there are several sequences of events that ends up in the marked state Z.

An automaton can have a desired sequence of events. As seen in the example in Figure 2.1 there are several sequences of events leading to the marked state. Imagine that to execute the process described by the automaton most efficiently, it could for example be unwanted to have the event g repeated or having to perform two sequential events ad to reach state X, instead of just performing event c. Then there would be two desired sequences of events to reach the marked state Z, namely ab and cef.

3

HMI Logging Context and Background

This chapter will describe the parts of the truck and HMI that was used for the development of the logging setup. To understand how the logging setup is implemented it is required to have knowledge of the components involved and how they interact, both the Instrument Cluster itself and the components providing the Instrument Cluster with data and functionality used for the logging. Apart from the components involved, there are also some fundamental vehicle concepts that need to be understood to understand the implementation and to reason about how data can be logged in the best way. Finally, for the reader to know what logging that is implemented, the design of the HMI is described with high focus on the logged features.

3.1 Truck Hardware

The following subsection will present the hardware in the trucks used throughout the project.

3.1.1 Instrument Cluster

A dynamic Instrument Cluster, seen in Figure 3.1, has been developed for the Volvo Trucks. The Instrument Cluster consists of two electronic control units (ECU): one back–end ECU called Vehicle Processor (VP) and one front–end ECU called Graphics (GFX). The HMI layer in the front–end ECU is written in QML and C++ and the ECU is running in a Linux environment. The back–end ECU is running in an AUTOSAR environment with the application layer written in C.

3.1.2 Secondary Information Display

The Infotainment system comes in three different variants: Base, Mid and High. In the high variant the Instrument Cluster is accompanied by a Secondary Infor-

Figure 3.1: The hardware of the Instrument Cluster developed by Volvo Trucks that will be used for logging of user interaction in the thesis.

mation Display (SID), see Figure 3.2. In this case a lot of features are moved from the Instrument Cluster to the SID, only the most critical information is kept available in the Instrument Cluster. The SID has a touch display, but can similar to the Instrument Cluster be controlled through buttons on the steering wheel. Which of the two displays that at the moment should be controlled by the buttons can be changed using a Focus Shift function.

Figure 3.2: Picture of the Secondary Information Display. The display has touch functionality, but can also be controlled by the steering wheel switch buttons.

3.1.3 Buttons

The HMI in the Instrument Cluster can be controlled using the steering wheel switch buttons, see Figure 3.3. The buttons are connected to the Instrument Cluster via Local Interconnect Network (LIN) nodes. Button push signals are handled by the AUTOSAR component in the Instrument Cluster and sent towards the front–end ECU, where the button pushes are used to navigate or perform tasks in the HMI. Buttons on the right hand side are used for navigation in the HMI and phone functionality, while the buttons on the left hand side are connected to media functions and Cruise Control (CC)/Adaptive Cruise Control (ACC). A list of all the buttons on the steering wheel switch can be seen in Table 3.1.

Button	Position	Usage
Ok/Enter	RHS	Enter app from menu or acknowledge message
Up navigation	RHS	Navigate up in HMI
Down navigation	RHS	Navigate down in HMI
Right navigation	RHS	Navigate right in HMI
Left navigation	RHS	Navigate left in HMI
Menu	RHS	Open the menu in HMI
Escape/Back	RHS	Go back in HMI
Home	RHS	Return to Home view in HMI
Focus Shift	RHS	Manually change display focus
Green phone	RHS	Answer call
Red phone	RHS	Hang up call
Push To Talk	RHS	Voice command
Activate CC/ACC	LHS	Activate CC/ACC
Set CC/ACC	LHS	Set speed for CC/ACC
Resume CC/ACC	LHS	Resume a paused CC/ACC
Pause/Off	LHS	Pause CC/ACC
Increase CC	LHS	Increase set speed for CC
Decrease CC	LHS	Decrease set speed for CC
Time Gap	LHS	Enter ACC time gap setting view
Downhill Cruise Control	LHS	Activate Downhill Cruise Control
Eco Settings	LHS	Change CC Eco Setting
Volume up	LHS	Increase media sound
Volume down	LHS	Decrease media sound
Mute	LHS	Mute media sound

Table 3.1: List of buttons on the steering wheel switches. The position of a button is either Left Hand Side (LHS) or Right Hand Side (RHS). The third column describes how the button is used.

Figure 3.3: Picture of the steering wheel switches (SWS) in a Volvo Truck. On the left hand side buttons connected to CC/ACC functionality and volume control are placed. On the right hand side are buttons for navigation inside the HMI and phone functionality.

3.1.4 Digital Tachograph

Within the truck industry a Digital Tachograph, see Figure 3.4, is used to track drive and rest times for the drivers to make sure they fulfill the drive time regulations. The driver is identified through a driver card, see Figure 3.5, inserted into the Digital Tachograph. Drive and rest times data are stored on the driver card. The Digital Tachograph sends information to the Instrument Cluster about which driver card that is inserted at the moment. The driver card information can be used to connect data to different drivers. This signalling can hence also be used for logging purposes.

3.1.5 Onboard Diagnostic Contact

In the truck there is an Onboard Diagnostic Contact (OBD Contact), which can be connected to for reading information from the truck. In a left hand drive Volvo Truck the OBD contact is placed close to where the driver would typically place their left foot while driving. By connecting to the contact with OBD tools, information from the truck can be read per ECU. This includes data like parameters, Diagnostic Object values and fault codes.

Figure 3.4: Picture of a Digital Tachograph. The driver card is inserted into the card holder to have drive time data logged for the driver. The tachograph has two card slots: the first slot is for the driver and slot two is to be used by a co-driver.

Figure 3.5: Picture of a Swedish driver card. The driver card is inserted into the Digital Tachograph to have drive and rest time data logged for the driver. Driver card information can be used for logging purposes to identify different drivers of a truck.

3.2 Fundamental Vehicle Concepts

This subsection will present some fundamental concepts used in the trucks that are needed to understand how the logging is implemented.

3.2.1 Vehicle Modes

To describe the current state of the truck, the Vehicle Mode concept is fundamental. The Vehicle Mode is one of the concepts used to determine what parts of the software and ECUs in the truck that should be awake. The Vehicle Mode is changed by the driver via the key positions. Additionally, there is timer handling to lower the Vehicle Mode and put the truck to sleep when left unused. A list of the Vehicle Modes can be seen in Table 3.2.

Vehicle Mode	Value	Comment
Hibernate	0	Most of the truck turned off. Mode used in production.
Parked	1	Most of the truck turned off.
Living	2	Key inserted or truck recently used.
Accessory	3	Key in Position 1.
PreRunning	4	Key is in driving position.
Cranking	5	The engine is being cranked.
Running	6	The engine is running.

Table 3.2: List of the Vehicle Modes in the trucks. The Vehicle Modes are one of the concepts used to determine what parts of the systems in the trucks that shall be active. The display in the Instrument Cluster can be lit from Vehicle Mode Living and up to Running. Most functionality is active from Vehicle Mode PreRunning and up. Therefore, Living and Accessory will be referred to as lower Vehicle Modes, while PreRunning, Cranking and Running will be referred to as higher Vehicle Modes.

In the lower Vehicle Modes only some critical services in the truck are still active, while in Vehicle Mode PreRunning or higher most of the software in the truck is executed. In PreRunning or higher the display in the Instrument Cluster is always lit, with most features available. The display in the Instrument Cluster will light up for a limited time from Vehicle Mode Living and higher from some specific events with a limited HMI available, but most of the time it will be turned off in lower Vehicle Modes. However, more features are availble in the Secondary Information Display in the lower Vehicle Modes.

Since the HMI in the Instrument Cluster is limited in Vehicle Modes below PreRunning, the logging of user data in the Instrument Cluster is focused at

interactions in higher Vehicle Modes. All the parts logged in the scope of this thesis are only available in Vehicle Modes PreRunning and higher.

3.2.2 CAN, LIN and Ethernet

Communication to the Instrument Cluster is handled over CAN (Controller Area Network), LIN (Local Interconnect Network) and Ethernet. A number of LIN nodes are connected to the Instrument Cluster for handling of button pushes. For instance the steering wheel switches are connected to the Instrument Cluster via LIN nodes. CAN is the main communication protocol between different ECUs in the truck, while the Instrument Cluster mainly uses Ethernet for communication with back office and the Secondary Information Display.

3.2.3 AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) is a partnership within the automotive industry to create a standardized software architecture for ECUs within automotive development [25]. Some of the visions of AUTOSAR is scalability, transferability of software and safety.

The architecture in AUTOSAR is made up of three layers:

- Basic software. Includes standardized software modules with core functionality needed to run the other parts of the ECU.
- Runtime Environment (RTE). The middleware handling information exchange between ECUs and between the Basic and Application layers within an ECU. The RTE uses information from the network topology.
- Application layer. Consists of application software components (SWC) interacting with the RTE. Most of the functional logic is placed in this layer.

An overview of the architecture can be seen in Figure 3.6.

3.2.4 APX

APX (AUTOSAR Port eXchange) is an open source software technology developed to enable AUTOSAR components to communicate with non-AUTOSAR components [26]. In the Instrument Cluster there are three APX clients used for communication. One is located in the backend AUTOSAR component, the second one is connected to the HMI layer in the GFX and the third one is placed in the GFX platform. These APX clients allow signals to be sent between all these three parts of the Instrument Cluster. In practice this means that signals can be sent both from the Vehicle Processor and the HMI layer to the GFX platform.

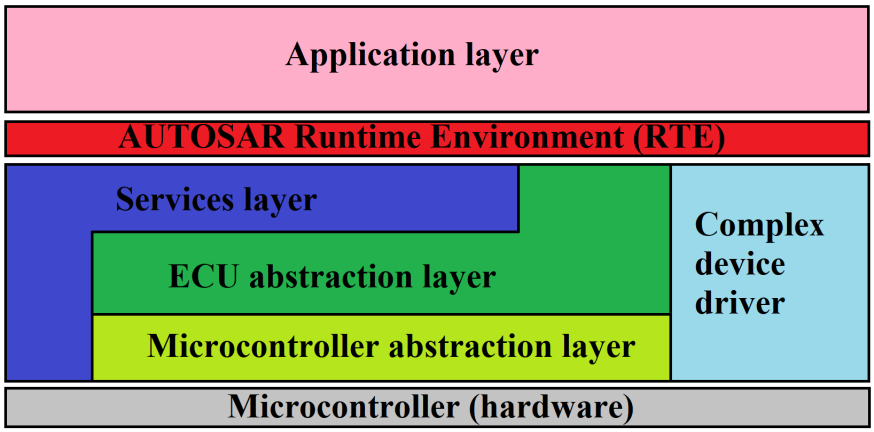

Figure 3.6: An overview of the AUTOSAR architecture. In short AUTOSAR consists of three layers: the Application layer where most of the functional logic is placed, RTE handling information exchange between the different layers and towards other ECUs and the Basic Software including core functionality needed to run the ECU. The Basic Software layer can be split into Services layer, ECU abstraction layer, Microcontroller abstraction layer and Complex device drivers.

3.2.5 HMI Architecture – Model-View-Controller

The Human Machine Interface implementation is based on the Model-View-Controller (MVC) design pattern. The MVC architecture was first described by Trygve Reenskaug [27] and these days an article by Derek Greer is used for reference at Volvo GTT [28].

The purpose of the MVC design pattern is to separate the concerns of the different components in the system. A graphical representation of the MVC design pattern can be seen in Figure 3.7. The responisibilites of the different components are as follows:

- **Model** - Handles the core logic of the application.
- **View** - Graphical representation of the information provided from the model.
- **Controller** - Collects inputs to request updates in the Model or the View.

In the adaptation of MVC used at Volvo GTT the Model is usually referred to as Core. Inside the Core component most of the logic is kept. The *what* and *when* conditions for presentation is handled by a Feedback component. *What* Control Services that should be offered and *when* those are available is handled by a Control component. To determine *how* and *where* to render feedback data Presentation Objects are defined. To perform user requests and present current status, physical

user input/output devices can also be used. A graphical description of the HMI architecture can be seen in Figure 3.7. The components can be placed in different parts of the truck. They do not all need to be placed inside the same ECU. The most common software design is that the Control and Feedback components are placed inside the VP in the Instrument Cluster, although they can be placed elsewhere. The Core component can be placed in any ECU, usually depending on the placement of the sensors, etc that are needed by the function.

Based on the HMI architecture it is clear that different types of data is available in different parts of the system. If user requests are studied they are best collected from the Control component or the individual components providing the requests to the Control component, as there can be multiple sources providing the same Control Services. If the current status of a function is of interest this data is preferably collected from the Core or Feedback components. Some more general HMI concepts like Instrument Views, Gauges and Menu position however are handled directly in the HMI layer and is not known by the Control, Feedback and Core components on the AUTOSAR side, i.e. it is handled as Presentation Objects in the HMI layer.

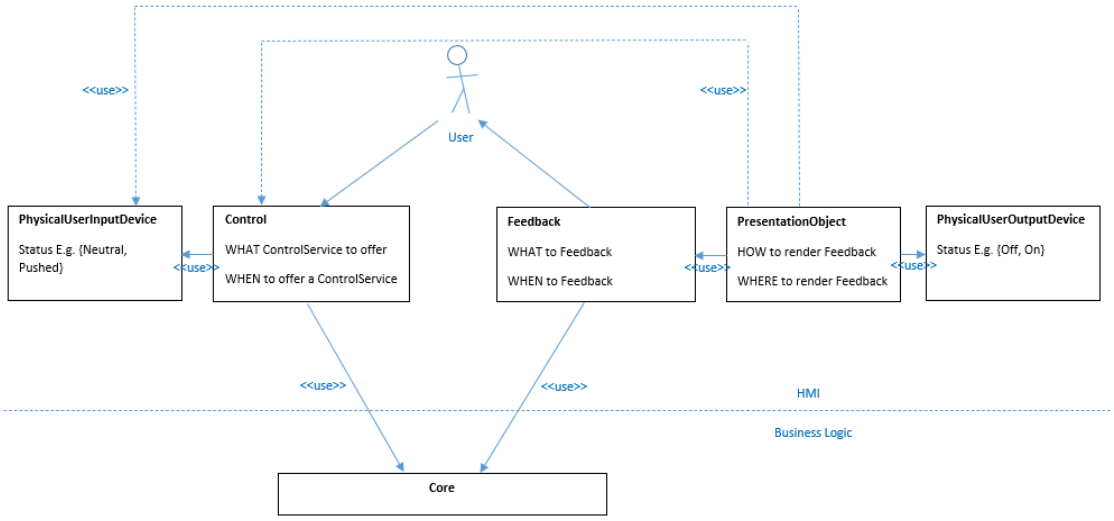

Figure 3.7: Representation of the the design pattern used for the HMI implementation. The architecture is based on the Model-View-Controller design pattern. The architecture consists of three main components. Core holding most of the logic of a function, Control handling the different Control Services, i.e. requests to update status in Core and Feedback handling What and When to feedback data to the user. Presentation Objects are used to describe where and how to render the user feedback and physical input/output devices can also be used to give requests and present current status.

3.3 HMI in the Instrument Cluster

The following subsection will describe the different parts of the graphical HMI within the Instrument Cluster.

3.3.1 Instrument Views

The HMI in the Instrument Cluster consists of four different main views, which are usually referred to as the Instrument Views. The Instrument Views are available from Vehicle Mode Pre–Running or higher and one of the Instrument Views is then always shown. The driver can easily switch between the different views by pushing the buttons Up or Down, while no other menu or application is open. The Instrument Views have different looks and content to suit different driver needs. The available Instrument Views are the following:

- Analog View – This view consists of a speedometer meant to look like a classic mechanical speedometer from the old non-digital Instrument Clusters. Also fuel and AdBlue gauges are meant to look more like classic gauges. Inside the speedometer there are a number of gauges that the driver can choose between, presenting the status of different systems in the truck. The Analog View can be seen in Figure 3.8.
- Focus View – In the Focus View (sometimes also referred to as the Digital View) the speed, fuel level and AdBlue level are presented in digital form. More data on drive times is also presented in this view. The Focus View can be seen in Figure 3.9.
- Load Indicator View – The Load Indicator View presents information about the load on the different axles of the truck and trailers. This view also presents speed and consumtion values in digital numbers. The Load Indicator View can be seen in Figure 3.10.
- Navigation View – In Navigation View the driver can see a map with directions to a set location. This view also presents speed and consumption values in digital numbers. A snapshot of the Navigation View can be seen in Figure 3.11.

3.3.2 Gauges

In the Analog Instrument View a number of gauges can be found inside the speedometer, see Figure 3.12. One gauge is displayed at a time and the driver

Figure 3.8: Picture of the Analog Instrument View taken in HMI simulation tool. The Analog View is meant to be reminiscent of a classic mechanical Instrument Cluster with an analog gauge.

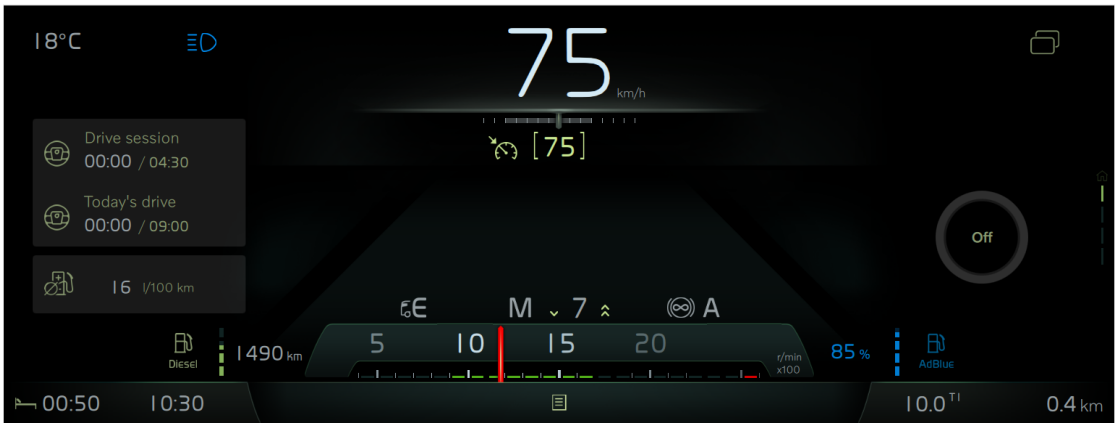

Figure 3.9: Picture of the Focus Instrument View taken in HMI simulation tool.

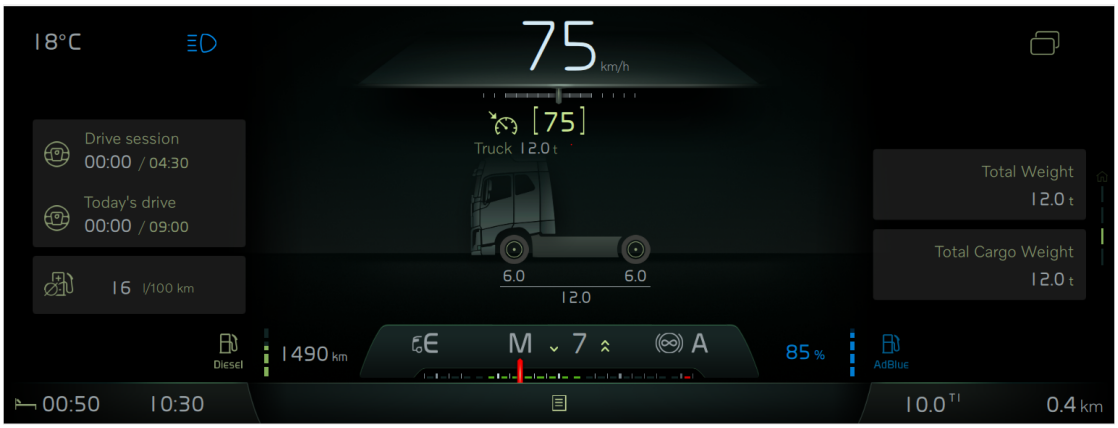

Figure 3.10: Picture of the Load Indicator Instrument View taken in HMI simulation tool.

Figure 3.11: Picture of the Navigation Instrument View taken in a truck.

can change which gauge that is shown with the left and right buttons on the steering wheel switches. A complete list of all currently available gauges can be seen in Table 3.3.

Gauge	Logging signal value	Remark
Engine Oil Temperature	0	–
Air Pressure Monitor	1	–
Engine Coolant Temperature	2	–
Engine Oil Level	3	–
Engine Oil Pressure	4	–
Battery Digital Combined	5	–
Battery Living	6	–
Battery Start	7	–
Gearbox Oil Level	8	–
Gearbox Oil Temperature	9	–
Battery Digital Volt	10	–
Turbo Boost Pressure	11	–
Body Builder One	12	–
Body Builder Two	13	–
Body Builder Three	14	–
Speed	15	Digital speed displayed
Empty	16	No information displayed

Table 3.3: List of all currently available gauges in the Instrument Cluster HMI.

3.3.3 Menus

The menu, see Figure 3.13, can easily be opened by a push on the Menu button on the steering wheel switch. The menu items available in the Instrument Cluster depends on the configuration of the truck. When a Secondary Information Display is installed some of the menu items are moved to that device, this includes the Settings and Vehicle tabs. The user can navigate between the tabs with the left and right buttons and scroll up and down under a tab with the up and down buttons.

3.3.4 Applications

Through the menus in the Instrument Cluster a number of applications can be reached. The applications are designed with the idea to be easily accessible, understandable and offer flexibility for the user.

Figure 3.12: Example of two of the gauges available in the Instrument Cluster. In the upper picture the Engine Oil Temperature Gauge can be seen inside the speedometer and in the picture below the Turbo Boost Pressure Gauge can be seen. In total there are up to 17 different gauges available in the HMI, depending on the configuration of the truck.

Figure 3.13: Picture of the menu in the Instrument Cluster. Seen here is two of the main headers: Vehicle and Maintenance. The user can navigate between the headers with the left and right buttons and scroll up and down under a header with the up and down buttons.

One of the applications found under the Maintenance tab in the menu is Pre-Trip Inspection (PTI). This function offers the user the possibility to get a status report of the truck at start-up. Any issues found by the function is listed in a report and the checks performed include for instance exterior lights, battery status, fuel and AdBlue levels and tire pressure. When entering the PTI application the user first encounters the last generated PTI report. If scrolling down from the PTI report view, the user reaches the PTI menu options. There, the user can choose how to filter what is shown in the report, generate a new report, enable autotriggered report generation and enable exterior lights checks. An overview of the PTI application can be seen in Figure 3.14.

Figure 3.14: The views found in the Pre-Trip Inspection application in the Instrument Cluster. When entering the application a summary of the latest generated report is displayed, see in the upper picture. By scrolling left or right the driver can get more details on the issues identified by the function as seen in the middle picture. In the bottom picture the Options view for Pre-Trip Inspection is seen.

4
Methods

This chapter describes the methodology used in the thesis. Starting by describing the procedure to understand what type of data that is relevant and how it can change the way HMI development is done. Thereafter follows a description of the different systems and tools used during the thesis and the chapter is concluded with descriptions of the implementation, verification and data collection.

4.1 Interviews with HMI Engineers

To gather information on what data that is of interest and how data-driven development could support HMI development, semi-structured interviews where held with some handpicked HMI Engineers at Volvo Group Trucks Technology involved in Instrument Cluster development. In total nine interviews were held. A list of the people participating in the interviews can be seen in Table 4.1.

Interviewee	Role description	Automotive experience
Interviewee 1	HMI Specification Engineer	4 years
Interviewee 2	HMI Specification Engineer	4 years
Interviewee 3	UX Design Engineer	5 years
Interviewee 4	UX Design Engineer	2 years
Interviewee 5	Product Owner Instrument Cluster	9 years
Interviewee 6	Product Owner Instrument Cluster	9 years
Interviewee 7	Driver Interface Feature Leader	12 years
Interviewee 8	Feature Specialist Driver Environment	21 years
Interviewee 9	Human Factors and Automation Specialist	19 years

Table 4.1: List of participants in the conducted interviews at Volvo Group Trucks Technology with engineers involved in work with the HMI in the Instrument Cluster.

The interviews were conducted at the Volvo Group Trucks Technology office between September 28 and October 2 2020 with one interviewee at the time. In to-

tal five main questions were asked at the interviews with some follow-up questions to keep an open dialogue and get out more details. Not all follow-up questions were used each time, depending on the direction the previous answers were taking. The interviews were transcribed live on screen during the sessions with all notes visible to the participant to make sure all answers were interpreted correct. Each interview ran for approximately 45 minutes. The main questions asked at the interviews were:

1. What is the current role of vehicle field data in the automotive UX development?
2. What are the needs, challenges and concerns in the context of data-driven UX development?
3. What is specific to the automotive UX development and what can be adopted from the non-automotive areas of UX development?
4. How can the automotive UX development benefit from data-driven approach?
5. What type of user behavior information is of interest?

The full questionnaire used at the interviews can be seen in Appendix A.1.

After the interviews the individual answers were analyzed and compared. Matching answers were grouped together and a lot of patterns in the answers could be identified. The result of the interview analysis can be seen in the following subsections. Also some interesting quotes from the interviews were highlighted.

All of the interviewees expressed the interest to keep giving feedback during the development of the project. Progress made was at later stages presented at demos with most of the participants from the interviews participating.

4.1.1 Outcome of HMI Engineer Interviews

Based on the answers from the interviews, it can first of all be concluded that the automotive industry and Volvo Trucks are behind a lot of other industries when it comes to logging user behavior data. Especially logging of data closely connected to the HMI.

Most user behavior evaluations done today at Volvo Trucks are done through clinics where truck drivers are invited to perform certain tasks where their performance is closely monitored. These clinics give lots of relevant data, although they require a lot of resources and are time consuming. The number of participants are low, meaning that if some issue is indicated by only a few users, it can blow up in proportions. Additionally, some believe that there is an issue with the demography of the participants, for instance creating a generation gap for new technologies.

Additionally to the clinics, drive-alongs are also performed where HMI Engineers join truck drivers for a day and record the usage. Outcome of these drive-alongs is often used to define use cases.

There is some data being logged from customer trucks on the road. However, this mostly includes data useful for performance tracking and data to be used at Service Centers. This data is typically logged over CAN. Trucks in test fleet usually have quite extensive logging possibilities, but it is used for the purpose of logging bugs and not user behavior.

All interviewees argued that data-driven development is needed to be truly competitive and can make the company better at prioritizing and putting the resources where it is most needed. HMI development today is based too much on opinions and not enough on facts. By looking at actual user data, stronger arguments can be used and decision making can be simplified. One interviewee expressed her opinion in the following way: ”If we not are data driven we will not create products that work for people”.

However, it was clear from the interviews that data-driven development based on logged user data can not be the complete solution. From quantitative methods like logged user data it is possible to study user and interaction patterns with the HMI, but why the user is acting in the way that is observed neccessarily can not be determined. What the participants instead wanted to point out is how powerful quantitative methods can be when combined with qualitative methods like clinics and driver interviews. The results found from quantitative methods can be used as input for qualitative methods to understand the ”why”.

What also cannot be determined from quantitative methods is features that the user is missing. To really understand the needs of the user a qualitative approach is neccessary. Quantitative and qualitative methods can complement each other very well and be used as an iterative way of working. Based on inputs from qualitative methods, new content can be added or updates done and then the impact on the user can be observed from quantitative methods. This impact can then be further understood from qualitative methods and the development cycle continues.

On the question about the uniqueness of the automative and truck industry, what most explained is that in a vehicle the interaction with a HMI is a secondary task. The primary task in a vehicle is the driving. In for instance a mobile app the interaction with the HMI is the primary task. Not having the HMI as the primary task creates some limitations on the HMI. It is of highest importance that the HMI does not distract the user during driving, meaning that safety needs to be considered when designing the HMI. What typically differentiates the truck from the car industry is that the truck is a work environment, putting more requirements on the driver environment and probably making the user group more narrow.

The interviewees all work very close to the HMI and thought that more de-

tails about the interaction can be gained by looking at data logged from a HMI compared to logging data between ECUs (CAN, Ethernet, etc). Most interviewees believed that the CAN data can give more of an overall system level status, while the HMI data could be more valueable for their own work. One of the participants proposed that maybe the two approaches can be combined to create a really powerful logging setup.

To the interviewees it was important that the data is presented and visualized in a clear and understandable way. Best would be if it is sorted based on functions. One interviewee stood out from the rest, wanting access to the complete raw data. One wish from many participants was to have the data linked to driving conditions. Some expressed concerns about the availability of data and being able to understand the data that is logged.

Another concern that was raised was GDPR. As the interest is to find user patterns and understand different functions and not to study individual drivers those with concerns still thought there will be ways to get around this issue.

The wish from the interviewees was to have logging introduced as early as possible in projects. When starting up a new project it would be interesting to have data from previous projects to base early decisions on. During development logging would also be of great need as issues can be identified early and finally during maintenance phase to fix customer issues and collect data for future projects. However, it was mentioned that basing user analysis on internal testers might not be optimal as they typically will not reflect the general driver.

4.1.2 What to Log Based on HMI Engineer Interviews

At the interviews the participants were asked what information they find most interesting to log. Below is the list of what was most frequently wished for:

- Navigation inside the HMI.
- Focus Shift.
- Instrument Views.
- Time spent in different views.
- Time to complete different tasks.
- What functions that are used.
- Comparison of usage of functions between SID and Instrument Cluster.
- What buttons are pushed the most.

- Where in the HMI is the driver focused with its glance.
- What pop-ups are triggered.

Some of the bullets listed like logging of triggered pop-ups do connect to the Instrument Cluster, but as these are triggered based on inputs describing the status of the vehicle and not triggered by user interaction, it is not considered in this thesis, although an interesting topic to study. Data like where the user is looking is interesting from the perspective of a HMI Engineer, but this type of data can not be retrieved from inside the Instrument Cluster and is therefore not considered.

When going into the interviews there was already quite a clear idea of what is interesting to log and what the possibilities were. The feedback at the interviews only strenghtened the initial thoughts and decision was then to focus implementation on the top items in the above list, namely Navigation inside HMI, Focus Shift, Instrument Views and time spent in different views.

4.2 Software Tools

Below is a description of some software tools that has been used, programming languages, etc.

4.2.1 Programming Languages

Implementation in the Vehicle Processor component of the Instrument Cluster has been done in the language C, with the code executing in an AUTOSAR environment. Also implementation in Python was required as some scripts are used to generate the ARXML files used in AUTOSAR.

To send data from the HMI layer, implementation in QML was done. QML (Qt Modelling Language) is a declarative programming language used for developing user interfaces [29]. The language allows the programmer to describe how visual components relates to each other and how they interact. Some pros with QML is customization and reusability of components. The QML layer is executing with a C++ platform in the background, consisting of most of the HMI logic.

C++ implementation was also needed in the GFX platform where the user interaction data is stored. The logger main functions are all written in C++ with complementing classes describing the structure of how the data is stored and the methods (called from the logger main functions) that are needed to update the data to be stored.

4.2.2 Engineering Tool

To during development read the values of the DOIDs used for storing the logged data the at Volvo GTT inhouse developed program Engineering Tool is used. The program can be used both in rig and truck tests to read the values of all parameters in an ECU. In a rig a VOCOM is needed to connect the Instrument Cluster to the computer and in a truck the computer is connected to the truck via the Onboard Diagnostic Contact. Once connected to the system, parameters can be read in diagnostic sessions.

4.3 Modelling of HMI

Parts of the HMI has been modelled using Discrete Event System theory described in section 2.2. By creating a discrete event model, the possible states and events in the system get identified. This is great support for deciding what and how to log in a certain function.

An automaton describing the handling of Instrument Views can be seen in Figure 4.1. The model describes how the user can navigate between the different Instrument Views using the Up (u) and Down (d) buttons on the steering wheel switch. If Vehicle Mode is lowered to Accessory or Living ($V_\vee$) the HMI with Instrument Views is no longer displayed, but the previous Instrument View is remembered such that the HMI will return to the view if Vehicle Mode is raised again ($V^\wedge$). If truck is turned off the system returns to its initial Off state.

The automaton describing handling of Gauges will be very similar to the automaton for Instrument Views, but much larger as there are up to 17 different gauges, instead of four Instrument Views. The events Up and Down button push will be replaced by button push Left and Right to scroll between the gauges. The gauge used last is remembered just like the Instrument View when the Vehicle Mode is lowered. One more difference is that there is another set of states where the previous gauge will be remembered. If the driver navigates out of the Analog Instrument View no gauge will be displayed. When returning to the Analog Instrument View the previous gauge is displayed again.

The handling of Focus Shift is decribed by the automaton seen in Figure 4.2. From the model it can be seen that manual Focus Shift only is possible in higher Vehicle Modes. Additionally, it is seen that display focus defaults to Instrument Cluster in higher Vehicle Modes and to SID in lower Vehicle Modes.

The original plan was to also create detailed models of the menu applications that would be logged, i.e. Pre-Trip Inspection in this case. With a detailed model of all the states and events the exact navigation inside the application could be tracked, but with the current static way of logging data defined by DOIDs it is

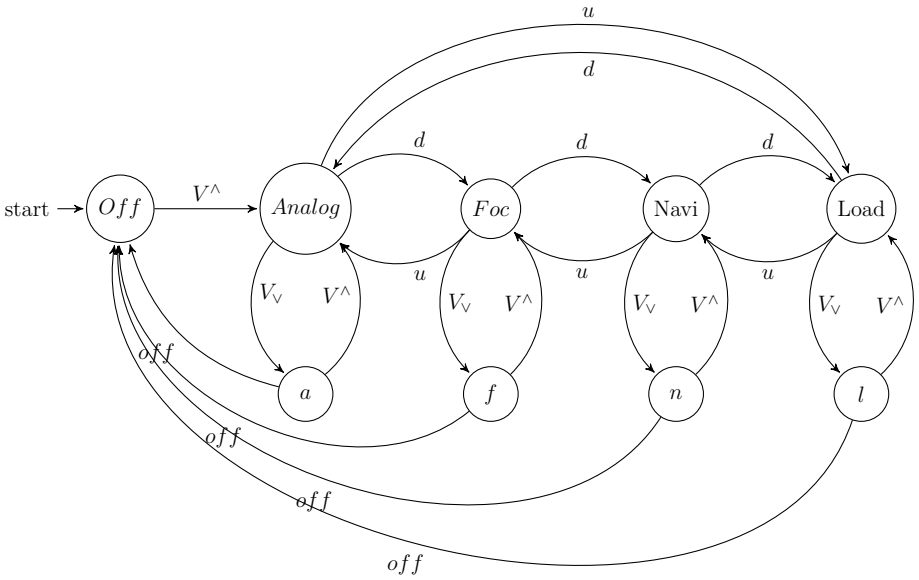

Figure 4.1: Automaton describing the model of Instrument View handling. When the Instrument Cluster is started the driver will end up in the Analog Instrument View. With the Up (u) and Down (d) buttons on the steering wheel switch the driver can scroll between the four different Instrument Views: Analog, Focused, Load Indicator and Navigation. If Vehicle Mode is lowered to Accessory or Living $V_\vee$ the Instrument Views are no longer available, but once Vehicle Mode is raised to Pre–Running or higher again ($V^\wedge$) the HMI will return to the previous Instrument View.

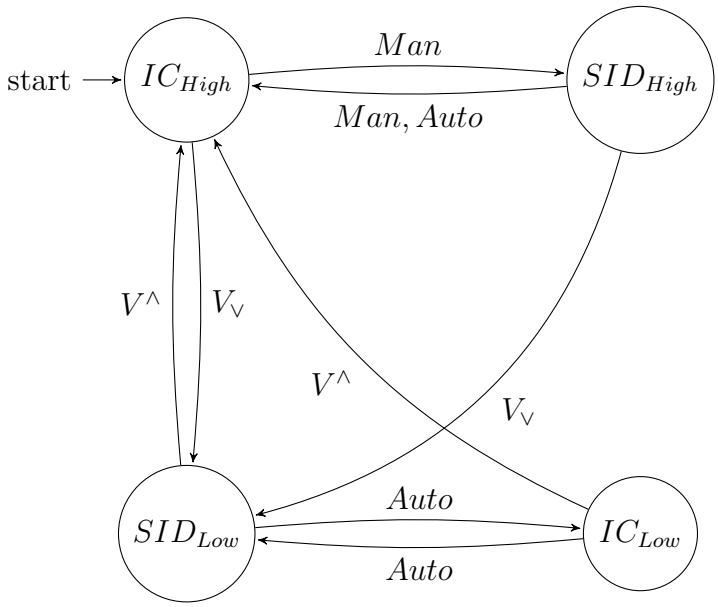

Figure 4.2: Automaton describing the handling of Focus Shift between the Instrument Cluster (IC) and Secondary Information Display (SID). $High$ (Pre-Running or higher) and Low (Accessory or Living) is meant to represent the Vehicle Modes that the states can appear in. Focus can either be changed Manually (Man) by the driver with the Focus Shift button, automatically ($Auto$) by system status updates, Vehicle Mode lowered ($V_{\vee}$) or Vehicle Mode raised ($V^{\wedge}$). From the model it is clear that display focus only can be shifted manually in the higher Vehicle Modes.

not possible to log data on that detailed level.

4.4 Software Implementation of Logging System

This subsection describes the software implementation of the user interaction logging.

4.4.1 Driver Card Identification

Compared to the car industry where it might be hard to differentiate between different users of a car when logging data, there is the possibility within the truck industry to identify different users of a truck based on driver card information. Each driver card has an unique Driver Card Number, which is sent over CAN from the Digital Tachograph to the Instrument Cluster. Inside the AUTOSAR component of the Instrument Cluster, the Driver Card Number is stored for up to ten drivers in different memory blocks. Which of these memory blocks from 1–10 that is utilized is then sent in a signal, that is used in the GFX platform. Additionally there is one memory block used for a default driver when no specific driver can be identified, i.e. no driver card has been correctly inserted to the Digital Tachograph.

If all ten driver unique memory blocks are already in use and an eleventh driver starts using the truck, then the information about the driver that used the truck the longest time ago is replaced by the new driver. A reset of a driver memory is sent in the same signal as used to signal which of the ten memory blocks that is in use.

User behavior data is logged based on the current driver number memory block that is in use. This means that all logged data will be saved in eleven different sets, one for the unidentified driver memory and one for each memory block from 1–10. In this way it can be secured that as long as a driver card is used, data is logged on an individual level. By translating the Driver Card Number to a memory block internally in the Instrument Cluster and not connecting the data directly to the Driver Card Number, one can argue that the data has been somewhat anonymized. Still, there is the need to make sure that the data cannot be connected to a specific truck.

When a reset of a Driver Card Number memory block occurs, the user behavior data connected to that driver memory will also be reset. This is needed to make sure that data for multiple drivers is not saved to the same driver unique data set. The option would be not to use the already available signalling for driver data memory sets. Instead the logged data could be directly connected to the Driver Card Number and make space for even more data sets, but on the other hand it

is important to limit the amount of data. It is assumed that most trucks will not be used by more than ten drivers over a significant time frame.

The C++ implementation in the GFX platform for driver memory handling can be seen in Appendix A.4.2.

4.4.2 Diagnostic Objects to Store Data

Data is logged with the help of Diagnostic Object Identifiers (DOIDs). These are a type of parameters constructed for logging purposes and with both read and write access. Data is saved to the memory of the ECU, but the DOID is used to describe how the data is structured. When accessing the data described by the DOID the read and write functions are pointing to the memory position where the data is stored within the ECU.

DOIDs has to be defined prior to implementation and how the data in the DOID is structured has to be clearly stated, element by element. Before using a DOID it has to pass a review process to ensure it is constructed in a correct and understandable way. Once a DOID has been approved and implemented it is hard to have it updated because of backwards compatibility reasons.

How the DOIDs used for logging user behavior data in this project was defined can be seen in Appendix A.2.

4.4.3 Logging of Time Intervals

For logging of time spent in different parts of the HMI or time spent on different tasks the UTC time is used as the basis. The UTC time is sent over CAN to the Vehicle Processor ECU and from there sent to the GFX over APX. When an event to enter a state is triggered the current UTC time is stored in the logging implementation. Later when the state is exited, the new UTC time is also saved. By calculating the difference between the two UTC times the time spent in a state is determined. This calculated time difference is then added to the summed up total time.

For the function implemented to check the difference between two UTC times the C Time Library in C++ was used [30], the code is found in Appendix A.4.1.5. The UTC times received from CAN is sent to a function that rewrites the times to the struct tm format used by the library. Then the function mktime is used to convert both times into number of seconds since January 1st 1900 and the function difftime is used to get the total difference in seconds. One limitation of the mktime function was found during the verification of the time logging. In 31-bit the function only supports calculations up to year 2037. Larger inputs will cause the function to return -1 and no time difference can be determined.

4.4.4 Logging of Menu Applications

In the QML layer a Page Stack keeps track of what view in the HMI that is currently visible. To determine if a menu application is open, data from the Page Stack was used. A logging signal was introduced such that if the Page Stack value matches a menu application the signal is set to a value corresponding to that specific application. The signal is sent from HMI layer to the GFX platform.

For now only the currently logged menu applications are translated to this logging signal, but the signal and the implementation is designed to easily have more logged applications added. The signal size supports more values to be added and the logic consists of an if-statement that can be adjusted.

4.4.5 Logging of Instrument Views

For logging of Instrument Views the following attributes are logged per user of the truck:

- Counter of amount of times each Instrument View has been entered.
- Total amount of time spent in each Instrument View.

This information can be used to calculate the mean time spent in each Instrument View. In the current implementation four different Instrument Views are available, but the DOID has support for a total of 15 different Instrument Views. This is needed to easily support potential future updates based on needs seen when logging user data. The complete defintion of the DOID can be seen in Appendix A.2.1.

To keep track of what Instrument View that is shown a new logging signal sent from the HMI to the GFX platform was introduced. Each Instrument View has an ID assigned and the logging signal is set based on the ID of the currently displayed Instrument View. This signal value is then used to determine which index in the array of the DOID that shall be used to store the data. For instance the Load Indicator View has ID 5 and therefore Load Indicator values are stored in column index 5 of the DOID.

Since multiple Instrument Views can be available in the Instrument Cluster, the driver might have to browse past some Instrument Views to reach the wanted one. To not count when an Instrument View is quickly browsed past, a timer is implemented to have a minimum time an Instrument View has to be open before it is counted. The length of this timer was determined to 5 seconds. Many drivers will most likely navigate faster than this in the HMI. However, it is possible that a few users will navigate to one Instrument View for only a few seconds to see one specific set of information, to then leave the view immediately. Still, the decision was taken that it is more important to have a robust solution and not risk logging any false data, as there probably are some users who are navigating slower.

The C++ implementation in the GFX platform for logging of Instrument Views can be seen in Appendix A.4.3.

4.4.6 Logging of Focus Shift and Button Pushes

For logging of Focus Shift the following attributes are logged per user of the truck:

- Total time spent with Focus on Instrument Cluster.
- Total time spent with Focus on SID.
- Counter of total amounts of button pushes while display focus is on Instrument Cluster.
- Counter of total amounts of button pushes while display focus is on SID.
- Counter of activations of Focus Shift while the vehicle is standstill.
- Counter of activations of Focus Shift while the vehicle is moving.

A more detailed description of the DOID can be seen in Appendix A.2.2.

In lower Vehicle Modes display focus is automatically set to the Secondary Information Display. Since what is of relevance is the user behavior these Vehicle Modes are not considered. Data is only logged in Vehicle Modes where the manual Focus Shift functionality is available. Not all buttons on the steering wheel switch are impacted by the Focus Shift. Only buttons that change impact between Instrument Cluster and SID together with the Focus Shift are being counted. What buttons that are considered for Focus Shift logging can be seen in Table 4.2.

To determine the reason for why a Focus Shift is triggered a new logging signal ”FocusShiftActivationTriggerLog” was introduced. The signal is sent from the SWC in the VP responsible for handling the core logic for the display focus. The signal tells whether a Focus Shift was manually or automatically triggered and if the shift was from Instrument Cluster to SID or vice versa. As only manual Focus Shifts to SID are counted a clear way to distinct between reasons for a Focus Shift was needed. To keep track of the current display focus the already available signal used to distribute the focus was used.

The C++ files used in the GFX platform for logging of Focus Shift can be seen in Appendix A.4.4.

4.4.7 Logging of Pre-Trip Inspection HMI

For the Pre-Trip Inspection HMI the following attributes are logged:

Button	Position	Focus Shift Impact	Button Push Counted
Ok/Enter	RHS	Yes	Yes
Up navigation	RHS	Yes	Yes
Down navigation	RHS	Yes	Yes
Right navigation	RHS	Yes	Yes
Left navigation	RHS	Yes	Yes
Menu	RHS	Yes	Yes
Escape/Back	RHS	Yes	Yes
Home	RHS	Yes	Yes
Focus Shift	RHS	Activation	–
Green phone	RHS	No	No
Red phone	RHS	No	No
Push To Talk	RHS	No	No
Activate CC/ACC	LHS	No	No
Set CC/ACC	LHS	No	No
Resume CC/ACC	LHS	No	No
Pause/Off	LHS	No	No
Increase CC	LHS	No	No
Decrease CC	LHS	No	No
Time Gap	LHS	No	No
Downhill Cruise Control	LHS	No	No
Eco Settings	LHS	No	No
Volume up	LHS	No	No
Volume down	LHS	No	No
Mute	LHS	No	No

Table 4.2: List of buttons on the steering wheel switches. The position of a button is either Left Hand Side (LHS) or Right Hand Side (RHS). Third column describes if the button is impacted by Focus Shift, i.e. if the display it impacts moves along with the focus. Fourth column tells if the button is counted or not in the Focus Shift and button push logging.

- Counter of amount of times the application is entered while the vehicle is standstill.
- Counter of amount of times the application is entered while the vehicle is moving.
- Total time spent inside the application.
- Counter of manual report generations for Pre-Trip Inspection.

- Counter of amount of times the report filter option has been changed.
- Counter of amount of times the setting for autogenerated reports has been changed.
- Counter of amount of times the setting for Exterior Light Checks has been changed.
- Current setting for report filtering.
- Current setting for autogenerated reports.
- Current setting for exterior light checks.

A more detailed description of the DOID can be found in Appendix A.2.3.

To decide if the Pre-Trip Inspection menu application is in use the menu application logger signal described in Subsection 4.4.4 is used. Requests to update settings are tracked by checking the requests signals sent from the HMI, while the current settings are based on the presentation signals sent from the feedback SWC for the Pre-Trip Inspection function. The C++ implementation in the GFX platform for logging of Focus Shift is found in Appendix A.4.5.

4.4.8 Logging of Gauges

For gauges the following attributes are logged for each gauge.

- Counter of amount of times each gauge has been entered.
- Total amount of time spent with each gauge shown.

To determine which gauge that is currently shown in the HMI a new logging signal sent from the HMI to the GFX platform was created. Most HMI objects has a Unique ID in the QML layer and by checking the current active Unique ID the gauge shown is determined. The values in the logging signal corresponding to the different gauges can be seen in Table 3.3. The signal value corresponds to the array index in the DOID for the gauge.

The DOID defined for logging of gauges has support for up to 30 gauges, although only up to 17 are available in the HMI at the moment. More detailed definition of the DOID can be seen in Appendix A.2.4.

Similar to the Instrument View logging, a timer is defined to check if the gauge is shown long enough to be considered in use by the driver. The length of this timer was set to 5 seconds. If the timer has not elapsed, the gauge usage will not be counted and the time spent is not added to the total. As the driver quickly can scroll between the different gauges using the steering wheel switches, it is

important not to count usage of a gauge that is only passed while the driver is looking for another gauge.

4.4.9 Scheduling

In the first iteration of the implementation, the main functions of the loggers in the GFX platform were scheduled to execute every 5 ms. This was done out of simplicity as most other functions running in the platform are executing with the same frequency. As user interactions are nowhere near that fast, it was realized that the frequency could be lowered. Instead a periodicity of 100 ms was implemented, with no loss of functionality. Reducing the frequency is important from a performance point of view on component level. To reduce the load even more, the logger functions were scheduled to not execute at the same time. Instead the triggering of the functions was spread out in time.

4.5 Verification

This section describes the verification process of the logging setup, all the way from unit tests of the code, to system tests using actual hardware. The testing strategy used can be linked to the classic V-Model for verification [31], seen in Figure 4.3. In the first level of testing, often referred to as V1, unit tests are performed to test the smallest possible entities. This is made to verify that the smallest entities are functioning as expected when isolated. In practice, this usually means testing of isolated functions or parts of the code. Component test, called V3, is carried out to verify that the implemented solution is working according to specification on an isolated ECU. System test (V5) is performed with the solution integrated into the complete system, which can consist of multiple ECUs.

4.5.1 Unit Tests

Implementation in the application layer of the AUTOSAR ECU of the Instrument Cluster is tested using the CUnit framework. Tests of the HMI implementation are performed using an inhouse developed tool at Volvo GTT called GESTI. Unit tests written for the platform part of the GFX ECU are written in Google Tests. The Google Tests written for Focus Shift logging are presented in Appendix A.4.6.

All unit tests are run in a continuous integration flow for each new commit added to a branch, to protect against that any future updates do not break the existing implementation. A condition for merging to the master branch is that all unit tests pass. Any failing unit test renders a fail in the Jenkins build system, where a passing build is a condition for merging.

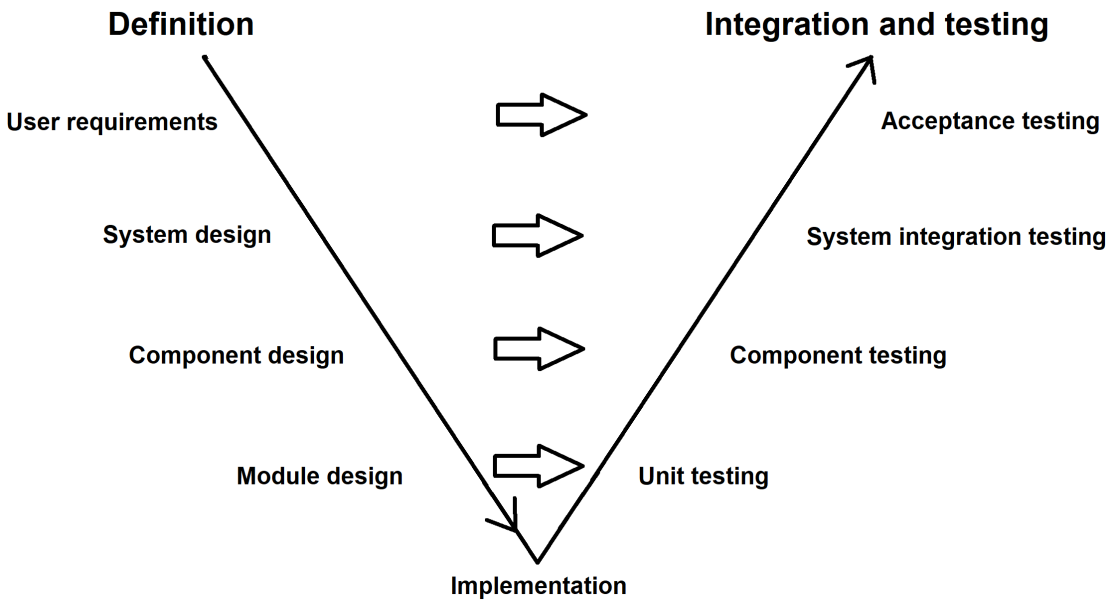

Figure 4.3: Picture of the V-Model explaining the usual steps of verification during software development. The V-Model is often criticized as a waterfall way of working, explaining the requirement and design phases as the first steps on the left side with testing on the right side performed after implementation. The idea is to start with requirements on a high level and then go more and more detailed, while testing starts on a low level and progresses to higher and higher levels. In an agile setup it is preferable to do development in smaller iterations, but the same steps of testing are typically still required.

However, it is important to note that a passing unit test does not necessarily ensure that nothing will break based on a new commit. It is a matter of what is covered by the tests. If a corner case is missed by the unit tests, there is no protection that this case will not break by future updates. To minimize the risk that some cases are missed by the tests, tools like requirement and code coverage are used.

For the implementation in the GFX platform, the code coverage of the Google tests was determined to be almost 100% for all the added logger functions and classes. The only none covered lines of code were three identical lines, one in each of the main functions of the Gauge, Pre-Trip Inspection and Instrument View loggers. These lines of code most likely can not be reached. Still, they are kept for a safety measure not to risk missing any initialization when returning from a lower Vehicle Mode without having the GFX restarted inbetween.

4.5.2 Component Test

Component testing is performed with the Instrument Cluster isolated, combining both the VP and GFX ECU. Typically component testing is carried out for only one ECU, but as the Instrument Cluster in this case includes two ECUs in the same hardware, the integration between the ECUs is tested on component level.

Tests were run using the tool CANoe [32]. CANoe is a tool developed by Vector for analysis of network communication, diagnostic communication, running simulations and performing tests. With the help of CANoe inputs on CAN and LIN can be simulated to the Instrument Cluster.

In the component tests user interactions like button pushes were simulated via CANoe. Vehicle Mode changes was simulated to change the state of the Instrument Cluster. Time spent in the different views and applications was tested by simulating the UTC time over CAN.

After simulating different interactions and state changes, the value of the DOIDs could be read using Engineering Tool. With the high control of the system one has when testing on component level, it is easy to verify if the DOIDs are updated as expected between the readouts.

4.5.3 System Test

On system level, tests were performed in system rigs, which are rigs including more or less the full electric system of a truck. This means that all physical HMI components like, for instance, the steering wheel switch and the key module are installed. Tests were performed manually by interacting with the HMI and studying if the DOIDs used for logging were updated as expected. The values of the DOIDs were read using Engineering Tool.

For instance, in the case of logging of Instrument Views, the view can easily be changed using the steering wheel switches. By reading the value of the DOID at the beginning of the test, then interacting with the HMI and keeping track of the interesting events and the time spent with different statuses, the functionality could be verified by rereading the value of the DOID at the end of the test.

4.5.4 Findings and Adaptations During Verification

A few issues and bugs were found during the verification process of the HMI logging. In the first round of component tests, it was thought that the logic to add times was not functioning. In the CANoe simulation the default UTC year was set to 2048. However, it was later discovered that the library used only supports calculations up to year 2037 in the current system. This limitation is still valid, but shall be adressed by a future kernel update.

Another initial bug found in component testing was that data for two of the gauges was not logged. The root cause was identified to be a faulty define value in the code in the GFX platform. The gauges that were impacted was the Speed and Empty gauge, with signal value 15 and 16. The faulty define value was found to block all gauges with signal value 15 or higher. Instead the value should have been 30, to support the full size of the DOID and block any attempts to store data outside the range of the DOID.

Two more tricky bugs was found first in system testing. For logging of Instrument Views, it was in the system rig found that logging was not reinitialized correct when lowering the key position to below Vehicle Mode PreRunning and then turning the key back up again. The issue can be understood by studying the discrete event model of the Instrument View handling in Figure 4.1. As the current Instrument View is remembered in the HMI layer when the Vehicle Mode is lowered, it meant in the first implementation that the value of the logging signal remained unchanged when turning the key. The implementation in the GFX platform includes callback functions for signal handling that only triggers on value updates. As the value did not change, the logging was not reinitialized when going back up in Vehicle Mode again. What was needed to solve this bug was to add a toggling of the Instrument View logging signal in lower Vehicle Modes, i.e. set it to Not Available below Vehicle Mode PreRunning since no Instrument View is then shown.

The second bug found during system testing was related to logging of manual Focus Shift activations. As seen in the automaton describing the Focus Shift handling in Figure 4.2, manual activations can only occur in higher Vehicle Modes. Additionally, it is only Focus Shifts to SID from Instrument Cluster that are counted and this event can only be manually triggered in higher Vehicle Modes. It was therefore assumed that the signal describing the current display focus in

combination with knowledge of the current Vehicle Mode, would be enough to determine if a manual Focus Shift has occurred or not. However, during system verification it was obvious that a lot of false manual activations was counted. Looking again at the automaton, it can be seen that display focus also can be automatically changed from Instrument Cluster to SID when Vehicle Mode is lowered. It was discovered that the falsely counted manual activations was due to that it cannot be ensured that the shift in display focus is signalled before the lowering of Vehicle Mode is signalled to the GFX. This lead to a lot of automatically triggered focus shifts from Instrument Cluster to SID falsely being counted as manual activations. The way to solve this bug was to introduce a new logging signal sent from the core component for Focus Shift handling, keeping track of the triggering event of all Focus Shifts.

4.6 Collection of Data

A small amount of user interaction data was logged from two test trucks in the internal test fleet at Volvo GTT. Parameter readouts were performed in connection to software updates of the trucks. In Truck 1, data was collected between May 5th and May 12th, 2021. This truck had a software version including all four of the DOIDs. In Truck 2, data was collected between April 29th and May 20th, 2021. This truck had an older software version and only included the DOIDs for Instrument Views and Focus Shift logging. To measure the usage during the time period, the difference was calculated from the first to the second readout. Based on the collected data both trucks have been used by two drivers identified by driver card. Additionally, data has been logged to the default memory block, when no driver card is inserted.

Performing user behavior evaluation based on data from test drivers is not ideal as their task is to test certain parts of the system and their usage can not be considered to imitate a normal customer. Still, the collected data can be worked with to understand what to visualize and what potential future analysis may have.

5
Results

The following chapter will present the main results found in the thesis.

5.1 Verification Results

As described in subsection 4.5.4, the DOIDs implemented for logging of user interactions with the Instrument Cluster were verified to be updated as expected after a few iterations and verification loops. Verification was performed all the way starting from unit tests, followed by component tests and finishing with system tests. In Figure 5.1 and 5.2 readouts of the values of the DOIDs for logging of Focus Shift and Pre-Trip Inspection application from a system rig can be seen.

[redacted]	Logs the usage of manual Focus Shift between Instrument Cluster (IC) and Seconda	Display Focus, Focus Shift, Log	
Array			
Array 0			
FocusInstrumentClusterTotalTime		Focus on IC, Total Time	668
FocusSIDTotalTime		Focus on SID, Total Time	45
CounterButtonPressesFocusIC		Focus on IC, Number of Steering Wheel Button Presses	66
CounterButtonPressesFocusSID		Focus on SID, Number of Steering Wheel Button Presses	25
CountActFocusShiftVehStandstill		Focus Shift, Number of Activations at Standstill	1
CountActFocusShiftVehMoving		Focus Shift, Number of Activations while Moving	3
Array 1			
Array 2			

Figure 5.1: Engineering Tool readout of DOID for Focus Shift logging from system rig verification. Here values for the unidentified driver memory, i.e. when no driver card is inserted can be seen. All values in the struct was verified to be updated as expected based on performed user interactons.

This means in practice that it has been proven that it is possible to extract user interaction data from internally inside the Instrument Cluster and log this data in order to perform user interaction analysis. User interaction data was successfully logged for Instrument Views, Gauges, Focus Shift and the Pre-Trip Inspection application. This leads to the possibility of a deeper understanding of the user performance in the HMI.

Code	Description	Caption	Value
[redacted]	Logs data regarding usage of the Pre-Trip Inspection application in the Instrument	Instrument Cluster, Pre-Trip Inspection Usage, Log	
Struct			
PTIApplicationTotalTime		Pre-Trip Inspection Application, Total Time	134
CountPTIAppEnteredStandstill		Pre-Trip Inspection Application, Vehicle Standstill, Number o	2
CountPTIAppEnteredMoving		Pre-Trip Inspection Application, Vehicle Moving, Number of	2
CountPTIManualReportActivations		Pre-Trip Inspection, Manual Report Activations	2
CountPTIReportFilterChanged		Pre-Trip Inspection, Report Filter Show All Setting, Number c	2
CountPTIReportAutogenChanged		Pre-Trip Inspection, Report Autogeneration Setting, Number	4
CountPTIExtLightsCheckChanged		Pre-Trip Inspection, Exterior Lights Check Setting, Number of	1
PTIReportFilterCurSetting		Pre-Trip Inspection, Report Filter Show All, Current Setting	false
PTIReportAutogenCurSetting		Pre-Trip Inspection, Report Autogeneration, Current Setting	true
PTIExtLightsCheckCurSetting		Pre-Trip Inspection, Exterior Lights Check, Current Setting	false
[redacted]	Logs data regarding what gauges that has been displayed in the Instrument Cluste	Instrument Cluster, Gauge Usage, Log	

Figure 5.2: Engineering Tool readout of DOID for Pre-Trip Inspection application logging from system rig verification. All values in the struct was verified to be updated as expected based on performed user interactons.

5.2 Visualization of Logged User Data

Usage data has been logged from two trucks in the internal test fleet at Volvo GTT. Since the users are test drivers, no proper user behavior evaluation can be made based on their data. Still, the data can be analyzed and visualized to show what kind of results could be achieved once proper user data is available on a larger scale. It is also important to remember that the data sets referred to as Unidentified Driver, mean that no driver card was inserted at the time of logging, i.e. the Unidentified Driver data sets may include usage from multiple drivers. Tables of raw data from the data collection can be found in Appendix A.3.

5.2.1 Instrument View Logging

The Instrument Views present different sets of data and how the presentation is done varies between the views. When logging usage of Instrument Views one of the most important questions to answer is:

Which Instrument Views are preferred by the drivers?

Usage time data for the Instrument Views from the two trucks is visualized in Figure 5.3. From the graphs it is seen that in Truck 1 both the Unidentified Driver memory and Driver 1 had most time recorded in the Navi View. For Driver 2 most usage was logged in the Analog View. Both the Load and Focus View had comparatively small amount of logged time in Truck 1. In Truck 2 however, the Analog View was used the most by two data sets, both the Unidentified Driver memory and Driver 4. Driver 3 has a very limited total amount of time logged,

therefore the usage is hard to compare.

With a large sample of customer data available more clear patterns on driver preferences can be found. Furthermore, the identified patterns can be used to ask a subset of users why they have their preferences.

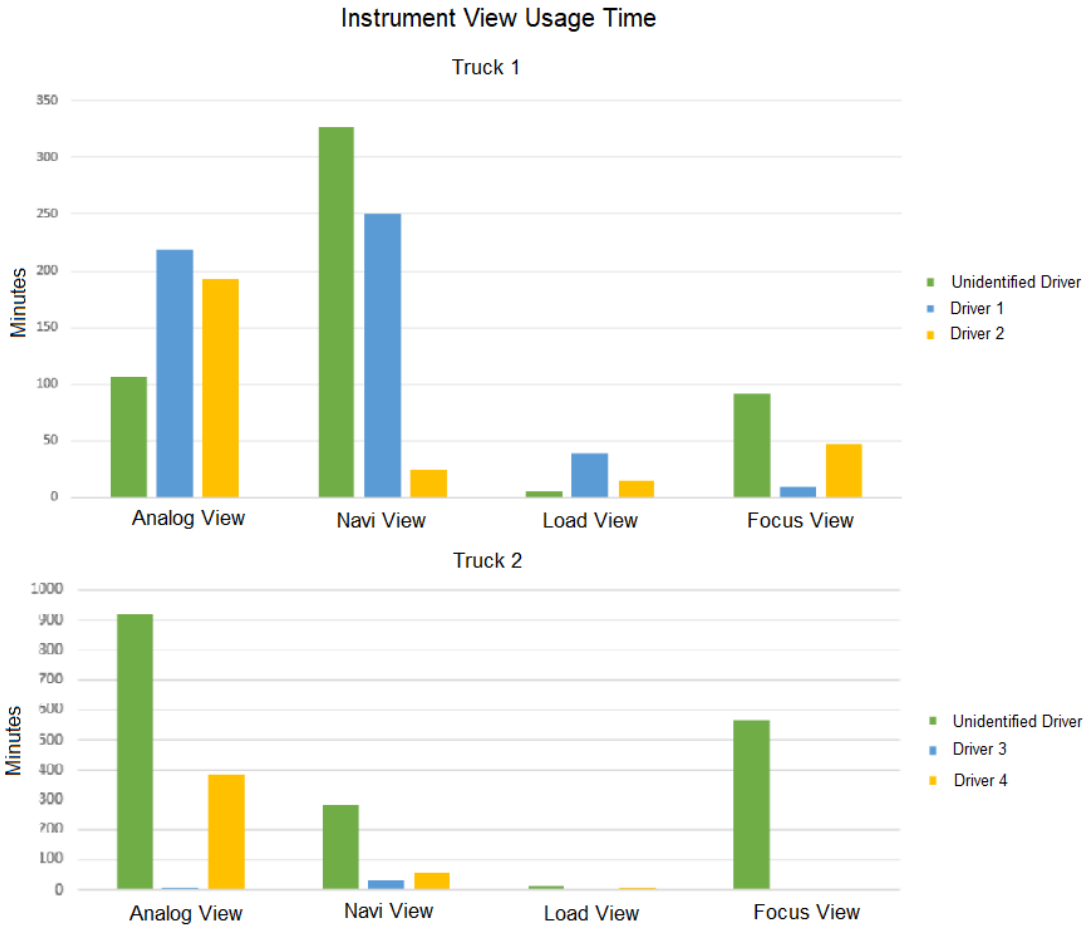

Figure 5.3: Instrument Views usage time from two trucks in the test fleet.

5.2.2 Focus Shift Logging

When studying the Focus Shift function one of the main questions engineers want to know is:

How much is the Focus Shift function utilized by the drivers?

To answer this question data on total time per display focus, number of man-

ual Focus Shift activations and number of button pushes towards each display is needed. The total time per display focus indicates the usage of Focus Shift since display focus by default is assigned to the Instrument Cluster. The number of button pushes towards each display tells how much the driver has navigated inside each display using the steering wheel switch buttons. Finally, the number of manual activations gives the amount of times the driver has taken the conscious decision to move focus to SID. All these parameters combined yield how often the function has been utilized and the mean amount of usage per activation.

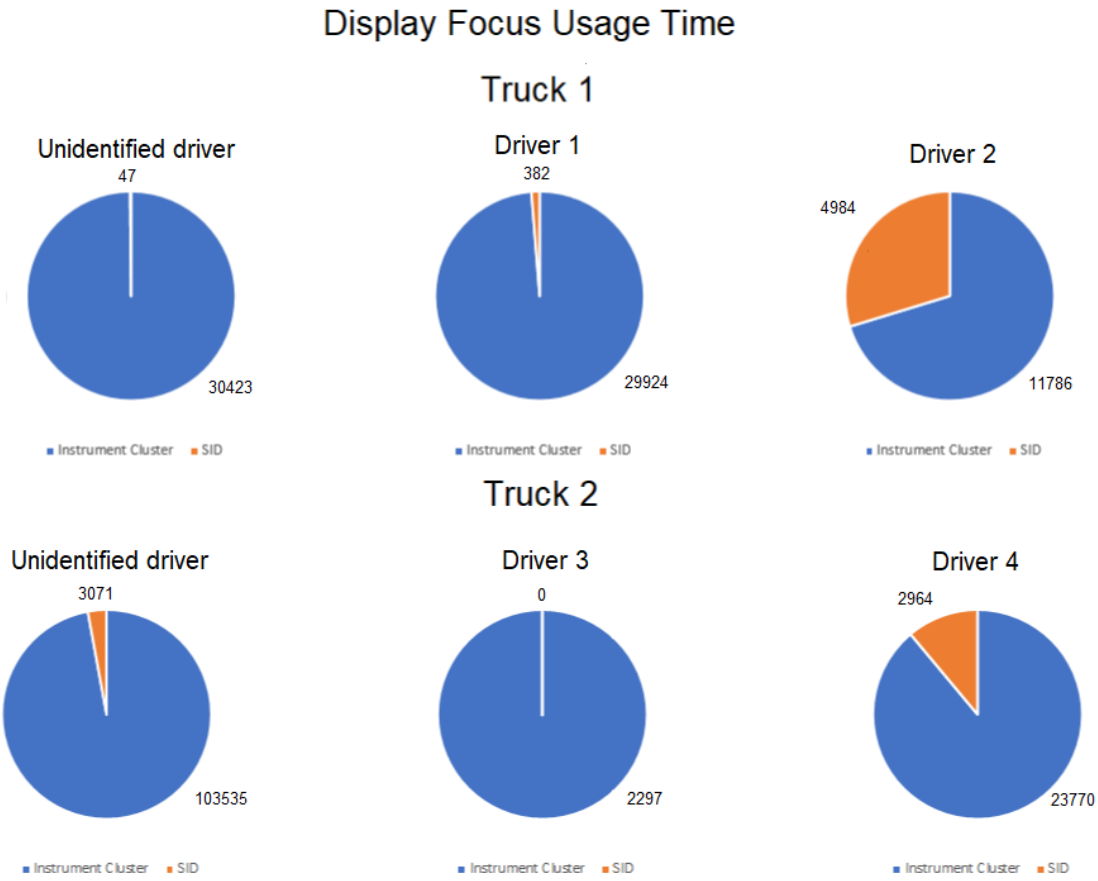

Figure 5.4: Display Focus usage time from two trucks in the test fleet. Times are presented in seconds.

Data on usage time per display focus from the two trucks is presented in Figure 5.4. In Truck 1, Driver 2 has a display focus on the SID almost equivalent to 30% of the total time. Both Unidentified Driver memory and Driver 1 have more total drive time recorded, but have significantly lower time with the display focus on SID compared to the Driver 2. From the raw data in Table A.3 (see Appendix A.3)

it can also be seen that Driver 2 has many more manual Focus Shift activations than the other driver memory sets in that truck. From Truck 2 it is seen that both Unidentified Driver memory and Driver 4 have used the Focus Shift function a considerable amount of time, while Driver 3 never used the function during a quite limited time.

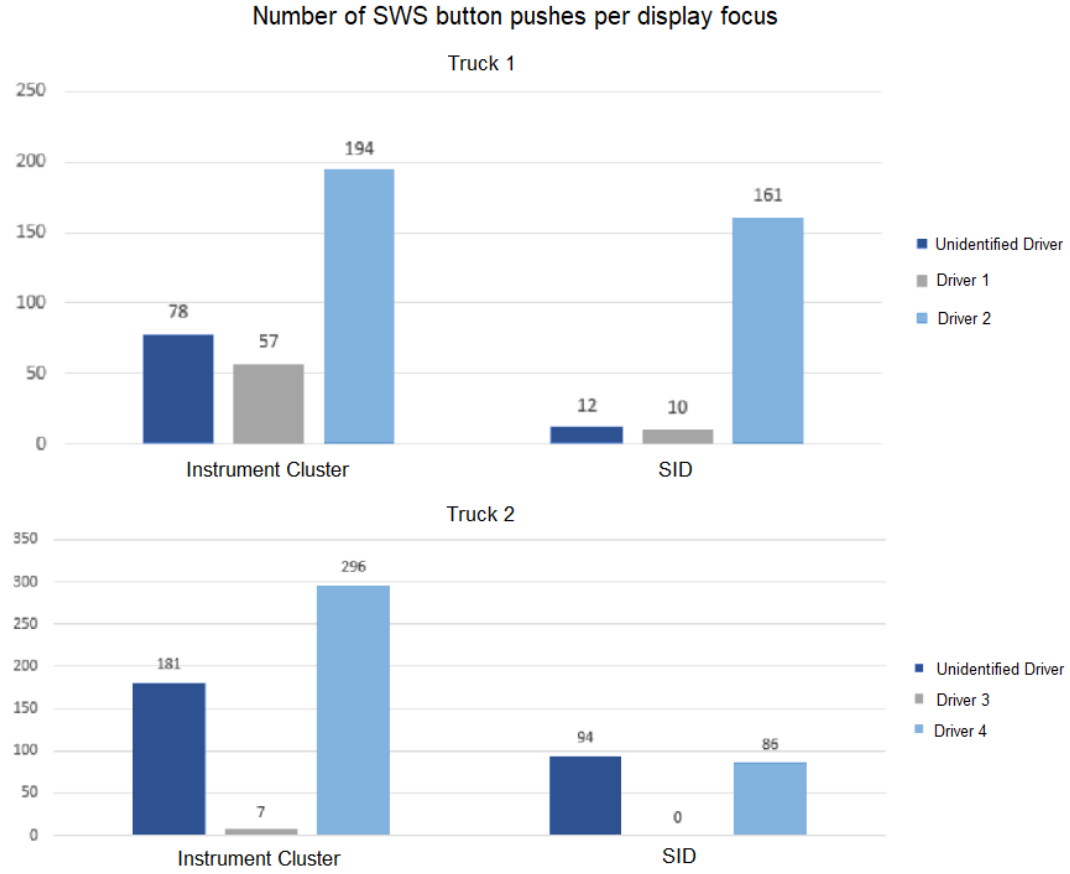

Figure 5.5: Button pushes per display focus from two trucks in the test fleet.

In Figure 5.5 data on button pushes per display focus is visualized for both the trucks. In Truck 1 the Unidentified Driver memory and Driver 1 have many more button pushes logged with display focus on Instrument Cluster than SID, while Driver 2 has a more even distribution. This result aligns well with the logged total times of display focus per user for that same truck. Also, from Truck 2, where the Instrument Cluster had by far the most display focus for all logged data sets in the truck, the usage of the buttons matches that result. However, the proportion between total time per display focus and number of button pushes does not match in either truck. The likely explanation is that display focus default is assigned to the Instrument Cluster and a lot of time will pass without any user interaction. On

the other hand, button pushes towards SID have to be initiated by the conscious decision of a driver to switch focus. As a result, more button pushes will likely occur towards SID within a shorter time frame. Combining data on usage time and the number of button pushes helps to identify how actively the driver has used the display focus on SID. From Truck 2 the Unidentified Driver memory and Driver 4 have both very even numbers on total SID display focus time and button pushes towards SID, indicating that when switching the focus to SID, the activity level from the different users has been similar.

Another interesting finding from the Focus Shift data is that the majority of manual Focus Shift activations have occurred while driving, see from raw data in Tables A.3 and A.4 (see Appendix A.3). This is the expected behavior since the touch functionality in the SID should be easier to use during a standstill.

From this small sample of Focus Shift data great variations between drivers are recorded. Especially Driver 2 in Truck 1 sticks out with a huge amount of Focus Shift usage recorded, both in total time with display focus on SID compared to Instrument Cluster, the number of button pushes towards SID and amount of manual activations. Also, Driver 4 in Truck 2 has a rather high Focus Shift usage recorded, while the other logged data sets have a small amount or no usage recorded. It is likely that similar variations will be found from customer data. Some users appreciate the Focus Shift and utilize it a lot, while for some it is rarely or never used. To find this distribution between frequent and rare users is important to understand the necessity of the function.

5.2.3 Gauge Logging

From the gauge list the driver can choose between various types of data to present. When studying the gauges the main research question is:

How important is each gauge and how to design the HMI considering the gauge usage and relevance?

In Figures 5.6 and 5.7 the occurrence counter and total time per gauge are presented from Truck 1. It is clear from the visualization that the speed gauge has been used the most. This is not surprising as the speed is presented in a clear way in the gauge and the value may vary a lot over time. One interesting gauge to study is the Engine Oil Temperature gauge. It is the gauge with second most occurrences, but with a comparatively small usage time. This probably indicates that the Engine Oil Temperature is data that the drivers like to keep track of, but they only need to check it quickly. A majority of the gauges have no recorded usage in the truck. Some data will not need regular monitoring from the drivers, but still need to be easy accessible whenever the data is required. In other words,

less usage of some gauges should be expected, but still considered sufficient. The relevance of all gauges needs to be evaluated. Then, a larger sample of customer gauge data can help in deciding what gauges to include and how to order the gauges in the HMI, based on the usage and relevance.

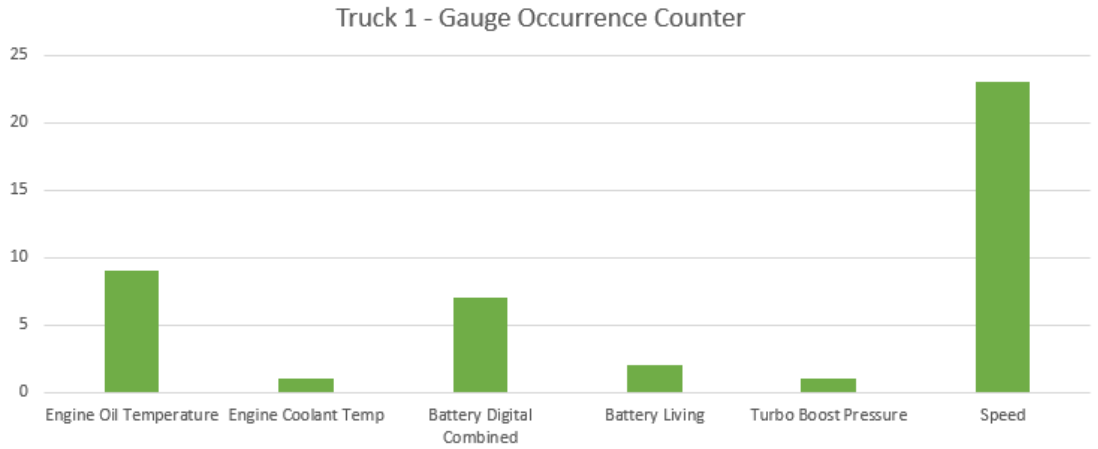

Figure 5.6: Logged number of occurrences for each gauge from Truck 1 in the test fleet. Gauges with no logged usage are not added to the graph.

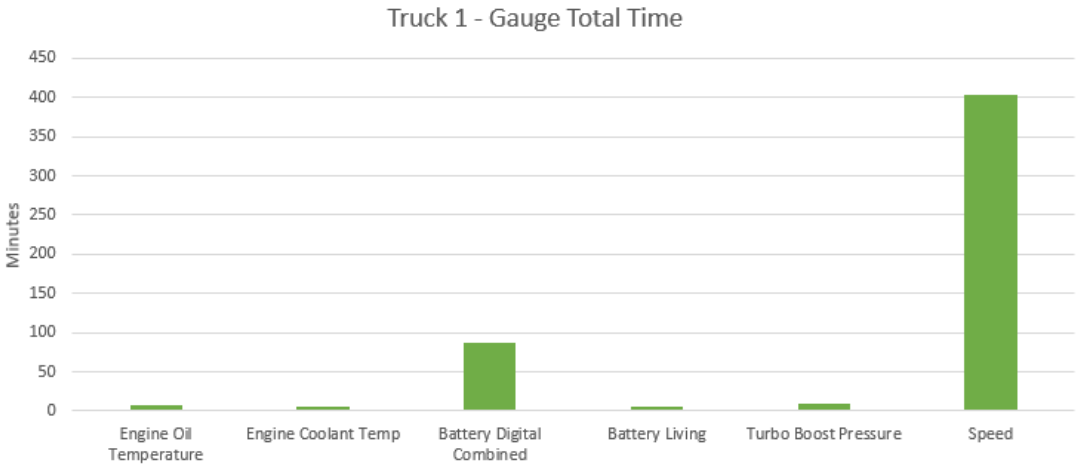

Figure 5.7: Logged total time spent with each gauge shown from Truck 1 in the test fleet. Gauges with no logged usage are not added to the graph.

5.2.4 Pre-Trip Inspection Logging

One of the standard questions for logging of all applications inside the Instrument Cluster HMI is:

How much is the application used by the drivers and how is it used?

From Truck 1, the DOID for the Pre-Trip Inspection menu application logging was unchanged over the time period, see Table A.6. This means that the application was not entered at all for the week. There could be a few reasons behind this. The first option is that the drivers have not found the feature, which seems very unlikely for internal test drivers. The second option is that the automatic Pre-Trip Inspection generation rendered no warnings, and therefore the drivers saw no reasons to check the reports. The last option is that the drivers are not satisfied with the feature and see no point in checking the reports. With a large sample of customer data and mixed methods research, the correct interpretation can be made.

5.3 The Proposed Way of Working With Logged User Interaction Data

Based on the literature review and interviews performed, when logged user data has become available, a mixed methods approach is recommended. The data collected first needs to be analyzed with the help of statistical analysis tools, like for instance Power BI. By identifying different patterns in the data, it shall be possible to identify different groups of users. With the flexibility provided by mixed methods design, questions can be directed to users from each group to understand the patterns they represent. This way the weaknesses of both quantitative and qualitative methods can be minimized. Interviews with the drivers will clarify the reasons for specifically identified driver behavior.

To do this complete analysis, collaborations between several fields are needed. Software developers are needed to implement the logging setup. To dig into the collected data, data analysts will be required. Finally, HMI engineers will be needed to request data and design and perform interviews based on the outcome of the logged data.

To understand the learning process of the HMI, data shall be collected over a longer time. By recognizing when the user behavior stops to vary over time, the length in time of the learning process can possibly be decided. Identifying the learning and usage processes can improve the analysis of different conclusions that can be made during the different steps.

Another reason to log data over time is to determine the outcome of newly introduced updates. Observing development in the collected data, conclusions on whether the updates sparked the intended user behavior can be made.

6
Discussion

6.1 Pros of More Dynamic Data Logging

Logging data using DOIDs means that the way the data is stored and which data to consider has to be predefined. In development of a logging setup, flexibility is of the highest importance. With a DOID the developer has to figure out beforehand which data that is interesting, but typically during development you will not have all the answers right away.

For example when implementing support for the DOID used for logging of Focus Shift, one of the code reviewers raised the question if it would be interesting to try to identify when the driver pushes a button and finds out that the focus is not on the intended display. The idea was that this could be identified by logging a button push which very shortly after would be followed by a Focus Shift activation. As the DOID had already been defined and this case had not been thought of it was already too late, unless creating a new DOID would be considered.

The risk is that there will be many cases where interesting scenarios to log are thought of first at a later stage. In those cases, having to define and implement a new or updated DOID is too much of a struggle. For a logging setup to be really powerful there is a need to easily define new scenarios to study or identify.

The alternative to storing data in DOIDs could be to utilize a high-frequency logging setup. By sending the logged attributes as signals to a central logging system data could be connected to time stamps. With the support of time stamps the user interactions could be connected to a lot of other factors like the status of the vehicle, driving conditions, etc.

Looking at high-frequency signals would also give the possibility to study the HMI on a more detailed level. By modelling the HMI according to the Automata theory described in section 2.2, events and states could be mapped to signal values in logging signals. Then the data could be postprocessed and the navigation inside the HMI could be followed in more detail. By defining a wanted user behavior it would be possible to define desired sequences of events inside the applications. Based on the logged events it could be determined if the user has followed the wanted interaction patterns or not. The idea is that signal definitions could be

more dynamic and new signals easier added than DOIDs. In this way, the logging implementation is more flexible and could faster be updated based on changes in the HMI implementation.

6.2 Logging Data Connected to Driver

One of the problems described by Orlovska et al. when logging vehicle data in cars is to make sure that the logged data is connected to the one and the same driver of the car [3]. Cars are often shared between multiple drivers and there is no good way to determine the current driver without risking the integrity of the drivers. Drivers can possibly be identified by monitoring certain vehicle settings and identifying different driver profiles, but this approach introduces lots of uncertainties.

It was concluded that when logging data for trucks there is a way to get around this issue as the current driver can be identified through driver card information. This ensures that logged data is connected to only one driver. Since behavior between different users of the same truck can differ, this is critical to separate patterns between different users.

However, with the current implementation of logging data defined by DOIDs, it was noted that also having to keep track of the current driver ID added a lot of complexity to the implementation. Although it has been proven that it is indeed possible to use the driver card data to identify the truck driver and log data per user, for the more complex functionality studied it was decided to only study user behavior per truck to reduce complexity and speed up implementation.

The hypothesis is that when utilizing high speed logging instead of predefined DOIDs it is more convenient to include the driver data and extract data sets per user.

Additionally, if storing data per user in DOIDs it is also worth considering the increase in data size that such a strategy requires. Logging data for 10 identified drivers and one unidentified driver memory means that 11 times more data is needed than if only logging data on a truck level. For instance, the DOID for Instrument Views logging is 1320 bytes, but would only have been 120 bytes if individual user data was not considered. Although handling of data gets cheaper and cheaper, there is still a cost associated with sending data from the truck to back office. To save data cost it is therefore necessary to question the value added by having access to data on user level compared to on truck level. For some commonly used features, it is interesting to see differences between different users, while for some features it might be enough to see the overall amount of usage.

The oldest used data set per user is designed to be reset if a new user is added and all data sets are already in use. With the current implementation of the DOIDs, there is no identification of resets of data sets. If the DOIDs were

redefined, one additional counter would have been added for each user to keep track of the number of resets done. With the current implementation it might be risky to check the usage over a certain time period since we cannot tell if any resets have occurred, unless it is obvious from how the data values have developed.

Although the impact that regulations like GDPR impose have not really been considered, it is clear that connecting data directly to the Driver Card Number would violate these types of regulations. By instead in the software translating the Driver Card Number to a signal value 1–10 which indicates the driver of the truck, it should not be possible to identify the driver unless the one studying the data also has access to for instance the VIN (Vehicle Identification Number). Once again, it is important to note that it is not the user behavior of one individual user that is of interest. What instead is important is to study patterns found among the users. In this sense, the help provided by driver card information to distinguish different drivers of a truck is crucial.

6.3 Collecting Data for Logging at the Right Level

When logging data, it is important to find the right level in the system to extract the data from. As described from the verification process in Subsection 4.5.4, one bug was found for the identification of manual Focus Shift activations. The main reason behind this bug was that conclusions from the available data was done at the wrong level. Looking at the HMI Architecture presented in Subsection 3.2.5, it is clear that the knowledge of how an activation occurred is available in the Core component. In the first implementation of Focus Shift logging the identification of manual Focus Shift activations was attempted at a later stage in the chain. Although, based on the system model, only manual activations would be possible in the higher Vehicle Modes when the logging is executing, the lack of control over the order signals are received over APX in the GFX platform lead to a bug.

Therefore, to have full control of the logging, it is important to collect the data at the right level. Data on status updates and events needs to be gathered from the primary source. Data like user requests need to be collected from the source of the request. For instance, in the case of the Pre-Trip Inspection application logging, the amount of user requests from the Instrument Cluster HMI to update each setting is counted. Then the request signals are sent directly from the HMI to the logging implementation in the GFX platform. If the request data collected instead had been the output signals from the Control component, there would have been no way to determine from where the request had occurred, only that a request has taken place. Meanwhile, to keep track of the current status of the same settings, data is collected from the Core component in the Instrument Cluster VP that is responsible for updating the status.

6.4 What Data Shall Be Logged in HMI?

When investigating what data to log, a number of requests on different data were made. As the interviews were conducted together with HMI engineers, most of the requests dealt with data directly connected to the HMI, but there have also been requests for data not connected to the HMI. As the Instrument Cluster gathers information for most functions inside the truck to present current status, warnings, etc it can be easy to make the conclusion that the Instrument Cluster is a good place to log all types of data. However, the purpose of the Instrument Cluster is to be a HMI component and the logging performed in the Instrument Cluster should be focused on HMI usage and functionality with core logic placed inside the Instrument Cluster.

One simple example could be the tachometer, i.e. the revolutions per minute (RPM) of the engine, which might be interesting to log to understand the performance and the usage of the truck. This data is presented in the Instrument Cluster HMI, but it requires no interaction from the user. The core component responsible for measuring the RPM is located in another ECU and the best place to collect data for logging would be directly from this ECU. To log data like this inside the Instrument Cluster would be to log from a secondary source. Then it can not be ensured that the data have not been filtered or altered in any way. Ambition should always be to use the raw data from the primary source for logging.

The same goes for things like warnings. It is possible to log if a warning is displayed in the Instrument Cluster, but it would be more accurate to track a CAN signal responsible for triggering the warning in the first place.

Therefore, whenever some data is considered to be logged, developers need to analyze how and where the data is best suited to be logged. If data relates to user interaction with HMI or core logic in the Instrument Cluster, then logging the data inside the Instrument Cluster should be considered. Otherwise, the data is probably better suited to be collected directly from its primary source in another ECU.

6.5 Logging Shall Be Considered in Early Phase

The logging implemented, which is part of this project, has been merged to software that is already in production. The logged data can then be used to understand how the product is used by the end customer. Still, as made clear from the interviews with HMI engineers, there is a wish to include logging of user interaction already in earlier stages of a project. Although data from test drivers might not be feasible for identifying user patterns, as they are instructed to test most features in the truck, it can be applicable for parts of the test fleet. Then it can be analysed in

an earlier stage of a project whether a solution is used as intended or not, before reaching the end customer. Furthermore, it is important to remember that it is needed to get proper verification of the logging systems before going to production, such that the logged data can be trusted.

Additionally, a good software architecture requires consideration of data logging in the early software design decisions. When a new feature is implemented, the availability of any data that is needed to understand the usage shall be considered already in the concept phase. This approach can enable a simpler implementation where logging is part of the initial design. It is typically much harder and riskier to refactor existing code in later phases of a project.

6.6 How Can Logged User Interaction Data Benefit Development?

Although no real user interaction data from customers has been collected yet for the implemented DOIDs, it is possible to speculate on what types of patterns could be found once data is available and how that could influence future development.

Based on the logged data for Instrument Views, it will hopefully be possible to identify patterns on what data drivers are most interested in presenting during driving and in what form they want it presented. For instance, it could be possible to identify if most drivers prefer to have the speed presented in digital format or the way it is presented in the Analog View. Based on the information available in the widgets in the different views, it is likely that some insights on what data the drivers prefer to have presented there can also be gained. With data on Instrument View usage, the ambition is that the Instrument Views can be refactored to include the most important information. It will also be possible to see how the usage develops with future updates to determine if changes were met positively by the users or not.

With data on usage of the gauges it might be possible to see which data presented in the gauges that is prioritized by the drivers. Since both amount of time and number of occurrences are logged, it might be possible to make conclusions on how regularly the driver wants to follow up a specific value. It could be that some gauges will have a short total time, but still a lot of occurrences. Then it is likely that the driver is interested in the value, but only checks it quickly from time to time. On the other hand, some gauges might have a much larger total time, as they are probably followed up more continously. It might also be that some gauges are found to be checked very seldom. Improvements that could be made based on such data are, for instance, to order the gauges in the HMI based on the typical usage, such that the most frequently used are found first when scrolling in the list.

Another possible outcome could be to move the gauges that are very frequently used out of the gauge list and instead add the data directly to an Instrument View, to have it more available. Gauges that are very rarely used could be considered for removal.

For example, Focus Shift is a feature that adds a lot of complexity in especially the Secondary Information Display as it needs to support both touch and control button inputs. Still, the usage of Focus Shift is unknown, both the amount of usage and when it is used. Therefore, there is a question if resources shall be spent to keep supporting the feature. With logged user data, real facts can be used and the right decision can more likely be taken.

Another example is the Pre-Trip Inspection, a newly introduced and highly branded feature. As a new feature, there is still not known how much it is appreciated and used by the customers. The logged data can help understand if the drivers have identified the new function and if so how often and when they are checking the report. It can also tell what settings a majority of the drivers prefer, which can be a valuable input for future function improvement.

6.7 Future Work

As a first step in the future, the data logged in the introduced DOIDs needs to be analyzed once real customer data is available. As the lead time from software delivery to introduction of the software in the production is long, it will still take some time until data is available. With data available, good tools to visualize and analyze the data are needed. One such tool could be Power BI. Then it will hopefully be possible to identify a number of user patterns. As it was mentioned before, quantitative data do not explain why certain patterns are seen. To verify data-driven insights, qualitative data should later be collected from drivers representing the different groups of patterns identified. That can hopefully bring more understanding to the patterns that were found and help make future design decisions. After the HMI solutions have been refactored, new data can be collected to iteratively develop the HMI.

There is still a lot of available data in the Instrument Cluster that is not yet logged. The logical future direction would be the improvement of the logging system. The more data that is collected, the more connections can be found between different data. For instance, right now, only one of many menu applications is logged. With data logged for more applications, it will be possible to compare the usage between the different applications. Data logged within the scope of this project was based on input from HMI Engineers. However with this first data available there is now a need to build up a prioritized backlog of what to log next.

As described earlier, there is a need for a more dynamic high frequency way

of logging data. In this project, data logging has only been considered for the Instrument Cluster, but for the logging to be really powerful a system approach is required. By combining data from the Instrument Cluster, other HMI components, core logic in other ECUs and driving conditions the full system perspective will be given and even more understanding of driver interactions will be possible. Thus, there is a need to push for such a setup in the future, as the current DOID logging is limited.

Finally, HMI development at Volvo Trucks has historically mostly been based on inputs from qualitative methods. Now it is the time for the organization to switch focus and adopt a data-driven approach. The knowledge gained from this project has to be spread among the engineers to encourage more logging implementation, create a good common architecture for logging, and adapt solutions based on what is found in the quantitative data.

7
Conclusion

By implementing a user interaction logging setup in an Instrument Cluster, it has been proven that it is possible to extract data from the HMI to base user behavior evaluation on. Since navigation inside the HMI can be followed, user behavior evaluation can be performed on a higher level of detail than most existing studies, where data has been focused on CAN, LIN, Ethernet and video recordings. Logging has been implemented for Instrument Views, Focus Shift, Gauges and the Pre-Trip Inspection menu application.

Data has been stored in the form of DOIDs, parameters with a set structure that can be read using diagnostic tools. The DOIDs was used to log counters and current settings. To fully follow navigations inside the HMI, a more dynamic way of logging data is required. With a high frequency logging setup, each status or event of interest could be monitored continuously. This would give more freedom for postprocessing of the data and enable a deeper understanding of the usage and navigations in the HMI. Furthermore, with time stamps for logged events, it would be possible to connect the HMI data to external data like vehicle status, driving conditions, etc.

As support for the logging setup design, Discrete Event System theory was proven useful. Modelling parts of the HMI as automata helped in identifying available states and possible events. This, in turn, made it visible what type of data and signalling that was needed for the implementation. During design and implementation, it is important to identify the best level to collect interesting data from. Data collection at the wrong level, that is, not from the original source, increases the risk of design errors.

One advantage of logging user data in the truck industry compared to cars is the possibility of easily separating data from different drivers. In the car industry, it is often hard to identify the current driver of the vehicle, while in the truck industry the driver can be identified via the driver card. This means that logged data can be stored based on the current driver ID. However, due to GDPR restrictions, data shall not be logged directly connected to the Driver Card Number, instead data was stored to generic data sets for drivers 1-10. It is also important to mention that logging data per driver increases the required data size of the DOIDs. Therefore,

to save data costs, the need for individual user data shall be evaluated per function before implementation of logging.

Coming to the data analysis part, logging of user data can help identifying user patterns, but it does not explain why those patterns are observed. To get a complete understanding of the HMI usage a mixed methods approach is recommended. Qualitative research methods like interviews can be performed after the data collection to understand why the users behave a certain way. Qualitative methods standalone does not provide a large enough sample size to make decisions based on clear facts. Data shall also be logged over time to determine how the usage develops, based on factors like functional updates or enhanced knowledge from the driver. Since HMI development at Volvo Trucks historically has been dominated by qualitative methods, there is now a need for the organization to adopt a data-driven approach in order to stay competitive.

Bibliography

[1] Mike Cohn. *Are 64% of Features Really Rarely or Never Used?* 2015. URL: https://www.mountaingoatsoftware.com/blog/are-64-of-features-really-rarely-or-never-used#comments (visited on 05/19/2021).

[2] Julia Orlovska, Fjollë Novakazi, Casper Wickman, and Rikard Soderberg. "Mixed-Method Design for User Behavior Evaluation of Automated Driver Assistance Systems: An Automotive Industry Case". In: *Proceedings of the Design Society: International Conference on Engineering Design* 1 (July 2019), pp. 1803–1812. DOI: 10.1017/dsi.2019.186.

[3] Julia Orlovska, Casper Wickman, and Rikard Söderberg. "Big Data Usage Can Be a Solution for User Behavior Evaluation: An Automotive Industry Example." In: *Procedia CIRP* 72 (Jan. 2018), pp. 117–122. DOI: 10.1016/j.procir.2018.03.102.

[4] A. Strauss and J. Corbin. *Basics of qualitative research.* Vol. 15. Newbury Park, CA: SAGE, 1990.

[5] Louis Cohen, Lawrence Manion, and Keith Morrison. *Research methods in education (7th ed.)* Vol. 15. London: Routledge, 2011.

[6] Md Rahman. "The Advantages and Disadvantages of Using Qualitative and Quantitative Approaches and Methods in Language "Testing and Assessment" Research: A Literature Review". In: *Journal of Education and Learning* 6 (Nov. 2016), p. 102. DOI: 10.5539/jel.v6n1p102.

[7] Joseph A. Maxwell. *Qualitative Research design: An interactive approach.* London: Sage, 2012.

[8] N. K. Denzin and Y. S. Lincoln. *Handbook of qualitative research.* London: Sage Publications, 1994.

[9] U. Flick. *Introducing research methodology: A beginner's guide to doing a research project.* London: Sage Publications Ltd., 2011.

[10] S. M. Rasinger. *Quantitative research methods in linguistics: An introduction.* A C Black, 2013.

[11] A. Bryman. *Social research methods*. New York: Oxford University Press, 2012.

[12] L. Carr. "The strengths and weaknesses of quantitative and qualitative research: what method for nursing?" In: *Journal of advanced nursing* 20 4 (1994), pp. 716–21.

[13] P. Connolly. *Quantitative data analysis in education: A critical introduction using SPSS*. London New York, NY: Routledge, 2007.

[14] R. Johnson and Anthony Onwuegbuzie. "Mixed Methods Research: A Research Paradigm Whose Time Has Come". In: *Educational researcher* 33 (Oct. 2004), p. 14. DOI: 10.3102/0013189X033007014.

[15] Julia Orlovska, Fjollë Novakazi, Lars-Ola Bligård, Marianne Karlsson, Casper Wickman, and Rikard Söderberg. "Effects of the driving context on the usage of Automated Driver Assistance Systems (ADAS) -Naturalistic Driving Study for ADAS evaluation". In: *Transportation Research Interdisciplinary Perspectives* 4 (Feb. 2020), p. 100093. DOI: 10.1016/j.trip.2020.100093.

[16] DIN prEN ISO 9241-11. *Ergonomics of human-system interaction - Part 11: Usability: Definitions and concepts*. Berlin: Beuth Verlag, 2016.

[17] Saila Ovaska. "Usability as a goal for the design of computer systems". In: *Scandinavian Journal of Information Systems* 3 (Sept. 1991), pp. 47–62.

[18] Jan-Timo Stahlmann, Lucia Becerril, and Jesko Beck. "Usability of Processes in Engineering Design". In: *Proceedings of the 21st International Conference on Engineering Design*. Vol. 2. Aug. 2017.

[19] Melanie Peham, Gert Breitfuss, and Rafael Michalczuk. "The "EcoGator" App: Gamification for Enhanced Energy Efficiency in Europe". In: *Proceedings of the Second International Conference on Technological Ecosystems for Enhancing Multiculturality*. TEEM '14. Salamanca, Spain: Association for Computing Machinery, 2014, pp. 179–183. ISBN: 9781450328968. DOI: 10.1145/2669711.2669897. URL: https://doi.org/10.1145/2669711.2669897.

[20] Vicki Lewis, Thomas Dingus, Sheila Klauer, and Jeremy Sudweeks. "An overview of the 100-car naturalistic study and findings". In: *Proc. Int. Tech. Conf. Enhanced Safety Vehicles* 19 (Jan. 2005).

[21] Lex Fridman, Daniel Brown, Michael Glazer, William Angell, Spencer Dodd, Benedikt Jenik, Jack Terwilliger, Aleksandr Patsekin, Julia Kindelsberger, Li Ding, Sean Seaman, Alea Mehler, Andrew Sipperley, Anthony Pettinato, Bobbie Seppelt, Linda Angell, Bruce Mehler, and Bryan Reimer. "MIT Advanced Vehicle Technology Study: Large-Scale Naturalistic Driving Study of

Driver Behavior and Interaction With Automation". In: *IEEE Access* PP (July 2019), pp. 1–1. DOI: 10.1109/ACCESS.2019.2926040.

[22] Julia Orlovska, Casper Wickman, and Rikard Söderberg. "BIG DATA ANALYSIS AS A NEW APPROACH FOR USABILITY ATTRIBUTES EVALUATION OF USER INTERFACES: AN AUTOMOTIVE INDUSTRY CONTEXT". In: Jan. 2018, pp. 1651–1662. DOI: 10.21278/idc.2018.0243.

[23] Masakazu Adachi, Toshimitsu Ushio, and Yoshitaka Ukawa. "Bisimulation based design of user-interface for discrete event systems". In: *IFAC Proceedings Volumes* 37.18 (2004). 7th International Workshop on Discrete Event Systems (WODES'04), Reims, France, September 22-24, 2004, pp. 159–164. ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)30739-5. URL: https://www.sciencedirect.com/science/article/pii/S1474667017307395.

[24] Bengt Lennartson, Martin Fabian, and Knut Åkesson. *Introduction to Discrete Event Systems: Lecture Notes*. 2009.

[25] *GENERAL INFORMATION ABOUT AUTOSAR*. 2021. URL: https://www.autosar.org/about/ (visited on 05/19/2021).

[26] Conny Gustafsson. *APX*. URL: https://cogu.github.io/apx/ (visited on 05/19/2021).

[27] Trygve Reenskaug. "The original MVC reports". In: 1979.

[28] Derek Greer. *Interactive Application Architecture Patterns*. 2007. URL: https://lostechies.com/derekgreer/2007/08/25/interactive-application-architecture/ (visited on 05/19/2021).

[29] *QML Applications*. URL: https://doc.qt.io/qt-5/qmlapplications.html#what-is-qml (visited on 05/19/2021).

[30] *<ctime> (time.h)*. URL: https://www.cplusplus.com/reference/ctime/ (visited on 05/19/2021).

[31] Kevin Forsberg and H. Mooz. "The Relationship of System Engineering to the Project Cycle". In: *Engineering Management Journal* 4 (Apr. 2015), pp. 36–43. DOI: 10.1080/10429247.1992.11414684.

[32] *Testing ECUs and Networks with CANoe*. URL: https://www.vector.com/se/en-se/products/products-a-z/software/canoe/ (visited on 05/19/2021).

A
Appendix

A.1 Full Questionnaire for HMI Interviews

Below is the full questionnaire that was used for the interviews with HMI Engineers at Volvo Group Trucks Technology. All of the main questions were used and then the predefined follow-up questions were used to clarify some answers or change the perspective from how the main question first was approached by the interviewee.

1. What is the current role of vehicle field data in the automotive UX development?
   (a) When and how are user behavior evaluations done today?
   (b) How does the results you get from the user behavior evaluations done today help you in your work?
2. What are the needs, challenges and concerns in the context of data-driven UX development?
   (a) What type of information are you missing today when designing or specifying a HMI?
   (b) How do you want user behavior evaluations presented?
3. What is specific to the automotive UX development and what can be adopted from the non-automotive areas of UX development?
   (a) Do you see any pros and cons in basing user behavior evaluations on communication between ECUs (CAN, Ethernet) or based on internal data from a HMI?
4. How can the automotive UX development benefit from data-driven approach?
   (a) How can user behavior evaluations based on logged data from real trucks help you in your work?

(b) What are your thoughts on combining qualitative and quantitative approaches for user behavior evaluation?

(c) In what step of the development process do you want user behavior logging included? When in the process do you think the gain is the highest?

(d) How can user behavior evaluation based on real user data be integrated in the normal way of working?

(e) What is your vision for data-driven development?

5. What type of user behavior information is of interest?

(a) Are there any specific functions that are more interesting?

(b) Are there any specific use cases you are extra interested in knowing how the users interact?

A.2 Definition of DOIDs

A.2.1 DOID for Instrument View Logging

Caption: Instrument Cluster, Instrument View Usage, Log
Data size: 1320 bytes
Description:
Logs data regarding what Instrument Views that has been displayed in the Instrument Cluster (IC) per user.

This log contains data for one default set when no user has been identified and 10 sets for individual users of the truck identified by Driver Card Information. Each user data set contains information for up to 15 different Instrument Views.

Each Instrument View log contains:

- One occurrence counter for the total number of times the specific Instrument View has been entered.
- One counter for the total amount of time spent in the Instrument View.

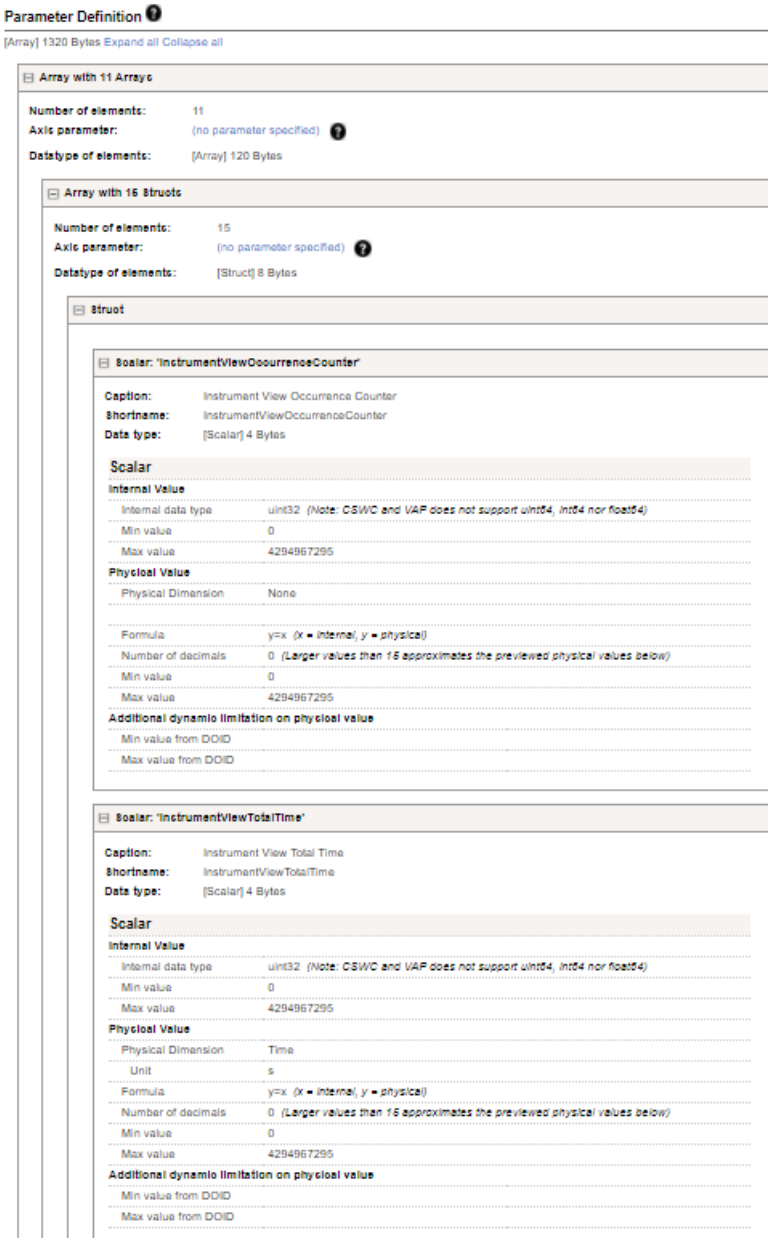

Figure A.1: Parameter definition for the DOID used for logging of Instrument Views. The parameter is an array of arrays with size 11×15 as there is support for $1 + 10$ driver memories and 15 Instrument Views. Each element is a struct of data logged per Instrument View per driver memory.

A.2.2 DOID for Focus Shift Logging

Caption: Display Focus, Focus Shift, Log
Data size: 220 bytes
Description:
Logs the usage of manual Focus Shift between Instrument Cluster (IC) and Secondary Information Display (SID) per user.

Each array index matches one user identified by Driver Card Information, with the first array index representing a default user when no driver has been identified.

When the vehicle is in accessory mode or any other lower modes, the focus is automatically set and therefore data will not be logged in these modes. In the pre-running mode or any other higher modes, the focus can be controlled by the driver and data is logged. Only buttons on the steering wheel switch that are impacted by the focus shift function are counted. Focus Shift activations are only counted when shifting the focus from IC to SID, not when changed from SID to IC.

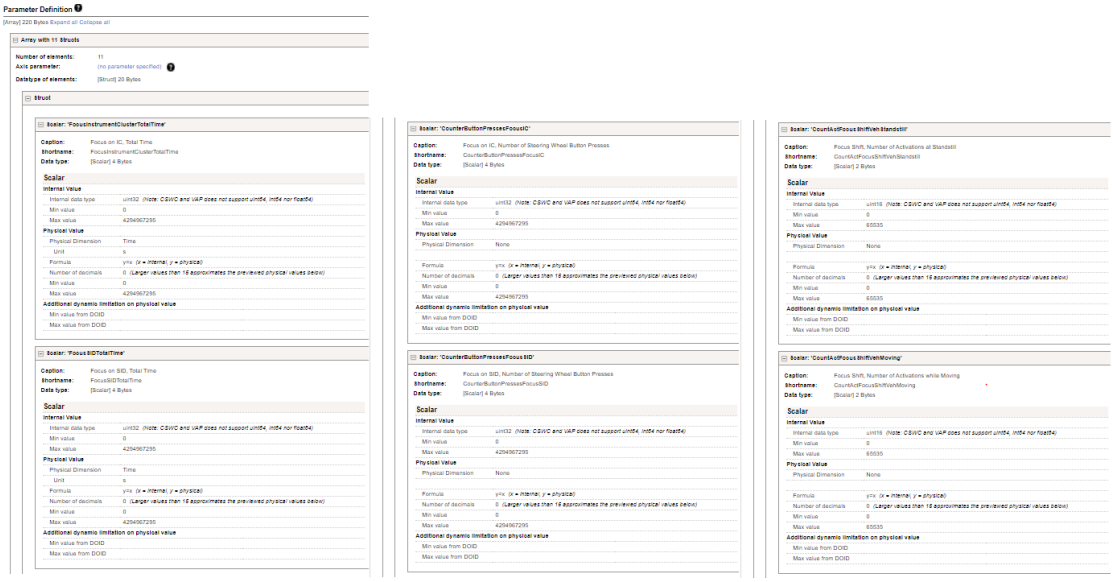

Figure A.2: Parameter definition for the DOID used for logging of Focus Shift and SWS buttons. The parameter is an array of size 11 as there is support for $1 + 10$ driver memories. Each element in the array is a struct of Focus Shift data logged per driver memory.

A.2.3 DOID for Pre-Trip Inspection Logging

Caption: Instrument Cluster, Pre-Trip Inspection Usage, Log
Data size: 19 bytes
Description:
Logs data regarding usage of the Pre-Trip Inspection application in the Instrument Cluster per truck.

The log is created to understand the usage of the Pre-Trip Inspection application in the Instrument Cluster. The Pre-Trip Inspection function provides the driver with a report of the status of the vehicle. Any problems found on the truck by Pre-Trip Inspection is presented to the driver. By collecting data on the usage of Pre-Trip Inspection when and how the report is used by the driver can be better understood.

The log contains the following data:

- Counter for the total amount of time spent inside the Pre-Trip Inspection application.
- Counter of amount of times the application is entered with vehicle at standstill.
- Counter of amount of times the application is entered while vehicle is moving.
- Counter of manual generations of Pre-Trip Inspection reports.
- Counter of times the filter report setting is changed by the user.
- Counter of times the report autogeneration setting is changed by the user.
- Counter of times the exterior lights check setting is changed by the user.
- The current filter report setting.
- The current report autogeneration setting.
- The current exterior lights check setting.

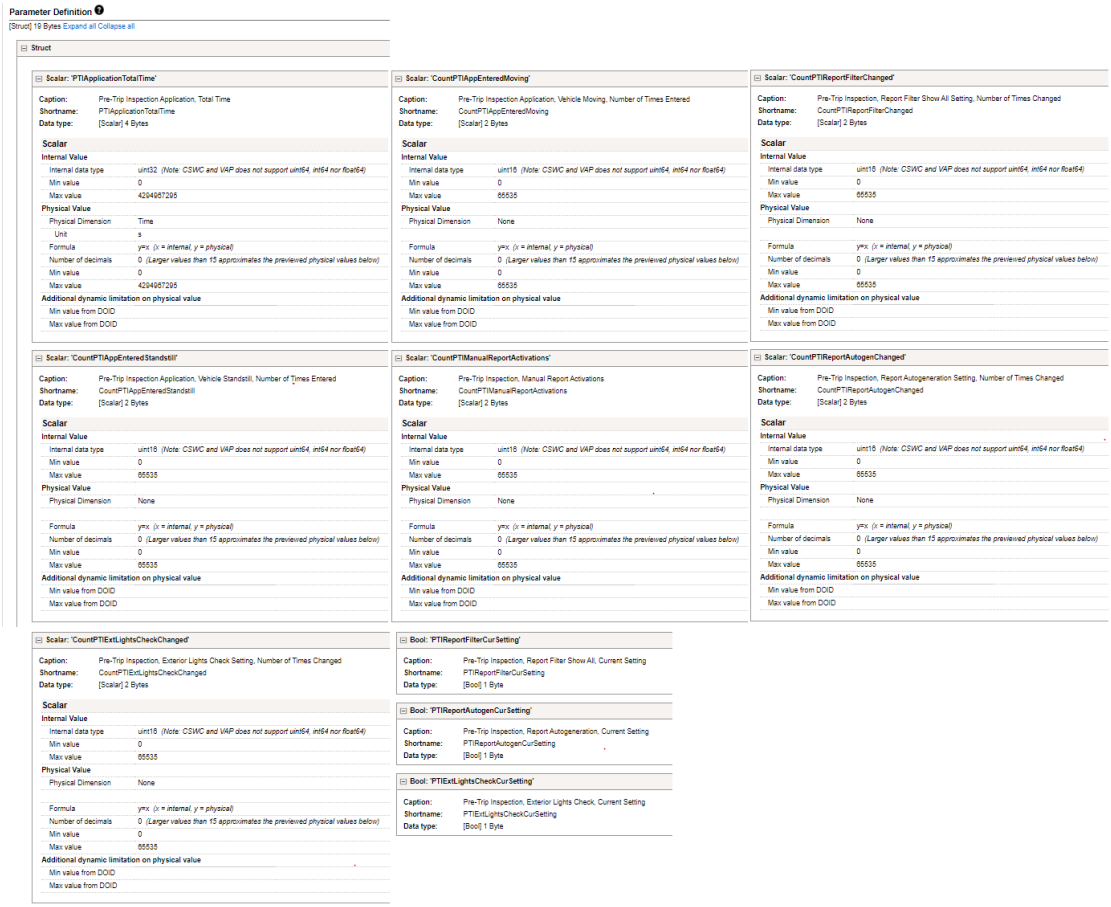

Figure A.3: Parameter definition for the DOID used for logging of the Pre-Trip Inspection application in the Instrument Cluster. The parameter is a struct storing data collected from the HMI. Data is logged on truck level and not per individual user.

A.2.4 DOID for Gauge Logging

Caption: Instrument Cluster, Gauge Usage, Log
Data size: 180 bytes
Description:
Logs data regarding what gauges that has been displayed in the Instrument Cluster (IC).

This log contains usage data for up to 30 different gauges. For each gauge the following data is logged:

- One occurrence counter for the total number of times the specific gauge has been entered.
- One counter for the total amount of time spent with the gauge shown.

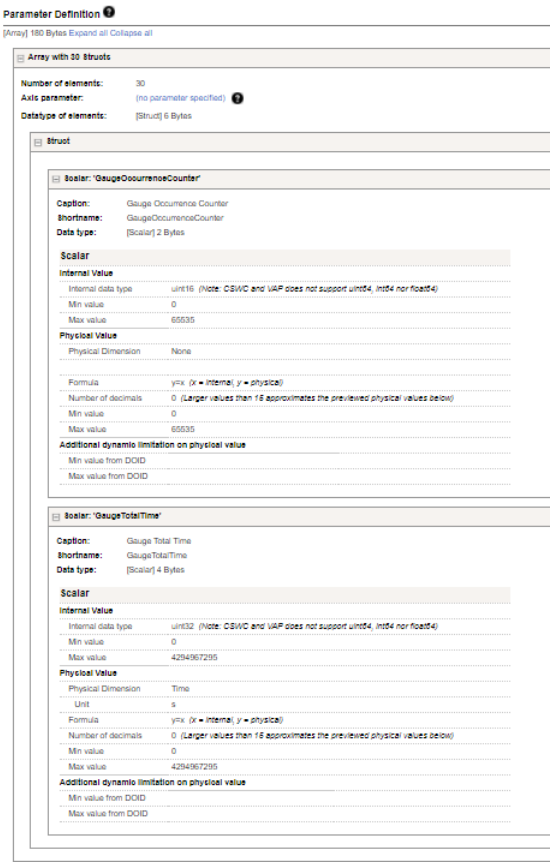

Figure A.4: Parameter definition for the DOID used for logging of Gauges. The parameter is an array of size 30 there each element is a struct of data logged per gauge. For each gauge number of occurrences and total time spent is logged.

A.3 Raw Data From Data Collection

Driver	Instrument View	Occurrence Counter	Total Time (s)
Unidentified Driver	Analog View	13	6386
	Navi View	13	19652
	Load View	2	294
	Focus View	7	5529
Driver 1	Analog View	11	13109
	Navi View	7	14971
	Load View	2	2292
	Focus View	1	557
Driver 2	Analog View	12	11536
	Navi View	11	1468
	Load View	6	890
	Focus View	7	2798

Table A.1: Raw data from logging of Instrument Views in Truck 1.

Driver	Instrument View	Occurrence Counter	Total Time (s)
Unidentified Driver	Analog View	40	55072
	Navi View	24	16827
	Load View	1	775
	Focus View	12	33882
Driver 3	Analog View	1	434
	Navi View	3	1650
	Load View	0	0
	Focus View	1	212
Driver 4	Analog View	21	23137
	Navi View	8	3050
	Load View	2	506
	Focus View	1	7

Table A.2: Raw data from logging of Instrument Views in Truck 2.

Driver	Unidentified Driver	Driver 1	Driver 2
FocusInstrumentClusterTotalTime (s)	103535	2297	23770
FocusSIDTotalTime (s)	3071	0	2964
CounterButtonPressesFocusIC	181	7	296
CounterButtonPressesFocusSID	94	0	86
CountActFocusShiftVehStandstill	0	0	2
CountActFocusShiftVehMoving	8	0	9

Table A.3: Raw data from logging of Focus Shift in Truck 1.

Driver	Unidentified Driver	Driver 3	Driver 4
FocusInstrumentClusterTotalTime (s)	30423	29924	11786
FocusSIDTotalTime (s)	47	382	4984
CounterButtonPressesFocusIC	57	78	194
CounterButtonPressesFocusSID	10	12	161
CountActFocusShiftVehStandstill	1	0	1
CountActFocusShiftVehMoving	0	5	20

Table A.4: Raw data from logging of Focus Shift in Truck 2.

Gauge	GaugeOccurrenceCounter	GaugeTotalTime (s)
Engine Oil Temperature	9	443
Air Pressure Monitor	0	0
Engine Coolant Temperature	1	291
Engine Oil Level	0	0
Engine Oil Pressure	0	0
Battery Digital Combined	7	5232
Battery Living	2	335
Battery Start	0	0
Gearbox Oil Level	0	0
Gearbox Oil Temperature	0	0
Battery Digital Volt	0	0
Turbo Boost Pressure	1	505
Body Builder One	0	0
Body Builder Two	0	0
Body Builder Three	0	0
Speed	23	24185
Empty	0	0

Table A.5: Raw data from logging of Gauges in Truck 1.

Data	Value	Unit
PTIApplicationTotalTime	0	s
CountPTIAppEnteredStandstill	0	–
CountPTIAppEnteredMoving	0	–
CountPTIManualReportActivations	0	–
CountPTIReportFilterChanged	0	–
CountPTIReportAutogenChanged	0	–
CountPTIExtLightsCheckChanged	0	–
PTIReportFilterCurSetting	false	boolean
PTIReportAutogenCurSetting	true	boolean
PTIExtLightsCheckCurSetting	true	boolean

Table A.6: Raw data from logging of Pre-Trip Inspection menu application in Truck 1. The DOID still has default value, meaning that no usage of the application has occurred.

A.4 Code Excerpts

This section will present excerpts of the code implemented for logging user interaction data in the DOIDs. The purpose is to let the reader understand the kind of logic that has been implemented, as a major part of the work performed in the project has been the actual implementation. Some of the code written is add-ons to already existing files in the production software and will not be presented here due to confidentiality. This means that for instance the implementation of the new logging signals sent from VP and HMI layer and how they are set can not be presented here. Also the implementation for how the signals are connected between the different APX servers is missing. Some parts in the files listed below, like for instance some general comments, has been left out as it is considered of less importance. As the gauge logging implementation is very similar to the Instrument View logging that code is also left out from here. The unit tests written in Google Test for the Focus Shift logging files are included to present how the lowest level of testing has been performed.

A.4.1 Shared Logging Functions and Data

A.4.1.1 LogListBase.h

```
#pragma once

#include "Platform_Types.h"
#include <stddef.h>

namespace LogListBase
{
   int readFromFile(const char* filename, char* buf, size_t bufSize);
   int writeToFile(const char* filename, char* buf, size_t bufSize);
   uint16 countToMaxUint16(uint16 value);
   uint32 countToMaxUint32(uint32 value);
   uint32 addTime(uint32 curTime, uint32 timeToAdd);
}
```

A.4.1.2 LoglistBase.cpp

```
#include "LogListBase.h"
#include "Platform_Types.h"
#include "GfxLog.h"
#include <string.h>
#include <stdbool.h>
#include <iostream>
#include <fstream>

namespace LogListBase
{
int readFromFile(const char* filename, char* buf, size_t bufSize)
{
   int retVal = -1;

   std::ifstream inFile(filename, std::ifstream::binary | std::ifstream::ate);

   if (inFile.is_open())
   {
      auto size = inFile.tellg();
      if (size == static_cast<long int>(bufSize))
      {
         inFile.seekg(0, std::ifstream::beg);
         inFile.read(buf, size);
         retVal = 0;
      }
      else
      {
         GFX_LOG_ERROR("File %s has not expected size of %u bytes",
               filename, (uint32_t)bufSize);
      }
   }
   else
   {
      GFX_LOG_ERROR("Unable to open file %s for reading", filename);
   }

   return retVal;
}

int writeToFile(const char* filename, char* buf, size_t bufSize)
{
   int retVal = -1;

   std::ofstream outFile(filename, std::ofstream::binary);

   if (outFile.is_open())
   {
      outFile.write(buf, bufSize);
      retVal = 0;
   }
   else
   {
      GFX_LOG_ERROR("Unable to open file %s for writing", filename);
   }

   return retVal;
}

uint16 countToMaxUint16(uint16 value) {
```

```
   if (value < UINT16_MAX) {
      value++;
   }

   return value;
}

uint32 countToMaxUint32(uint32 value) {
   if (value < UINT32_MAX) {
      value++;
   }

   return value;
}

uint32 addTime(uint32 curTime, uint32 timeToAdd)
{
   if (curTime <= UINT32_MAX - timeToAdd)
   {
      return curTime + timeToAdd;
   }

   return UINT32_MAX;
}
}
```

A.4.1.3 SharedLoggingData.h

```
#ifndef SHAREDLOGGINGDATA_H_
#define SHAREDLOGGINGDATA_H_

#include "LoggedMenuApplicationOpen_T.h"
#include "VehicleMovingStatus_T.h"

#ifdef __cplusplus
extern "C" {
#endif

void LoggedMenuApplicationOpen_CallbackFunc(uint8 value);
void PS_VehicleMoving_CallbackFunc(uint8 value);
LoggedMenuApplicationOpen_T GetLoggedMenuApplicationOpen();
VehicleMovingStatus_T GetVehicleMovingStatus();

#ifdef __cplusplus
}
#endif


#endif /* SHAREDLOGGINGDATA_H_ */
```

A.4.1.4 SharedLoggingData.cpp

```
#include "SharedLoggingData.h"

static LoggedMenuApplicationOpen_T mMenuApplicationOpen = LOGGED_MENU_APPLICATION_OPEN_NONE;
static VehicleMovingStatus_T mVehicleMoving = VEHICLE_MOVING_STATUS_NOT_AVAILABLE;


void LoggedMenuApplicationOpen_CallbackFunc(uint8 value) {

   mMenuApplicationOpen = value;
}

void PS_VehicleMoving_CallbackFunc(uint8 value) {

   mVehicleMoving = value;
}

LoggedMenuApplicationOpen_T GetLoggedMenuApplicationOpen() {
   return mMenuApplicationOpen;
}

VehicleMovingStatus_T GetVehicleMovingStatus() {
   return mVehicleMoving;
}
```

A.4.1.5 UTCTimeStamp.cpp

```
struct tm UtcTimestamp_setTmStructFromUtcDateTimeStruct(const UtcDateTime *utcDateTime_p)
{
   struct tm utcTmStruct;
   utcTmStruct.tm_sec = utcDateTime_p->second;
   utcTmStruct.tm_min = utcDateTime_p->minute;
   utcTmStruct.tm_hour = utcDateTime_p->hour;
   utcTmStruct.tm_mday = utcDateTime_p->day;
   utcTmStruct.tm_mon = utcDateTime_p->month - 1; // struct tm month has range 0-11, utc signal received range 1-12
   utcTmStruct.tm_year = utcDateTime_p->year+85; //time.h library counts years from 1900, Volvo from 1985
   utcTmStruct.tm_wday = 0; //ignored
   utcTmStruct.tm_yday = 0; //ignored
   utcTmStruct.tm_isdst = 0; //daylight saving off for UTC time

   return utcTmStruct;
}

uint32 UtcTimestamp_diffTwoUTCTimes(const UtcDateTime *curUTCDateTime_p, const UtcDateTime *oldUTCDateTime_p)
{
   struct tm curUtcDateTime = UtcTimestamp_setTmStructFromUtcDateTimeStruct(curUTCDateTime_p);
   struct tm oldUtcDateTime = UtcTimestamp_setTmStructFromUtcDateTimeStruct(oldUTCDateTime_p);
   // Max year handled by mktime is 2037
   time_t curTime = mktime(&curUtcDateTime);
   time_t oldTime = mktime(&oldUtcDateTime);

   double utcTimeDateDiffSecondsDouble = difftime(curTime, oldTime);
   uint32 utcTimeDateDiffSeconds;
   if (utcTimeDateDiffSecondsDouble > 0)
   {
      utcTimeDateDiffSeconds = static_cast<uint32>(utcTimeDateDiffSecondsDouble);
   }
   else
   {
      utcTimeDateDiffSeconds = 0u;
   }

   return utcTimeDateDiffSeconds;
}
```

A.4.2 Driver Memory Handling

A.4.2.1 DriverMemoryHandler.h

```
#ifndef DRIVERMEMORYHANDLER_H_
#define DRIVERMEMORYHANDLER_H_

#include "DriverMemory_T.h"

#ifdef __cplusplus
extern "C" {
#endif

void DriverMemoryHandler_Init();
void DriverMemoryHandler_Exit();
void DriverMemoryHandler_MainFunction();

void DriverMemory_rqst_CallbackFunc(uint8 value);
DriverMemory_T DriverMemory_GetDriverMemory();
bool DriverMemory_GetResetFlagInstrumentView();
bool DriverMemory_GetResetFlagFocusShift();
void DriverMemory_DeactivateResetFlagInstrumentView();
void DriverMemory_DeactivateResetFlagFocusShift();

#ifdef __cplusplus
}
#endif


#endif /* DRIVERMEMORYHANDLER_H_ */
```

A.4.2.2 DriverMemoryHandler.cpp

```
#include <mutex>
#include <stdbool.h>

#include "DriverMemoryHandler.h"

// Private functions
static void activateResetFlags();

#define NUMBER_OF_USER_DEDICATED_DRIVER_MEMORIES 10u

static DriverMemory_T mActiveDriverMemory = DRIVER_MEMORY_NOT_AVAILABLE;
static DriverMemory_T mRequestedDriverMemory = DRIVER_MEMORY_NOT_AVAILABLE;

static bool mResetFlagInstrumentView = false;
static bool mResetFlagFocusShift = false;

static std::mutex mDriverMemory_rqst_mutex;


void DriverMemoryHandler_Init()
{
   std::lock_guard<std::mutex> lock(mDriverMemory_rqst_mutex);

   mActiveDriverMemory = DRIVER_MEMORY_NOT_AVAILABLE;
   mRequestedDriverMemory = DRIVER_MEMORY_NOT_AVAILABLE;

   mResetFlagInstrumentView = false;
   mResetFlagFocusShift = false;
}


void DriverMemoryHandler_Exit()
{
   std::lock_guard<std::mutex> lock(mDriverMemory_rqst_mutex);

   mActiveDriverMemory = DRIVER_MEMORY_NOT_AVAILABLE;
   mRequestedDriverMemory = DRIVER_MEMORY_NOT_AVAILABLE;

   mResetFlagInstrumentView = false;
   mResetFlagFocusShift = false;
}

// Expected to be called periodically every 5 ms
void DriverMemoryHandler_MainFunction()
{
   std::lock_guard<std::mutex> lock(mDriverMemory_rqst_mutex);
```

```
   if ((mActiveDriverMemory == mRequestedDriverMemory) &&
       (mRequestedDriverMemory <= DRIVER_MEMORY_10)) {
      return;
   }
   else if ((mRequestedDriverMemory >= DRIVER_MEMORY_RESET_1) &&
            (mRequestedDriverMemory <= DRIVER_MEMORY_RESET_10)) {
      activateResetFlags();
      mActiveDriverMemory = mRequestedDriverMemory - NUMBER_OF_USER_DEDICATED_DRIVER_MEMORIES;
      mRequestedDriverMemory = mActiveDriverMemory;
   }
   else if ((mRequestedDriverMemory >= DRIVER_MEMORY_1) &&
            (mRequestedDriverMemory <= DRIVER_MEMORY_10)) {
      mActiveDriverMemory = mRequestedDriverMemory;
   }
   else if (mRequestedDriverMemory == DRIVER_MEMORY_RESET_ALL) {
      activateResetFlags();
      mActiveDriverMemory = DRIVER_MEMORY_DEFAULT;
      mRequestedDriverMemory = DRIVER_MEMORY_DEFAULT;
   }
   else {
      mActiveDriverMemory = DRIVER_MEMORY_DEFAULT;
      mRequestedDriverMemory = DRIVER_MEMORY_DEFAULT;
   }
}


// --- Public functions ---

void DriverMemory_rqst_CallbackFunc(uint8 value) {
   std::unique_lock<std::mutex> lock(mDriverMemory_rqst_mutex);

   if (value <= DRIVER_MEMORY_RESET_ALL) {
      mRequestedDriverMemory = value;
   }
   else {
      mRequestedDriverMemory = DRIVER_MEMORY_DEFAULT;
   }
}

DriverMemory_T DriverMemory_GetDriverMemory() {
   return mActiveDriverMemory;
}

bool DriverMemory_GetResetFlagInstrumentView() {
   return mResetFlagInstrumentView;
}

bool DriverMemory_GetResetFlagFocusShift() {
   return mResetFlagFocusShift;
}

void DriverMemory_DeactivateResetFlagInstrumentView() {
   mResetFlagInstrumentView = false;
}

void DriverMemory_DeactivateResetFlagFocusShift() {
   mResetFlagFocusShift = false;
}

static void activateResetFlags() {
   mResetFlagInstrumentView = true;
   mResetFlagFocusShift = true;
}
```

A.4.3 Files for Instrument View Logging

A.4.3.1 InstrumentViewList.h

```
#pragma once

#include "Platform_Types.h"
#include <stdbool.h>
#include <stddef.h>
#include <vector>
#include <string.h>


#define NUMBER_OF_DRIVER_MEMORIES 11u
#define NUMBER_OF_INSTRUMENT_VIEWS 15u

struct InstrumentViewProperties
{
   uint32 OccurrenceCounter;  // Number of occurrences for the Instrument View
   uint32 TotalTime;          // Total time Instrument View has been used in seconds
} __attribute__((__packed__));


class InstrumentViewList
{
public:
   int readFromFile();
   int writeToFile();

   void copyToDid(uint8* to_p);
   void setFromDid(const uint8 *data);

   void clearDriverMemory(uint8 driverID);
   void clearAllDriverMemories();
   void getInstrumentViewProperties(uint8 driverID, uint8 instrumentView, InstrumentViewProperties* item_p);
   void setInstrumentViewProperties(uint8 driverID, uint8 instrumentView, InstrumentViewProperties* item_p);
   void counter(uint8 driverID, uint8 instrumentViewNumber);
   void addTime(uint8 driverID, uint8 instrumentViewNumber, uint32 timeToAdd);

   size_t size() { return sizeof(m_instrumentViewList); }

private:
   InstrumentViewProperties m_instrumentViewList[NUMBER_OF_DRIVER_MEMORIES][NUMBER_OF_INSTRUMENT_VIEWS];
};
```

A.4.3.2 InstrumentViewList.cpp

```
#include "InstrumentViewList.h"
#include "LogListBase.h"
#include "Platform_Types.h"
#include "GfxLog.h"
#include <string.h>
#include <stdbool.h>
#include <stdint.h>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <cstdint>


#if defined(HOST) || defined(UNITTEST)
#define INSTRUMENTVIEW_LIST_FILE "./test_data/instrumentview_list.bin"

#else
#define INSTRUMENTVIEW_LIST_FILE "/mnt/persistent-nand/instrumentview_list.bin"
#endif


int InstrumentViewList::readFromFile()
{
   return LogListBase::readFromFile(INSTRUMENTVIEW_LIST_FILE, (char*)m_instrumentViewList, sizeof(m_instrumentViewList));
}

int InstrumentViewList::writeToFile()
{
   return LogListBase::writeToFile(INSTRUMENTVIEW_LIST_FILE, (char*)m_instrumentViewList, sizeof(m_instrumentViewList));
}

void InstrumentViewList::copyToDid(uint8* to_p)
```

```
{
   if (to_p != nullptr)
   {
      memcpy(to_p, m_instrumentViewList, sizeof(m_instrumentViewList));
   }
}

void InstrumentViewList::setFromDid(const uint8 *data)
{
   if (data != nullptr)
   {
      memcpy(&m_instrumentViewList, data, sizeof(m_instrumentViewList));
   }
}

void InstrumentViewList::clearDriverMemory(uint8 driverID)
{
   if (driverID < NUMBER_OF_DRIVER_MEMORIES)
   {
      for (uint8 i = 0; i < NUMBER_OF_INSTRUMENT_VIEWS; i++)
      {
         m_instrumentViewList[driverID][i].OccurrenceCounter = 0u;
         m_instrumentViewList[driverID][i].TotalTime = 0u;
      }
   }
}

void InstrumentViewList::clearAllDriverMemories()
{
   for (uint8 i = 0; i < NUMBER_OF_DRIVER_MEMORIES; i++)
   {
      clearDriverMemory(i);
   }
}

void InstrumentViewList::getInstrumentViewProperties(uint8 driverID, uint8 instrumentView, InstrumentViewProperties* item_p)
{
   if ((driverID < NUMBER_OF_DRIVER_MEMORIES) && (instrumentView < NUMBER_OF_INSTRUMENT_VIEWS))
   {
      memcpy(item_p, &m_instrumentViewList[driverID][instrumentView], sizeof(InstrumentViewProperties));
   }
}

void InstrumentViewList::setInstrumentViewProperties(uint8 driverID, uint8 instrumentView, InstrumentViewProperties* item_p)
{
   if ((driverID < NUMBER_OF_DRIVER_MEMORIES) && (instrumentView < NUMBER_OF_INSTRUMENT_VIEWS))
   {
      memcpy(&m_instrumentViewList[driverID][instrumentView], item_p, sizeof(InstrumentViewProperties));
   }
}

void InstrumentViewList::counter(uint8 driverID, uint8 instrumentViewNumber)
{
   if ((driverID < NUMBER_OF_DRIVER_MEMORIES) && (instrumentViewNumber < NUMBER_OF_INSTRUMENT_VIEWS))
   {
      InstrumentViewProperties* p = &m_instrumentViewList[driverID][instrumentViewNumber];
      p->OccurrenceCounter = LogListBase::countToMaxUint32(p->OccurrenceCounter);
   }
}

void InstrumentViewList::addTime(uint8 driverID, uint8 instrumentViewNumber, uint32 timeToAdd)
{
   if ((driverID < NUMBER_OF_DRIVER_MEMORIES) && (instrumentViewNumber < NUMBER_OF_INSTRUMENT_VIEWS))
   {
      InstrumentViewProperties* p = &m_instrumentViewList[driverID][instrumentViewNumber];
      p->TotalTime = LogListBase::addTime(p->TotalTime, timeToAdd);
   }
}
```

A.4.3.3 InstrumentViewLogger.h

```
#ifndef INSTRUMENTVIEWLOGGER_H_
#define INSTRUMENTVIEWLOGGER_H_

#include "Platform_Types.h"

#ifdef __cplusplus
extern "C" {
#endif

void InstrumentViewLogger_Init();
```

```
void InstrumentViewLogger_Exit();
void InstrumentViewLogger_MainFunction();

void InstrumentViewShown_CallbackFunc(uint8 value);
void InstrumentViewLogger_CopyList(uint8* to_p);
void InstrumentViewLogger_LockAndSetList(const uint8* data);
void InstrumentViewLogger_Unlock();
uint8 InstrumentViewLogger_GetInstrumentViewShown();

#ifdef __cplusplus
}
#endif

#endif /* INSTRUMENTVIEWLOGGER_H_ */
```

A.4.3.4 InstrumentViewLogger.cpp

```
#include <mutex>
#include <stdbool.h>
#include <cstring>

#include "InstrumentViewLogger.h"
#include "InstrumentViewList.h"
#include "DriverMemoryHandler.h"
#include "VehicleMode.h"
#include "UtcTimestamp.h"


extern "C" {
#include "ApxNode_DiagCtrl.h"
#include "ApxNode_DiagCtrl_Cbk.h"
#include "TickTimer.h"
}

// Private functions
static void initLogOfInstrumentView();
static void endLogOfInstrumentView();
static void checkDriverMemoryReset();
static uint32 getLoggedTimeWithInstrumentViewShown(uint8 stopLogUtcArray[6]);
static void resetStartLogUTCArray();


#define TIMER_PERIOD_MS 100 // Expected period of MainFunction runs
#define STAYED_AT_INSTRUMENT_VIEW_TIMER_LENGTH_MS 5000
// Time needed to consider user to have stayed in Instrument View long enough to have usage counted
#define INSTRUMENT_VIEW_NOT_AVAILABLE 15u

static uint8_t mActiveInstrumentViewShown = INSTRUMENT_VIEW_NOT_AVAILABLE;
static uint8_t mRequestedInstrumentViewShown = INSTRUMENT_VIEW_NOT_AVAILABLE;
static VehicleMode_T mVehicleMode_prev = VEHICLE_MODE_UNAVAILABLE;

static InstrumentViewList mInstrumentViewList;
static bool mInstrumentViewListUpdatesBlocked;

static uint8 mStartLogUtcArray[6] = { 255, 255, 255, 255, 255, 255 };

static std::mutex mInstrumentViewShown_mutex;

static TickTimer_t mTimerStayedInInstrumentView;

static DriverMemory_T mDriverMemoryPrev = DRIVER_MEMORY_NOT_AVAILABLE;
static DriverMemory_T mDriverMemory = DRIVER_MEMORY_NOT_AVAILABLE;


void InstrumentViewLogger_Init()
{
   std::unique_lock<std::mutex> lock(mInstrumentViewShown_mutex);

   mActiveInstrumentViewShown = INSTRUMENT_VIEW_NOT_AVAILABLE;
   mRequestedInstrumentViewShown = INSTRUMENT_VIEW_NOT_AVAILABLE;

   mVehicleMode_prev = VEHICLE_MODE_UNAVAILABLE;

   mDriverMemoryPrev = DRIVER_MEMORY_NOT_AVAILABLE;
   mDriverMemory = DRIVER_MEMORY_NOT_AVAILABLE;

   mInstrumentViewListUpdatesBlocked = false;

   TickTimer_Init(&mTimerStayedInInstrumentView, STAYED_AT_INSTRUMENT_VIEW_TIMER_LENGTH_MS/TIMER_PERIOD_MS);
   resetStartLogUTCArray();
```

```
   if (mInstrumentViewList.readFromFile() == -1)
   {
      mInstrumentViewList.clearAllDriverMemories();
   }
}


void InstrumentViewLogger_Exit()
{
   std::unique_lock<std::mutex> lock(mInstrumentViewShown_mutex);

   endLogOfInstrumentView();

   mInstrumentViewList.writeToFile();
}

// Expected to be called periodically every 100 ms
void InstrumentViewLogger_MainFunction()
{
   std::unique_lock<std::mutex> lock(mInstrumentViewShown_mutex);
   VehicleMode_T vehicleMode = VehicleMode_GetVehicleMode();

   mDriverMemory = DriverMemory_GetDriverMemory();

   // Only log instrument view in VM PreRunning or higher
   if ((vehicleMode >= VEHICLE_MODE_PRERUNNING) && (vehicleMode <= VEHICLE_MODE_RUNNING))
   {
      TickTimer_Tick(&mTimerStayedInInstrumentView);

      if ((mActiveInstrumentViewShown == mRequestedInstrumentViewShown) &&
          (mRequestedInstrumentViewShown < INSTRUMENT_VIEW_NOT_AVAILABLE))
      {
         if (mDriverMemory != mDriverMemoryPrev)
         {
            endLogOfInstrumentView();
            initLogOfInstrumentView();
         }
         else if ((mVehicleMode_prev < VEHICLE_MODE_PRERUNNING) || (mVehicleMode_prev > VEHICLE_MODE_RUNNING))
         {
            initLogOfInstrumentView();
         }
         else
         {
            // No change, do nothing
         }
      }
      else if (mRequestedInstrumentViewShown < INSTRUMENT_VIEW_NOT_AVAILABLE)
      {
         // New instrument view entered, save value of previous view and init log of newly entered view
         endLogOfInstrumentView();
         initLogOfInstrumentView();

         mActiveInstrumentViewShown = mRequestedInstrumentViewShown;
      }
      else
      {
         // Incorrect instrument view value received, stop logging if ongoing
         endLogOfInstrumentView();

         mActiveInstrumentViewShown = INSTRUMENT_VIEW_NOT_AVAILABLE;
         mRequestedInstrumentViewShown = INSTRUMENT_VIEW_NOT_AVAILABLE;
      }
   }
   else if ((mVehicleMode_prev >= VEHICLE_MODE_PRERUNNING) && (mVehicleMode_prev <= VEHICLE_MODE_RUNNING))
   {
      // In lower VM -> stop logging
      endLogOfInstrumentView();

      mActiveInstrumentViewShown = INSTRUMENT_VIEW_NOT_AVAILABLE;
      mRequestedInstrumentViewShown = INSTRUMENT_VIEW_NOT_AVAILABLE;
   }
   else
   {
      // Do nothing
   }

   checkDriverMemoryReset();

   mVehicleMode_prev = vehicleMode;
   mDriverMemoryPrev = mDriverMemory;
}
```

```
void InstrumentViewShown_CallbackFunc(uint8 value) {

   if (value != INSTRUMENT_VIEW_NOT_AVAILABLE) {
      std::unique_lock<std::mutex> lock(mInstrumentViewShown_mutex);
      mRequestedInstrumentViewShown = value;
   }
}

void InstrumentViewLogger_CopyList(uint8* to_p)
{
   mInstrumentViewList.copyToDid(to_p);
}

void InstrumentViewLogger_LockAndSetList(const uint8* data)
{
   std::unique_lock<std::mutex> lock(mInstrumentViewShown_mutex);

   // Block updates from the HMI until DID have been migrated and the tester has disconnected
   mInstrumentViewListUpdatesBlocked = true;

   mInstrumentViewList.setFromDid(data);
}

void InstrumentViewLogger_Unlock()
{
   std::unique_lock<std::mutex> lock(mInstrumentViewShown_mutex);
   mInstrumentViewListUpdatesBlocked = false;
}

uint8 InstrumentViewLogger_GetInstrumentViewShown() {
   return mActiveInstrumentViewShown;
}

static void initLogOfInstrumentView() {
   // Start timer for checking long enough time spent in instrument view to be logged
   TickTimer_Start(&mTimerStayedInInstrumentView);
   // Get time stamp to know time spent in Instrument View when changing
   UtcTimestamp_getArray(mStartLogUtcArray);
}

static void endLogOfInstrumentView() {
   uint8 stopLogUtcArray[6];

   // Save time spent in previous Instrument View if long enough time was spent
   if (TickTimer_HasExpired(&mTimerStayedInInstrumentView))
   {
      // Get current time stamp to compare from when log started
      UtcTimestamp_getArray(stopLogUtcArray);
      if (UtcTimestamp_isValid(stopLogUtcArray) && UtcTimestamp_isValid(mStartLogUtcArray) &&
          !mInstrumentViewListUpdatesBlocked)
      {
         uint32 timeSpentInInstrumentViewUint32 = getLoggedTimeWithInstrumentViewShown(stopLogUtcArray);

         mInstrumentViewList.counter(mDriverMemoryPrev, mActiveInstrumentViewShown);
         mInstrumentViewList.addTime(mDriverMemoryPrev, mActiveInstrumentViewShown, timeSpentInInstrumentViewUint32);
      }
   }

   TickTimer_Stop(&mTimerStayedInInstrumentView);
   // Reset StartLogUTCArray to prevent EndTimeLog from executing again unless InitTimeLog has been called first
   resetStartLogUTCArray();
}

static void checkDriverMemoryReset() {
   bool resetDriverMemory = DriverMemory_GetResetFlagInstrumentView();

   if (resetDriverMemory && !mInstrumentViewListUpdatesBlocked)
   {
      if (mDriverMemory == DRIVER_MEMORY_DEFAULT)
      {
         mInstrumentViewList.clearAllDriverMemories();
      }
      else
      {
         mInstrumentViewList.clearDriverMemory(mDriverMemory);
      }

      DriverMemory_DeactivateResetFlagInstrumentView();
   }
}

static uint32 getLoggedTimeWithInstrumentViewShown(uint8 stopLogUtcArray[6])
{
```

```
   UtcDateTime stopLogUtcDateTime;
   UtcDateTime startLogUtcDateTime;

   UtcTimestamp_setUtcDateTimeFromUtcTimestamp(&startLogUtcDateTime, mStartLogUtcArray);
   UtcTimestamp_setUtcDateTimeFromUtcTimestamp(&stopLogUtcDateTime, stopLogUtcArray);

   return UtcTimestamp_diffTwoUTCTimes(&stopLogUtcDateTime, &startLogUtcDateTime);
}

static void resetStartLogUTCArray() {
   memset(mStartLogUtcArray, 255u, sizeof(mStartLogUtcArray));
}
```

A.4.4 Files for Focus Shift Logging

A.4.4.1 FocusShiftList.h

```
#pragma once

#include "Platform_Types.h"
#include <stdbool.h>
#include <stddef.h>


#define NUMBER_OF_DRIVER_MEMORIES 11u

struct FocusShiftListProperties
{
   uint32 FocusICTotalTime;                       // Total time Focus has been on IC in seconds
   uint32 FocusSIDTotalTime;                      // Total time Focus has been on SID in seconds
   uint32 FocusICCounterButtonPresses;            // Counter of SWS button presses while focus is on IC
   uint32 FocusSIDCounterButtonPresses;           // Counter of SWS button presses while focus is on SID
   uint16 FocusShiftActivationsCounterStandstill; // Counter of activations of Focus Shift while vehicle is at standstill
   uint16 FocusShiftActivationsCounterMoving;     // Counter of activations of Focus Shift while vehicle is moving
} __attribute__((__packed__));


class FocusShiftList
{
public:
   int readFromFile();
   int writeToFile();

   void copyToDid(uint8* to_p);
   void setFromDid(const uint8 *data);

   void clearDriverMemory(uint8 driverID);
   void clearAllDriverMemories();
   void getFocusShiftListProperties(uint8 driverID, FocusShiftListProperties* item_p);
   void setFocusShiftListProperties(uint8 driverID, FocusShiftListProperties* item_p);
   void addTimeFocusIC(uint8 driverID, uint32 timeToAdd);
   void addTimeFocusSID(uint8 driverID, uint32 timeToAdd);
   void countButtonPressFocusIC(uint8 driverID, uint8 numberOfButtonPresses);
   void countButtonPressFocusSID(uint8 driverID, uint8 numberOfButtonPresses);
   void countFocusShiftActivationStandstill(uint8 driverID);
   void countFocusShiftActivationMoving(uint8 driverID);

   size_t size() { return sizeof(m_focusShiftList); }

private:
   FocusShiftListProperties m_focusShiftList[NUMBER_OF_DRIVER_MEMORIES];
};
```

A.4.4.2 FocusShiftList.cpp

```
#include "FocusShiftList.h"
#include "LogListBase.h"
#include "Platform_Types.h"
#include "UtcTimestamp.h"
#include "GfxLog.h"
#include <string.h>
#include <stdbool.h>
#include <iostream>
#include <fstream>

#if defined(HOST) || defined(UNITTEST)
#define FOCUS_SHIFT_LIST_FILE "./test_data/focus_shift_list.bin"
#else
#define FOCUS_SHIFT_LIST_FILE "/mnt/persistent-nand/focus_shift_list.bin"
#endif


int FocusShiftList::readFromFile()
{
   return LogListBase::readFromFile(FOCUS_SHIFT_LIST_FILE, (char*)m_focusShiftList, sizeof(m_focusShiftList));
}

int FocusShiftList::writeToFile()
{
   return LogListBase::writeToFile(FOCUS_SHIFT_LIST_FILE, (char*)m_focusShiftList, sizeof(m_focusShiftList));
}
```

```
void FocusShiftList::copyToDid(uint8* to_p)
{
   if (to_p != nullptr)
   {
      memcpy(to_p, m_focusShiftList, sizeof(m_focusShiftList));
   }
}

void FocusShiftList::setFromDid(const uint8 *data)
{
   if (data != nullptr)
   {
      memcpy(&m_focusShiftList, data, sizeof(m_focusShiftList));
   }
}

void FocusShiftList::clearDriverMemory(uint8 driverID)
{
   if (driverID < NUMBER_OF_DRIVER_MEMORIES)
   {
      m_focusShiftList[driverID].FocusICTotalTime = 0u;
      m_focusShiftList[driverID].FocusSIDTotalTime = 0u;
      m_focusShiftList[driverID].FocusICCounterButtonPresses = 0u;
      m_focusShiftList[driverID].FocusSIDCounterButtonPresses = 0u;
      m_focusShiftList[driverID].FocusShiftActivationsCounterStandstill = 0u;
      m_focusShiftList[driverID].FocusShiftActivationsCounterMoving = 0u;
   }
}

void FocusShiftList::clearAllDriverMemories()
{
   for (uint8 i = 0; i < NUMBER_OF_DRIVER_MEMORIES; i++)
   {
      clearDriverMemory(i);
   }
}

void FocusShiftList::getFocusShiftListProperties(uint8 driverID, FocusShiftListProperties* item_p)
{
   if (driverID < NUMBER_OF_DRIVER_MEMORIES)
   {
      memcpy(item_p, &m_focusShiftList[driverID], sizeof(FocusShiftListProperties));
   }
}

void FocusShiftList::setFocusShiftListProperties(uint8 driverID, FocusShiftListProperties* item_p)
{
   if (driverID < NUMBER_OF_DRIVER_MEMORIES)
   {
      memcpy(&m_focusShiftList[driverID], item_p, sizeof(FocusShiftListProperties));
   }
}

void FocusShiftList::addTimeFocusIC(uint8 driverID, uint32 timeToAdd)
{
   if (driverID < NUMBER_OF_DRIVER_MEMORIES)
   {
      FocusShiftListProperties* p = &m_focusShiftList[driverID];
      p->FocusICTotalTime = LogListBase::addTime(p->FocusICTotalTime, timeToAdd);
   }
}

void FocusShiftList::addTimeFocusSID(uint8 driverID, uint32 timeToAdd)
{
   if (driverID < NUMBER_OF_DRIVER_MEMORIES)
   {
      FocusShiftListProperties* p = &m_focusShiftList[driverID];
      p->FocusSIDTotalTime = LogListBase::addTime(p->FocusSIDTotalTime, timeToAdd);
   }
}

void FocusShiftList::countButtonPressFocusIC(uint8 driverID, uint8 numberOfButtonPresses)
{
   if (driverID < NUMBER_OF_DRIVER_MEMORIES)
   {
      if (m_focusShiftList[driverID].FocusICCounterButtonPresses <= UINT32_MAX - numberOfButtonPresses)
      {
         m_focusShiftList[driverID].FocusICCounterButtonPresses += numberOfButtonPresses;
      }
      else
      {
         m_focusShiftList[driverID].FocusICCounterButtonPresses = UINT32_MAX;
      }
```

```
   }
}

void FocusShiftList::countButtonPressFocusSID(uint8 driverID, uint8 numberOfButtonPresses)
{
   if (driverID < NUMBER_OF_DRIVER_MEMORIES)
   {
      if (m_focusShiftList[driverID].FocusSIDCounterButtonPresses <= UINT32_MAX - numberOfButtonPresses)
      {
         m_focusShiftList[driverID].FocusSIDCounterButtonPresses += numberOfButtonPresses;
      }
      else
      {
         m_focusShiftList[driverID].FocusSIDCounterButtonPresses = UINT32_MAX;
      }
   }
}

void FocusShiftList::countFocusShiftActivationStandstill(uint8 driverID)
{
   if (driverID < NUMBER_OF_DRIVER_MEMORIES)
   {
      FocusShiftListProperties* p = &m_focusShiftList[driverID];
      p->FocusShiftActivationsCounterStandstill = LogListBase::countToMaxUint16(p->FocusShiftActivationsCounterStandstill);
   }
}

void FocusShiftList::countFocusShiftActivationMoving(uint8 driverID)
{
   if (driverID < NUMBER_OF_DRIVER_MEMORIES)
   {
      FocusShiftListProperties* p = &m_focusShiftList[driverID];
      p->FocusShiftActivationsCounterMoving = LogListBase::countToMaxUint16(p->FocusShiftActivationsCounterMoving);
   }
}
```

A.4.4.3 FocusShiftLogger.h

```
#ifndef FOCUSSHIFTLOGGER_H_
#define FOCUSSHIFTLOGGER_H_

#include "Platform_Types.h"

#include "DisplayFocus_T.h"

#ifdef __cplusplus
extern "C" {
#endif

void FocusShiftLogger_Init();
void FocusShiftLogger_Exit();
void FocusShiftLogger_MainFunction();

void DisplayFocusItemStat_CallbackFunc(uint8 value);
void FocusShiftActivationTriggerLog_CallbackFunc(uint8 value);
void PS_VehicleMoving_CallbackFunc(uint8 value);
void SW_Home_ButtonsStatus_6_CallbackFunc(uint8 value);
void SW_Menu_ButtonStatus_6_CallbackFunc(uint8 value);
void SW_Down_ButtonStatus_6_CallbackFunc(uint8 value);
void SW_Enter_ButtonStatus_6_CallbackFunc(uint8 value);
void SW_Esc_ButtonStatus_6_CallbackFunc(uint8 value);
void SW_Left_ButtonStatus_6_CallbackFunc(uint8 value);
void SW_Right_ButtonStatus_6_CallbackFunc(uint8 value);
void SW_Up_ButtonStatus_6_CallbackFunc(uint8 value);
void FocusShiftLogger_CopyList(uint8* to_p);
void FocusShiftLogger_LockAndSetList(const uint8* data);
void FocusShiftLogger_Unlock();
DisplayFocus_T FocusShiftLogger_GetDisplayFocus();

#ifdef __cplusplus
}
#endif



#endif /* FOCUSSHIFTLOGGER_H_ */
```

A.4.4.4 FocusShiftLogger.cpp

```
#include <mutex>
#include <stdbool.h>
#include <cstring>

#include "FocusShiftLogger.h"
#include "FocusShiftList.h"
#include "DriverMemoryHandler.h"
#include "SharedLoggingData.h"
#include "VehicleMode.h"
#include "UtcTimestamp.h"

#include "PushButtonStatus_T.h"
#include "VehicleMovingStatus_T.h"


extern "C" {
#include "ApxNode_DiagCtrl.h"
#include "ApxNode_DiagCtrl_Cbk.h"
}

// Private functions
static void initTimeLog();
static void endTimeLog();
static void checkFocusShiftActivation();
static void checkDriverMemoryReset();
static uint32 getLoggedTimeWithSetFocus(uint8 stopLogUtcArray[6]);
static void addTimeToLoggedDisplayFocus(uint32 timeSpentWithSetFocus);
static void resetStartLogUTCArray();
static void clearButtonPushes();
static void countButtonPushes();

#define MANUAL_FOCUS_SHIFT_TO_SID 2u
#define FOCUS_SHIFT_ACTIVATION_LOG_NOT_AVAILABLE 15u

#define NUMBER_OF_BUTTONS_TRACKED 8u

static FocusShiftList mFocusShiftList;
static bool mFocusShiftListUpdatesBlocked;

static std::mutex mFocusShiftLogger_mutex;

static DisplayFocus_T mActiveDisplayFocusItemStat = DISPLAY_FOCUS_NOT_AVAILABLE;
static DisplayFocus_T mRequestedDisplayFocusItemStat = DISPLAY_FOCUS_NOT_AVAILABLE;

static VehicleMovingStatus_T mVehicleMoving = VEHICLE_MOVING_STATUS_NOT_AVAILABLE;

static VehicleMode_T mVehicleMode_prev = VEHICLE_MODE_UNAVAILABLE;
static DriverMemory_T mDriverMemoryPrev = DRIVER_MEMORY_NOT_AVAILABLE;
static DriverMemory_T mDriverMemory = DRIVER_MEMORY_NOT_AVAILABLE;

static uint8 mFocusShiftActivationTriggerLog = FOCUS_SHIFT_ACTIVATION_LOG_NOT_AVAILABLE;

static PushButtonStatus_T mButtonPushedArray[NUMBER_OF_BUTTONS_TRACKED] =
      { PUSH_BUTTON_STATUS_NOT_AVAILABLE, PUSH_BUTTON_STATUS_NOT_AVAILABLE,
        PUSH_BUTTON_STATUS_NOT_AVAILABLE, PUSH_BUTTON_STATUS_NOT_AVAILABLE,
        PUSH_BUTTON_STATUS_NOT_AVAILABLE, PUSH_BUTTON_STATUS_NOT_AVAILABLE,
        PUSH_BUTTON_STATUS_NOT_AVAILABLE, PUSH_BUTTON_STATUS_NOT_AVAILABLE };

typedef enum {
   BTN_HOME,
   BTN_MENU,
   BTN_DOWN,
   BTN_ENTER,
   BTN_ESC,
   BTN_LEFT,
   BTN_RIGHT,
   BTN_UP
} ButtonE;

static uint8 mStartLogUtcArray[6] = { 255, 255, 255, 255, 255, 255 };

void FocusShiftLogger_Init()
{
   std::unique_lock<std::mutex> lock(mFocusShiftLogger_mutex);

   mActiveDisplayFocusItemStat = DISPLAY_FOCUS_NOT_AVAILABLE;
   mRequestedDisplayFocusItemStat = DISPLAY_FOCUS_NOT_AVAILABLE;

   mVehicleMoving = VEHICLE_MOVING_STATUS_NOT_AVAILABLE;

   mVehicleMode_prev = VEHICLE_MODE_UNAVAILABLE;
```

```
   mDriverMemoryPrev = DRIVER_MEMORY_NOT_AVAILABLE;
   mDriverMemory = DRIVER_MEMORY_NOT_AVAILABLE;

   mFocusShiftActivationTriggerLog = FOCUS_SHIFT_ACTIVATION_LOG_NOT_AVAILABLE;

   mFocusShiftListUpdatesBlocked = false;

   clearButtonPushes();

   resetStartLogUTCArray();

   if (mFocusShiftList.readFromFile() == -1)
   {
      mFocusShiftList.clearAllDriverMemories();
   }
}


void FocusShiftLogger_Exit()
{
   std::unique_lock<std::mutex> lock(mFocusShiftLogger_mutex);

   endTimeLog();

   mFocusShiftList.writeToFile();
}

// Expected to be called periodically every 100 ms
void FocusShiftLogger_MainFunction()
{
   std::unique_lock<std::mutex> lock(mFocusShiftLogger_mutex);
   VehicleMode_T vehicleMode = VehicleMode_GetVehicleMode();

   mDriverMemory = DriverMemory_GetDriverMemory();
   mVehicleMoving = GetVehicleMovingStatus();

   // Only log Focus Shift in VM PreRunning or higher
   if ((vehicleMode >= VEHICLE_MODE_PRERUNNING) && (vehicleMode <= VEHICLE_MODE_RUNNING))
   {
      if ((mActiveDisplayFocusItemStat == mRequestedDisplayFocusItemStat) &&
          (mRequestedDisplayFocusItemStat <= DISPLAY_FOCUS_IC))
      {
         if (mDriverMemory != mDriverMemoryPrev)
         {
            endTimeLog();
            initTimeLog();
         }
         else if ((mVehicleMode_prev < VEHICLE_MODE_PRERUNNING) || (mVehicleMode_prev > VEHICLE_MODE_RUNNING))
         {
            initTimeLog();
         }
         else
         {
            // No change, do nothing
         }
      }
      else if (mRequestedDisplayFocusItemStat <= DISPLAY_FOCUS_IC)
      {
         // Focus changed
         endTimeLog();
         initTimeLog();

         mActiveDisplayFocusItemStat = mRequestedDisplayFocusItemStat;
      }
      else
      {
         // Invalid DisplayFocusItemStat
         endTimeLog();

         mActiveDisplayFocusItemStat = DISPLAY_FOCUS_NOT_AVAILABLE;
         mRequestedDisplayFocusItemStat = DISPLAY_FOCUS_NOT_AVAILABLE;
      }

      checkFocusShiftActivation();
      countButtonPushes();
   }
   else if ((mVehicleMode_prev >= VEHICLE_MODE_PRERUNNING) && (mVehicleMode_prev <= VEHICLE_MODE_RUNNING))
   {
      // In lower VM -> stop logging
      endTimeLog();

      mActiveDisplayFocusItemStat = DISPLAY_FOCUS_NOT_AVAILABLE;
      mRequestedDisplayFocusItemStat = DISPLAY_FOCUS_NOT_AVAILABLE;
```

```
   }
   else
   {
      // Do nothing
   }

   checkDriverMemoryReset();

   clearButtonPushes();

   mVehicleMode_prev = vehicleMode;
   mDriverMemoryPrev = mDriverMemory;
}


void DisplayFocusItemStat_CallbackFunc(uint8 value) {

   std::unique_lock<std::mutex> lock(mFocusShiftLogger_mutex);

   if (value != DISPLAY_FOCUS_NOT_AVAILABLE) {
      mRequestedDisplayFocusItemStat = value;
   }
}

void FocusShiftActivationTriggerLog_CallbackFunc(uint8 value) {

   std::unique_lock<std::mutex> lock(mFocusShiftLogger_mutex);

   if (value != FOCUS_SHIFT_ACTIVATION_LOG_NOT_AVAILABLE) {
      mFocusShiftActivationTriggerLog = value;
   }
}

void SW_Home_ButtonsStatus_6_CallbackFunc(uint8 value) {

   std::unique_lock<std::mutex> lock(mFocusShiftLogger_mutex);

   if (value != PUSH_BUTTON_STATUS_NOT_AVAILABLE) {
      mButtonPushedArray[BTN_HOME] = value;
   }
}

void SW_Menu_ButtonStatus_6_CallbackFunc(uint8 value) {

   std::unique_lock<std::mutex> lock(mFocusShiftLogger_mutex);

   if (value != PUSH_BUTTON_STATUS_NOT_AVAILABLE) {
      mButtonPushedArray[BTN_MENU] = value;
   }
}

void SW_Down_ButtonStatus_6_CallbackFunc(uint8 value) {

   std::unique_lock<std::mutex> lock(mFocusShiftLogger_mutex);

   if (value != PUSH_BUTTON_STATUS_NOT_AVAILABLE) {
      mButtonPushedArray[BTN_DOWN] = value;
   }
}

void SW_Enter_ButtonStatus_6_CallbackFunc(uint8 value) {

   std::unique_lock<std::mutex> lock(mFocusShiftLogger_mutex);

   if (value != PUSH_BUTTON_STATUS_NOT_AVAILABLE) {
      mButtonPushedArray[BTN_ENTER] = value;
   }
}

void SW_Esc_ButtonStatus_6_CallbackFunc(uint8 value) {

   std::unique_lock<std::mutex> lock(mFocusShiftLogger_mutex);

   if (value != PUSH_BUTTON_STATUS_NOT_AVAILABLE) {
      mButtonPushedArray[BTN_ESC] = value;
   }
}

void SW_Left_ButtonStatus_6_CallbackFunc(uint8 value) {

   std::unique_lock<std::mutex> lock(mFocusShiftLogger_mutex);

   if (value != PUSH_BUTTON_STATUS_NOT_AVAILABLE) {
```

```
      mButtonPushedArray[BTN_LEFT] = value;
   }
}

void SW_Right_ButtonStatus_6_CallbackFunc(uint8 value) {

   std::unique_lock<std::mutex> lock(mFocusShiftLogger_mutex);

   if (value != PUSH_BUTTON_STATUS_NOT_AVAILABLE) {
      mButtonPushedArray[BTN_RIGHT] = value;
   }
}

void SW_Up_ButtonStatus_6_CallbackFunc(uint8 value) {

   std::unique_lock<std::mutex> lock(mFocusShiftLogger_mutex);

   if (value != PUSH_BUTTON_STATUS_NOT_AVAILABLE) {
      mButtonPushedArray[BTN_UP] = value;
   }
}

void FocusShiftLogger_CopyList(uint8* to_p) {
   mFocusShiftList.copyToDid(to_p);
}

void FocusShiftLogger_LockAndSetList(const uint8* data) {
   std::unique_lock<std::mutex> lock(mFocusShiftLogger_mutex);

   // Block updates from the HMI until DID have been migrated and the tester has disconnected
   mFocusShiftListUpdatesBlocked = true;

   mFocusShiftList.setFromDid(data);
}

void FocusShiftLogger_Unlock() {
   std::unique_lock<std::mutex> lock(mFocusShiftLogger_mutex);
   mFocusShiftListUpdatesBlocked = false;
}

DisplayFocus_T FocusShiftLogger_GetDisplayFocus() {
   return mActiveDisplayFocusItemStat;
}

static void initTimeLog() {
   // Get time stamp to know time spent with Focus on IC/SID when changing
   UtcTimestamp_getArray(mStartLogUtcArray);
}

static void endTimeLog() {
   uint8 stopLogUtcArray[6];

   // Get current time stamp to compare from when log started
   UtcTimestamp_getArray(stopLogUtcArray);
   if (UtcTimestamp_isValid(stopLogUtcArray) && UtcTimestamp_isValid(mStartLogUtcArray) &&
       !mFocusShiftListUpdatesBlocked)
   {
      uint32 timeSpentWithSetFocus = getLoggedTimeWithSetFocus(stopLogUtcArray);
      addTimeToLoggedDisplayFocus(timeSpentWithSetFocus);
   }

   // Reset StartLogUTCArray to prevent EndTimeLog from executing again unless InitTimeLog has been called first
   resetStartLogUTCArray();
}

static void checkFocusShiftActivation()
{
   if ((mFocusShiftActivationTriggerLog == MANUAL_FOCUS_SHIFT_TO_SID) &&
       ((!mFocusShiftListUpdatesBlocked))) // Only count Focus Shift activation when focus manually moved to SID
   {
      if (mDriverMemory <= DRIVER_MEMORY_10)
      {
         if (mVehicleMoving == VEHICLE_MOVING_STATUS_STANDSTILL)
         {
            mFocusShiftList.countFocusShiftActivationStandstill(mDriverMemory);
         }
         else if (mVehicleMoving == VEHICLE_MOVING_STATUS_MOVING)
         {
            mFocusShiftList.countFocusShiftActivationMoving(mDriverMemory);
         }
         else
         {
            // Unknown Vehicle Moving status, don't count activation
```

```
         }
      }
   }

   mFocusShiftActivationTriggerLog = FOCUS_SHIFT_ACTIVATION_LOG_NOT_AVAILABLE; // Reset variable to only count activation once
}

static void checkDriverMemoryReset() {
   bool resetDriverMemory = DriverMemory_GetResetFlagFocusShift();

   if (resetDriverMemory && !mFocusShiftListUpdatesBlocked)
   {
      if (mDriverMemory == DRIVER_MEMORY_DEFAULT)
      {
         mFocusShiftList.clearAllDriverMemories();
      }
      else
      {
         mFocusShiftList.clearDriverMemory(mDriverMemory);
      }

      DriverMemory_DeactivateResetFlagFocusShift();
   }
}

static uint32 getLoggedTimeWithSetFocus(uint8 stopLogUtcArray[6])
{
   UtcDateTime stopLogUtcDateTime;
   UtcDateTime startLogUtcDateTime;

   UtcTimestamp_setUtcDateTimeFromUtcTimestamp(&startLogUtcDateTime, mStartLogUtcArray);
   UtcTimestamp_setUtcDateTimeFromUtcTimestamp(&stopLogUtcDateTime, stopLogUtcArray);

   return UtcTimestamp_diffTwoUTCTimes(&stopLogUtcDateTime, &startLogUtcDateTime);
}

static void addTimeToLoggedDisplayFocus(uint32 timeSpentWithSetFocus)
{
   if (mActiveDisplayFocusItemStat == DISPLAY_FOCUS_IC)
   {
      mFocusShiftList.addTimeFocusIC(mDriverMemoryPrev, timeSpentWithSetFocus);
   }
   else if (mActiveDisplayFocusItemStat == DISPLAY_FOCUS_SID)
   {
      mFocusShiftList.addTimeFocusSID(mDriverMemoryPrev, timeSpentWithSetFocus);
   }
   else
   {
      // Invalid DisplayFocusItemStat, don't log counted time
   }
}

static void resetStartLogUTCArray() {
   memset(mStartLogUtcArray, 255u, sizeof(mStartLogUtcArray));
}

static void clearButtonPushes()
{
   memset(mButtonPushedArray, PUSH_BUTTON_STATUS_NEUTRAL, sizeof(mButtonPushedArray));
}

static void countButtonPushes()
{
   uint8 numberButtonPresses = 0u;

   if ((mDriverMemory <= DRIVER_MEMORY_10) && (mRequestedDisplayFocusItemStat <= DISPLAY_FOCUS_IC) &&
       (!mFocusShiftListUpdatesBlocked))
   {
      for (uint8 i = 0; i < NUMBER_OF_BUTTONS_TRACKED; i++)
      {
         if (mButtonPushedArray[i] == PUSH_BUTTON_STATUS_PUSHED)
         {
            numberButtonPresses++;
         }
      }

      if (mRequestedDisplayFocusItemStat == DISPLAY_FOCUS_IC)
      {
         mFocusShiftList.countButtonPressFocusIC(mDriverMemory, numberButtonPresses);
      }
      else
      {
         mFocusShiftList.countButtonPressFocusSID(mDriverMemory, numberButtonPresses);
```

```
    }
  }
}
```

A.4.5 Files for Pre-Trip Inspection Application Logging

A.4.5.1 PreTripInspectionList.h

```
#pragma once

#include "Platform_Types.h"
#include <stdbool.h>
#include <stddef.h>


struct PreTripInspectionListProperties
{
   uint32 PTIApplicationTotalTime;                     // Total time spent inside Pre-Trip Inspection application
   uint16 PTIApplicationEnteredCounterStandstill;      // Counter of times PTI app is entered while vehicle is at standstill
   uint16 PTIApplicationEnteredCounterMoving;          // Counter of times PTI app is entered while vehicle is moving
   uint16 ManualPTIActivationsCounter;                 // Counter of manual activations of PTI report
   uint16 FilterReportSettingChangedCounter;           // Counter of times the Filter Report setting has been changed by the user
   uint16 AutogeneratedReportSettingChangedCounter;    // Counter of times the Autogenerated Report setting has been changed by the user
   uint16 ExteriorLightsCheckSettingChangedCounter;    // Counter of times the Exterior Lights Check setting has been changed by the user
   bool   FilterReportShowAllCurrentSetting;           // Current setting for Filtering of report
   bool   AutogeneratedReportCurrentSetting;           // Current setting for Autogeneration of report
   bool   ExteriorLightsCheckCurrentSetting;           // Current setting for Exterior Lights Check
} __attribute__((__packed__));


class PreTripInspectionList
{
public:
   int readFromFile();
   int writeToFile();

   void copyToDid(uint8* to_p);
   void setFromDid(const uint8* data);

   void resetList();
   void addTimePTIApplication(uint32 timeToAdd);
   void countPTIAppEnteredStandstill();
   void countPTIAppEnteredMoving();
   void countManualPTIActivations();
   void countFilterReportSettingChanged();
   void countAutogeneratedReportSettingChanged();
   void countExteriorLightsCheckSettingChanged();
   void updateCurrentFilterReportSetting(bool currentSetting);
   void updateCurrentAutogeneratedReportSetting(bool currentSetting);
   void updateCurrentExteriorLightsCheckSetting(bool currentSetting);
   bool getFilterReportShowAllCurrentSetting();

   #ifdef UNITTEST
   void getPreTripInspectionListProperties(PreTripInspectionListProperties* item_p);
   void setPreTripInspectionListProperties(PreTripInspectionListProperties* item_p);
   size_t size() { return sizeof(m_preTripInspectionList); }
   #endif

private:
   PreTripInspectionListProperties m_preTripInspectionList;
};
```

A.4.5.2 PreTripInspectionList.cpp

```
#include "PreTripInspectionList.h"
#include "LogListBase.h"
#include "Platform_Types.h"
#include "UtcTimestamp.h"
#include "GfxLog.h"
#include <string.h>
#include <stdbool.h>
#include <iostream>
#include <fstream>

#if defined(HOST) || defined(UNITTEST)
#define PRE_TRIP_INSPECTION_LIST_FILE "./test_data/pre_trip_inspection_list.bin"
#else
#define PRE_TRIP_INSPECTION_LIST_FILE "/mnt/persistent-nand/pre_trip_inspection_list.bin"
#endif

#define STATUS_DISABLED 0u
#define PTI_FILTER_SHOW_WARNINGS 0u
```

```
int PreTripInspectionList::readFromFile()
{
   return LogListBase::readFromFile(PRE_TRIP_INSPECTION_LIST_FILE, (char*)&m_preTripInspectionList, sizeof(m_preTripInspectionList));
}

int PreTripInspectionList::writeToFile()
{
   return LogListBase::writeToFile(PRE_TRIP_INSPECTION_LIST_FILE, (char*)&m_preTripInspectionList, sizeof(m_preTripInspectionList));
}

void PreTripInspectionList::copyToDid(uint8* to_p)
{
   if (to_p != nullptr)
   {
      memcpy(to_p, &m_preTripInspectionList, sizeof(m_preTripInspectionList));
   }
}

void PreTripInspectionList::setFromDid(const uint8* data)
{
   if (data != nullptr)
   {
      memcpy(&m_preTripInspectionList, data, sizeof(m_preTripInspectionList));
   }
}

void PreTripInspectionList::resetList()
{
   m_preTripInspectionList.PTIApplicationTotalTime = 0u;
   m_preTripInspectionList.PTIApplicationEnteredCounterStandstill = 0u;
   m_preTripInspectionList.PTIApplicationEnteredCounterMoving = 0u;
   m_preTripInspectionList.ManualPTIActivationsCounter = 0u;
   m_preTripInspectionList.FilterReportSettingChangedCounter = 0u;
   m_preTripInspectionList.AutogeneratedReportSettingChangedCounter = 0u;
   m_preTripInspectionList.ExteriorLightsCheckSettingChangedCounter = 0u;
   m_preTripInspectionList.FilterReportShowAllCurrentSetting = PTI_FILTER_SHOW_WARNINGS;
   m_preTripInspectionList.AutogeneratedReportCurrentSetting = STATUS_DISABLED;
   m_preTripInspectionList.ExteriorLightsCheckCurrentSetting = STATUS_DISABLED;
}

void PreTripInspectionList::addTimePTIApplication(uint32 timeToAdd)
{
   PreTripInspectionListProperties* p = &m_preTripInspectionList;
   p->PTIApplicationTotalTime = LogListBase::addTime(p->PTIApplicationTotalTime, timeToAdd);
}

void PreTripInspectionList::countPTIAppEnteredStandstill()
{
   PreTripInspectionListProperties* p = &m_preTripInspectionList;
   p->PTIApplicationEnteredCounterStandstill = LogListBase::countToMaxUint16(p->PTIApplicationEnteredCounterStandstill);
}

void PreTripInspectionList::countPTIAppEnteredMoving()
{
   PreTripInspectionListProperties* p = &m_preTripInspectionList;
   p->PTIApplicationEnteredCounterMoving = LogListBase::countToMaxUint16(p->PTIApplicationEnteredCounterMoving);
}

void PreTripInspectionList::countManualPTIActivations()
{
   PreTripInspectionListProperties* p = &m_preTripInspectionList;
   p->ManualPTIActivationsCounter = LogListBase::countToMaxUint16(p->ManualPTIActivationsCounter);
}

void PreTripInspectionList::countFilterReportSettingChanged()
{
   PreTripInspectionListProperties* p = &m_preTripInspectionList;
   p->FilterReportSettingChangedCounter = LogListBase::countToMaxUint16(p->FilterReportSettingChangedCounter);
}

void PreTripInspectionList::countAutogeneratedReportSettingChanged()
{
   PreTripInspectionListProperties* p = &m_preTripInspectionList;
   p->AutogeneratedReportSettingChangedCounter = LogListBase::countToMaxUint16(p->AutogeneratedReportSettingChangedCounter);
}

void PreTripInspectionList::countExteriorLightsCheckSettingChanged()
{
   PreTripInspectionListProperties* p = &m_preTripInspectionList;
   p->ExteriorLightsCheckSettingChangedCounter = LogListBase::countToMaxUint16(p->ExteriorLightsCheckSettingChangedCounter);
}
```

```
void PreTripInspectionList::updateCurrentFilterReportSetting(bool currentSetting)
{
   m_preTripInspectionList.FilterReportShowAllCurrentSetting = currentSetting;
}

void PreTripInspectionList::updateCurrentAutogeneratedReportSetting(bool currentSetting)
{
   m_preTripInspectionList.AutogeneratedReportCurrentSetting = currentSetting;
}

void PreTripInspectionList::updateCurrentExteriorLightsCheckSetting(bool currentSetting)
{
   m_preTripInspectionList.ExteriorLightsCheckCurrentSetting = currentSetting;
}

bool PreTripInspectionList::getFilterReportShowAllCurrentSetting()
{
   return m_preTripInspectionList.FilterReportShowAllCurrentSetting;
}

#ifdef UNITTEST

void PreTripInspectionList::getPreTripInspectionListProperties(PreTripInspectionListProperties* item_p)
{
   memcpy(item_p, &m_preTripInspectionList, sizeof(PreTripInspectionListProperties));
}

void PreTripInspectionList::setPreTripInspectionListProperties(PreTripInspectionListProperties* item_p)
{
   memcpy(&m_preTripInspectionList, item_p, sizeof(PreTripInspectionListProperties));
}

#endif
```

A.4.5.3 PreTripInspectionLogger.h

```
#ifndef PRETRIPINSPECTIONLOGGER_H_
#define PRETRIPINSPECTIONLOGGER_H_

#include "Platform_Types.h"

#include "LoggedMenuApplicationOpen_T.h"
#include "VehicleMovingStatus_T.h"

#ifdef __cplusplus
extern "C" {
#endif

void PreTripInspectionLogger_Init();
void PreTripInspectionLogger_Exit();
void PreTripInspectionLogger_MainFunction();

void PTIFilterShowAll_CallbackFunc(uint8 value);
void ChangePTIMode_CallbackFunc(uint8 value);
void RequestToStartPTI_CallbackFunc(uint8 value);
void TogglePTILampCheck_CallbackFunc(uint8 value);
void PTIAutoStartEnabled_CallbackFunc(uint8 value);
void PTILampCheckEnabled_CallbackFunc(uint8 value);

void PreTripInspectionLogger_CopyList(uint8* to_p);
void PreTripInspectionLogger_LockAndSetList(const uint8* data);
void PreTripInspectionLogger_Unlock();

#ifdef __cplusplus
}
#endif

#endif /* PRETRIPINSPECTIONLOGGER_H_ */
```

A.4.5.4 PreTripInspectionLogger.cpp

```
#include <mutex>
#include <stdbool.h>
#include <cstring>

#include "PreTripInspectionLogger.h"
#include "PreTripInspectionList.h"
#include "SharedLoggingData.h"
```

```
#include "VehicleMode.h"
#include "UtcTimestamp.h"


extern "C" {
#include "ApxNode_DiagCtrl.h"
#include "ApxNode_DiagCtrl_Cbk.h"
}

// Private functions
static void initTimeLog();
static void endTimeLog();
static uint32 getLoggedTimeInPTIMenuApplication(uint8 stopLogUtcArray[6]);
static void addTimeLoggedInPTIMenuApplication(uint32 timeSpentWithSetFocus);
static void resetStartLogUTCArray();
static void countPTIAppEntered();
static void checkPTIReportRequest();
static void checkPTISettingsUpdates();
static void checkPTIFilterStatus();
static void resetRequestSignals();
static void updatePTISettingsStatus();

#define REQUEST_DISABLE 0u
#define REQUEST_ENABLE 1u
#define REQUEST_NOT_AVAILABLE 3u

#define STATUS_DISABLED 0u
#define STATUS_ENABLED 1u
#define STATUS_NOT_AVAILABLE 3u

#define PTI_FILTER_SHOW_WARNINGS 0u
#define PTI_FILTER_SHOW_ALL 1u
#define PTI_FILTER_NOT_AVAILABLE 3u

#define PTI_REPORT_NOT_REQUESTED 0u
#define PTI_REPORT_REQUESTED 1u
#define PTI_REPORT_REQUEST_NOT_AVAILABLE 3u

static PreTripInspectionList mPreTripInspectionList;
static bool mPreTripInspectionListUpdatesBlocked;

static std::mutex mPreTripInspectionLogger_mutex;

static LoggedMenuApplicationOpen_T mLoggedMenuApplicationOpen = LOGGED_MENU_APPLICATION_OPEN_NOT_AVAILABLE;
static LoggedMenuApplicationOpen_T mLoggedMenuApplicationOpen_prev = LOGGED_MENU_APPLICATION_OPEN_NOT_AVAILABLE;
static VehicleMode_T mVehicleMode_prev = VEHICLE_MODE_UNAVAILABLE;
static uint8 mPTIFilterShowAll = PTI_FILTER_NOT_AVAILABLE;
static uint8 mChangePTIMode = REQUEST_NOT_AVAILABLE;
static uint8 mRequestToStartPTI = PTI_REPORT_REQUEST_NOT_AVAILABLE;
static uint8 mTogglePTILampCheck = REQUEST_NOT_AVAILABLE;
static uint8 mPTIAutoStartEnabled = STATUS_NOT_AVAILABLE;
static uint8 mPTILampCheckEnabled = STATUS_NOT_AVAILABLE;

static uint8 mStartLogUtcArray[6] = { 255, 255, 255, 255, 255, 255 };

void PreTripInspectionLogger_Init()
{
   std::unique_lock<std::mutex> lock(mPreTripInspectionLogger_mutex);

   mLoggedMenuApplicationOpen = LOGGED_MENU_APPLICATION_OPEN_NOT_AVAILABLE;
   mLoggedMenuApplicationOpen_prev = LOGGED_MENU_APPLICATION_OPEN_NOT_AVAILABLE;
   mVehicleMode_prev = VEHICLE_MODE_UNAVAILABLE;

   mPTIFilterShowAll = PTI_FILTER_NOT_AVAILABLE;
   mChangePTIMode = REQUEST_NOT_AVAILABLE;
   mRequestToStartPTI = PTI_REPORT_REQUEST_NOT_AVAILABLE;
   mTogglePTILampCheck = REQUEST_NOT_AVAILABLE;
   mPTIAutoStartEnabled = STATUS_NOT_AVAILABLE;
   mPTILampCheckEnabled = STATUS_NOT_AVAILABLE;

   mPreTripInspectionListUpdatesBlocked = false;

   resetStartLogUTCArray();

   if (mPreTripInspectionList.readFromFile() == -1)
   {
      mPreTripInspectionList.resetList();
   }
}

void PreTripInspectionLogger_Exit()
{
   std::unique_lock<std::mutex> lock(mPreTripInspectionLogger_mutex);
```

```
   endTimeLog();

   mPreTripInspectionList.writeToFile();
}

// Expected to be called periodically every 100 ms
void PreTripInspectionLogger_MainFunction()
{
   std::unique_lock<std::mutex> lock(mPreTripInspectionLogger_mutex);
   VehicleMode_T vehicleMode = VehicleMode_GetVehicleMode();

   mLoggedMenuApplicationOpen = GetLoggedMenuApplicationOpen();

   // Only log Pre-Trip Inspection usage in VM PreRunning or higher
   if ((vehicleMode >= VEHICLE_MODE_PRERUNNING) && (vehicleMode <= VEHICLE_MODE_RUNNING))
   {
      if (mLoggedMenuApplicationOpen == LOGGED_MENU_APPLICATION_OPEN_PTI)
      {
         // Inside Pre-Trip Inspection Menu Application
         if (mLoggedMenuApplicationOpen_prev != LOGGED_MENU_APPLICATION_OPEN_PTI)
         {
            // PTI Menu Application Entered
            countPTIAppEntered();
            initTimeLog();
         }

         checkPTIReportRequest();
         checkPTIFilterStatus();
         checkPTISettingsUpdates();
      }
      else if ((mLoggedMenuApplicationOpen != LOGGED_MENU_APPLICATION_OPEN_PTI) &&
               (mLoggedMenuApplicationOpen_prev == LOGGED_MENU_APPLICATION_OPEN_PTI))
      {
         // PTI Menu Application Exited
         endTimeLog();
      }
      else
      {
         // Not inside PTI Menu Application, do nothing
      }
   }
   else if ((mVehicleMode_prev >= VEHICLE_MODE_PRERUNNING) && (mVehicleMode_prev <= VEHICLE_MODE_RUNNING))
   {
      // In lower VM -> stop logging
      endTimeLog();

      mLoggedMenuApplicationOpen = LOGGED_MENU_APPLICATION_OPEN_NOT_AVAILABLE;
      mLoggedMenuApplicationOpen_prev = LOGGED_MENU_APPLICATION_OPEN_NOT_AVAILABLE;
   }
   else
   {
      // Do nothing
   }

   updatePTISettingsStatus();
   resetRequestSignals();

   mVehicleMode_prev = vehicleMode;
   mLoggedMenuApplicationOpen_prev = mLoggedMenuApplicationOpen;
}


void PTIFilterShowAll_CallbackFunc(uint8 value) {

   std::unique_lock<std::mutex> lock(mPreTripInspectionLogger_mutex);

   mPTIFilterShowAll= value;
}

void ChangePTIMode_CallbackFunc(uint8 value) {

   std::unique_lock<std::mutex> lock(mPreTripInspectionLogger_mutex);

   mChangePTIMode = value;
}

void RequestToStartPTI_CallbackFunc(uint8 value) {

   std::unique_lock<std::mutex> lock(mPreTripInspectionLogger_mutex);

   mRequestToStartPTI = value;
}
```

```
void TogglePTILampCheck_CallbackFunc(uint8 value) {

   std::unique_lock<std::mutex> lock(mPreTripInspectionLogger_mutex);

   mTogglePTILampCheck = value;
}

void PTIAutoStartEnabled_CallbackFunc(uint8 value) {

   std::unique_lock<std::mutex> lock(mPreTripInspectionLogger_mutex);

   mPTIAutoStartEnabled = value;
}

void PTILampCheckEnabled_CallbackFunc(uint8 value) {

   std::unique_lock<std::mutex> lock(mPreTripInspectionLogger_mutex);

   mPTILampCheckEnabled = value;
}

void PreTripInspectionLogger_CopyList(uint8* to_p) {
   mPreTripInspectionList.copyToDid(to_p);
}

void PreTripInspectionLogger_LockAndSetList(const uint8* data) {
   std::unique_lock<std::mutex> lock(mPreTripInspectionLogger_mutex);

   // Block updates from the HMI until DID have been migrated and the tester has disconnected
   mPreTripInspectionListUpdatesBlocked = true;

   mPreTripInspectionList.setFromDid(data);
}

void PreTripInspectionLogger_Unlock() {
   std::unique_lock<std::mutex> lock(mPreTripInspectionLogger_mutex);
   mPreTripInspectionListUpdatesBlocked = false;
}

//////////////////////////////////////////////////
/////// Private functions
//////////////////////////////////////////////////

static void initTimeLog() {
   // Get time stamp to know time spent in PTI Menu Application
   UtcTimestamp_getArray(mStartLogUtcArray);
}

static void endTimeLog() {
   uint8 stopLogUtcArray[6];

   // Get current time stamp to compare from when log started
   UtcTimestamp_getArray(stopLogUtcArray);
   if (UtcTimestamp_isValid(stopLogUtcArray) && UtcTimestamp_isValid(mStartLogUtcArray) &&
       !mPreTripInspectionListUpdatesBlocked)
   {
      uint32 timeSpentInPTIMenuApplication = getLoggedTimeInPTIMenuApplication(stopLogUtcArray);
      addTimeLoggedInPTIMenuApplication(timeSpentInPTIMenuApplication);
   }

   // Reset StartLogUTCArray to prevent EndTimeLog from executing again unless InitTimeLog has been called first
   resetStartLogUTCArray();
}

static uint32 getLoggedTimeInPTIMenuApplication(uint8 stopLogUtcArray[6])
{
   UtcDateTime stopLogUtcDateTime;
   UtcDateTime startLogUtcDateTime;

   UtcTimestamp_setUtcDateTimeFromUtcTimestamp(&startLogUtcDateTime, mStartLogUtcArray);
   UtcTimestamp_setUtcDateTimeFromUtcTimestamp(&stopLogUtcDateTime, stopLogUtcArray);

   return UtcTimestamp_diffTwoUTCTimes(&stopLogUtcDateTime, &startLogUtcDateTime);
}

static void addTimeLoggedInPTIMenuApplication(uint32 timeSpentInPTIMenuApplication)
{
   mPreTripInspectionList.addTimePTIApplication(timeSpentInPTIMenuApplication);
}

static void resetStartLogUTCArray() {
   memset(mStartLogUtcArray, 255u, sizeof(mStartLogUtcArray));
```

```
}

static void countPTIAppEntered() {
   if (!mPreTripInspectionListUpdatesBlocked)
   {
      VehicleMovingStatus_T vehicleMovingStatus = GetVehicleMovingStatus();

      if (vehicleMovingStatus == VEHICLE_MOVING_STATUS_STANDSTILL)
      {
         mPreTripInspectionList.countPTIAppEnteredStandstill();
      }
      else if (vehicleMovingStatus == VEHICLE_MOVING_STATUS_MOVING)
      {
         mPreTripInspectionList.countPTIAppEnteredMoving();
      }
      else
      {
         // Unknown Vehicle Moving Status, don't count entering menu application
      }
   }
}

static void checkPTIReportRequest() {
   if (!mPreTripInspectionListUpdatesBlocked)
   {
      if (mRequestToStartPTI == PTI_REPORT_REQUESTED)
      {
         mPreTripInspectionList.countManualPTIActivations();
      }
   }
}

static void checkPTISettingsUpdates() {
   if (!mPreTripInspectionListUpdatesBlocked)
   {
      if (mChangePTIMode <= REQUEST_ENABLE)
      {
         mPreTripInspectionList.countAutogeneratedReportSettingChanged();
      }

      if (mTogglePTILampCheck <= REQUEST_ENABLE)
      {
         mPreTripInspectionList.countExteriorLightsCheckSettingChanged();
      }
   }
}

static void checkPTIFilterStatus() {
   if (!mPreTripInspectionListUpdatesBlocked)
   {
      if ((mPTIFilterShowAll <= PTI_FILTER_SHOW_ALL) &&
          (mPTIFilterShowAll != mPreTripInspectionList.getFilterReportShowAllCurrentSetting()))
      {
         mPreTripInspectionList.countFilterReportSettingChanged();
         mPreTripInspectionList.updateCurrentFilterReportSetting(mPTIFilterShowAll);
      }
   }
}

static void resetRequestSignals() {
   mChangePTIMode = REQUEST_NOT_AVAILABLE;
   mTogglePTILampCheck = REQUEST_NOT_AVAILABLE;
   mRequestToStartPTI = PTI_REPORT_REQUEST_NOT_AVAILABLE;
}

static void updatePTISettingsStatus() {
   if (!mPreTripInspectionListUpdatesBlocked)
   {
      if (mPTIAutoStartEnabled <= STATUS_ENABLED)
      {
         mPreTripInspectionList.updateCurrentAutogeneratedReportSetting(mPTIAutoStartEnabled);
         mPTIAutoStartEnabled = STATUS_NOT_AVAILABLE;
      }

      if (mPTILampCheckEnabled <= STATUS_ENABLED)
      {
         mPreTripInspectionList.updateCurrentExteriorLightsCheckSetting(mPTILampCheckEnabled);
         mPTILampCheckEnabled = STATUS_NOT_AVAILABLE;
      }
   }
}
```

A.4.6 Unit Tests for Focus Shift Logging

A.4.6.1 Test_FocusShiftList.cpp

```
#include "gtest/gtest.h"
#include "FocusShiftList.h"
#include <unistd.h>
#include <iostream>
#include <fstream>


#define FOCUS_SHIFT_LIST_PATH "./test_data/"
#define FOCUS_SHIFT_LIST_FILE FOCUS_SHIFT_LIST_PATH "focus_shift_list.bin"

extern "C"
{

#include <DriverMemory_T.h>

}


using namespace ::testing;

FocusShiftList focusShiftList;

class Test_FocusShiftListItem : public ::testing::Test {
protected:
   Test_FocusShiftListItem() {}

   virtual ~Test_FocusShiftListItem() {}

   virtual void SetUp()
   {
      focusShiftList.clearAllDriverMemories();
   }
};

class Test_FocusShiftList : public ::testing::Test {
protected:
   Test_FocusShiftList() {}

   virtual ~Test_FocusShiftList() {}

   virtual void SetUp()
   {
      focusShiftList.clearAllDriverMemories();
   }
};

class Test_FocusShiftListFile : public ::testing::Test {
protected:
   Test_FocusShiftListFile() {}

   virtual ~Test_FocusShiftListFile() {}

   static void SetUpTestCase()
   {
      ::mkdir(FOCUS_SHIFT_LIST_PATH, 0755);
   }

   virtual void SetUp()
   {
      ::remove(FOCUS_SHIFT_LIST_FILE);
      focusShiftList.clearAllDriverMemories(); // Start all tests with a cleared list
   }
};


bool areFocusShiftListPropertiesEqual(FocusShiftListProperties* item1, FocusShiftListProperties* item2)
{
   return (memcmp(item1, item2, sizeof(FocusShiftListProperties)) == 0);
}

bool isListEqualToFocusShiftList(FocusShiftListProperties* list, uint8 startIdx, uint8 endIdx)
{
   bool areEqual = true;

   for (uint8 i = startIdx; i <= endIdx; i++)
   {
      FocusShiftListProperties item;
```

```
      focusShiftList.getFocusShiftListProperties(i, &item);
      if (!areFocusShiftListPropertiesEqual(list++, &item))
      {
         areEqual = false;
      }
   }
   return areEqual;
}

void fillUpList(FocusShiftListProperties* list_p)
{
   for (uint8 i = 0; i < NUMBER_OF_DRIVER_MEMORIES; i++)
   {
      focusShiftList.setFocusShiftListProperties(i, &list_p[i]);
   }
}

size_t readFileFocusShiftListBin(char* buf, size_t len)
{
   std::ifstream inFile(FOCUS_SHIFT_LIST_FILE, std::ifstream::binary | std::ifstream::ate);
   size_t size = inFile.tellg();
   inFile.seekg(0, std::ifstream::beg);
   inFile.read(buf, len);
   inFile.close();

   return size;
}


// Predefined items for the test cases
static FocusShiftListProperties itemCleared = { 0, 0, 0, 0, 0, 0 };
static FocusShiftListProperties item1 = { 111, 222, 333, 444, 555, 666 };
static FocusShiftListProperties item2 = { 123, 234, 345, 456, 567, 678 };
static FocusShiftListProperties itemCountersAlmostMax =
{ UINT32_MAX - 95, UINT32_MAX - 195, UINT32_MAX - 1, UINT32_MAX - 1, UINT16_MAX - 1, UINT16_MAX - 1 };
static FocusShiftListProperties itemCountersMax =
{ UINT32_MAX, UINT32_MAX, UINT32_MAX, UINT32_MAX, UINT16_MAX, UINT16_MAX };

// Predefined focusShiftLists for the test cases
static FocusShiftListProperties listBase[NUMBER_OF_DRIVER_MEMORIES] =
{
      { 100, 34, 4, 28, 12, 10 },
      { 101, 34, 4, 28, 12, 11 },
      { 102, 34, 4, 28, 12, 12 },
      { 103, 34, 4, 28, 12, 13 },
      { 104, 34, 4, 28, 12, 14 },
      { 105, 34, 4, 28, 12, 15 },
      { 106, 34, 4, 28, 12, 16 },
      { 107, 34, 4, 28, 12, 17 },
      { 108, 34, 4, 28, 12, 18 },
      { 109, 34, 4, 28, 12, 19 },
      { 109, 34, 4, 28, 12, 20 },
};

const size_t expectedFocusShiftListSize = 11 * 20; // 11 rows of 20 (4 + 4 + 4 + 4 + 2 + 2) bytes

////////////////////////////////////////////////
/// Test Focus Shift Properties Handling
////////////////////////////////////////////////

TEST_F(Test_FocusShiftListItem, CompareClearedList)
{
   FocusShiftListProperties item;
   for (uint8 i = 0; i < NUMBER_OF_DRIVER_MEMORIES; i++)
   {
      focusShiftList.getFocusShiftListProperties(i, &item);
      EXPECT_TRUE(areFocusShiftListPropertiesEqual(&item, &itemCleared));
   }
}

TEST_F(Test_FocusShiftListItem, SetAndGetItem)
{
   focusShiftList.setFocusShiftListProperties(DRIVER_MEMORY_DEFAULT, &item1);
   focusShiftList.setFocusShiftListProperties(NUMBER_OF_DRIVER_MEMORIES - 1, &item2);

   FocusShiftListProperties item;

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_DEFAULT, &item);
   EXPECT_TRUE(areFocusShiftListPropertiesEqual(&item, &item1));

   focusShiftList.getFocusShiftListProperties(NUMBER_OF_DRIVER_MEMORIES - 1, &item);
   EXPECT_TRUE(areFocusShiftListPropertiesEqual(&item, &item2));
}
```

```
TEST_F(Test_FocusShiftListItem, ClearDriverMemory)
{
   focusShiftList.setFocusShiftListProperties(DRIVER_MEMORY_7, &item2);

   FocusShiftListProperties item;

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_7, &item);
   EXPECT_TRUE(areFocusShiftListPropertiesEqual(&item, &item2));

   focusShiftList.clearDriverMemory(DRIVER_MEMORY_7);

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_7, &item);
   EXPECT_TRUE(areFocusShiftListPropertiesEqual(&item, &itemCleared));
}

TEST_F(Test_FocusShiftListItem, ClearAllDriverMemory)
{
   focusShiftList.setFocusShiftListProperties(DRIVER_MEMORY_4, &item1);
   focusShiftList.setFocusShiftListProperties(DRIVER_MEMORY_5, &item2);

   FocusShiftListProperties item;

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_4, &item);
   EXPECT_TRUE(areFocusShiftListPropertiesEqual(&item, &item1));

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_5, &item);
   EXPECT_TRUE(areFocusShiftListPropertiesEqual(&item, &item2));

   focusShiftList.clearAllDriverMemories();

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_4, &item);
   EXPECT_TRUE(areFocusShiftListPropertiesEqual(&item, &itemCleared));

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_5, &item);
   EXPECT_TRUE(areFocusShiftListPropertiesEqual(&item, &itemCleared));
}

TEST_F(Test_FocusShiftListItem, setAndGetItem_idxOutOfRange)
{
   focusShiftList.setFocusShiftListProperties(NUMBER_OF_DRIVER_MEMORIES, &item2);

   FocusShiftListProperties item = { 0, 0, 0, 0, 0, 0 };

   focusShiftList.getFocusShiftListProperties(NUMBER_OF_DRIVER_MEMORIES, &item);
   EXPECT_TRUE(areFocusShiftListPropertiesEqual(&item, &itemCleared));
}

TEST_F(Test_FocusShiftListItem, AddTimeFocusIC)
{
   focusShiftList.addTimeFocusIC(DRIVER_MEMORY_2, 200);

   FocusShiftListProperties item = { 0, 0, 0, 0, 0, 0 };

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_2, &item);
   EXPECT_EQ(item.FocusICTotalTime, 200u);
}

TEST_F(Test_FocusShiftListItem, AddTimeFocusIC_OverflowProtection)
{
   FocusShiftListProperties item = { 0, 0, 0, 0, 0, 0 };

   focusShiftList.setFocusShiftListProperties(DRIVER_MEMORY_6, &itemCountersMax);

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_6, &item);
   EXPECT_EQ(item.FocusICTotalTime, UINT32_MAX);

   focusShiftList.addTimeFocusIC(DRIVER_MEMORY_6, 1u);

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_6, &item);
   EXPECT_EQ(item.FocusICTotalTime, UINT32_MAX);
}

TEST_F(Test_FocusShiftListItem, AddTimeFocusSID)
{
   focusShiftList.addTimeFocusSID(DRIVER_MEMORY_DEFAULT, 1337);

   FocusShiftListProperties item = { 0, 0, 0, 0, 0, 0 };

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_DEFAULT, &item);
   EXPECT_EQ(item.FocusSIDTotalTime, 1337u);

}
```

```
TEST_F(Test_FocusShiftListItem, AddTimeFocusSID_OverflowProtection)
{
   FocusShiftListProperties item = { 0, 0, 0, 0, 0, 0 };

   focusShiftList.setFocusShiftListProperties(DRIVER_MEMORY_9, &itemCountersMax);

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_9, &item);
   EXPECT_EQ(item.FocusSIDTotalTime, UINT32_MAX);

   focusShiftList.addTimeFocusSID(DRIVER_MEMORY_9, 1);

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_9, &item);
   EXPECT_EQ(item.FocusSIDTotalTime, UINT32_MAX);
}

TEST_F(Test_FocusShiftListItem, CountButtonPressesFocusIC)
{
   focusShiftList.countButtonPressFocusIC(DRIVER_MEMORY_3, 4u);

   FocusShiftListProperties item = { 0, 0, 0, 0, 0, 0 };

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_3, &item);
   EXPECT_EQ(item.FocusICCounterButtonPresses, 4u);
}

TEST_F(Test_FocusShiftListItem, CountButtonPressFocusIC_OverflowProtection)
{
   FocusShiftListProperties item = { 0, 0, 0, 0, 0, 0 };

   focusShiftList.setFocusShiftListProperties(DRIVER_MEMORY_4, &itemCountersAlmostMax);

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_4, &item);
   EXPECT_EQ(item.FocusICCounterButtonPresses, UINT32_MAX-1);

   focusShiftList.countButtonPressFocusIC(DRIVER_MEMORY_4, 1u);

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_4, &item);
   EXPECT_EQ(item.FocusICCounterButtonPresses, UINT32_MAX);

   focusShiftList.countButtonPressFocusIC(DRIVER_MEMORY_4, 1u);

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_4, &item);
   EXPECT_EQ(item.FocusICCounterButtonPresses, UINT32_MAX);
}

TEST_F(Test_FocusShiftListItem, CountButtonPressesFocusSID)
{
   focusShiftList.countButtonPressFocusSID(DRIVER_MEMORY_8, 7u);

   FocusShiftListProperties item = { 0, 0, 0, 0, 0, 0 };

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_8, &item);
   EXPECT_EQ(item.FocusSIDCounterButtonPresses, 7u);
}

TEST_F(Test_FocusShiftListItem, CountButtonPressFocusSID_OverflowProtection)
{
   FocusShiftListProperties item = { 0, 0, 0, 0, 0, 0 };

   focusShiftList.setFocusShiftListProperties(DRIVER_MEMORY_1, &itemCountersAlmostMax);

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_1, &item);
   EXPECT_EQ(item.FocusSIDCounterButtonPresses, UINT32_MAX-1);

   focusShiftList.countButtonPressFocusSID(DRIVER_MEMORY_1, 1u);

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_1, &item);
   EXPECT_EQ(item.FocusSIDCounterButtonPresses, UINT32_MAX);

   focusShiftList.countButtonPressFocusSID(DRIVER_MEMORY_1, 1u);

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_1, &item);
   EXPECT_EQ(item.FocusSIDCounterButtonPresses, UINT32_MAX);
}

TEST_F(Test_FocusShiftListItem, CountFocusShiftActivationStandstill)
{
   focusShiftList.countFocusShiftActivationStandstill(DRIVER_MEMORY_5);

   FocusShiftListProperties item = { 0, 0, 0, 0, 0, 0 };

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_5, &item);
```

```
   EXPECT_EQ(item.FocusShiftActivationsCounterStandstill, 1);
}

TEST_F(Test_FocusShiftListItem, CountFocusShiftActivationStandstill_OverflowProtection)
{
   FocusShiftListProperties item = { 0, 0, 0, 0, 0, 0 };

   focusShiftList.setFocusShiftListProperties(DRIVER_MEMORY_10, &itemCountersAlmostMax);

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_10, &item);
   EXPECT_EQ(item.FocusShiftActivationsCounterStandstill, UINT16_MAX-1);

   focusShiftList.countFocusShiftActivationStandstill(DRIVER_MEMORY_10);

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_10, &item);
   EXPECT_EQ(item.FocusShiftActivationsCounterStandstill, UINT16_MAX);

   focusShiftList.countFocusShiftActivationStandstill(DRIVER_MEMORY_10);

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_10, &item);
   EXPECT_EQ(item.FocusShiftActivationsCounterStandstill, UINT16_MAX);
}

TEST_F(Test_FocusShiftListItem, CountFocusShiftActivationMoving)
{
   focusShiftList.countFocusShiftActivationMoving(DRIVER_MEMORY_10);

   FocusShiftListProperties item = { 0, 0, 0, 0, 0, 0 };

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_10, &item);
   EXPECT_EQ(item.FocusShiftActivationsCounterMoving, 1);
}

TEST_F(Test_FocusShiftListItem, CountFocusShiftActivationMoving_OverflowProtection)
{
   FocusShiftListProperties item = { 0, 0, 0, 0, 0, 0 };

   focusShiftList.setFocusShiftListProperties(DRIVER_MEMORY_6, &itemCountersAlmostMax);

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_6, &item);
   EXPECT_EQ(item.FocusShiftActivationsCounterMoving, UINT16_MAX-1);

   focusShiftList.countFocusShiftActivationMoving(DRIVER_MEMORY_6);

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_6, &item);
   EXPECT_EQ(item.FocusShiftActivationsCounterMoving, UINT16_MAX);

   focusShiftList.countFocusShiftActivationMoving(DRIVER_MEMORY_6);

   focusShiftList.getFocusShiftListProperties(DRIVER_MEMORY_6, &item);
   EXPECT_EQ(item.FocusShiftActivationsCounterMoving, UINT16_MAX);
}

////////////////////////////////////////////////
/// Test List Handling
////////////////////////////////////////////////

TEST_F(Test_FocusShiftList, Size)
{
   EXPECT_EQ(focusShiftList.size(), expectedFocusShiftListSize);
}

TEST_F(Test_FocusShiftList, copyToDid)
{
   focusShiftList.setFocusShiftListProperties(DRIVER_MEMORY_7, &item1);
   focusShiftList.setFocusShiftListProperties(DRIVER_MEMORY_7, &item2);

   FocusShiftListProperties focusShiftListCopy[NUMBER_OF_DRIVER_MEMORIES];
   focusShiftList.copyToDid((uint8*)&focusShiftListCopy);

   EXPECT_TRUE(isListEqualToFocusShiftList(&focusShiftListCopy[0], 0, NUMBER_OF_DRIVER_MEMORIES - 1));
}

TEST_F(Test_FocusShiftList, setFromDid)
{
   fillUpList(listBase);
   FocusShiftListProperties focusShiftListCopy[NUMBER_OF_DRIVER_MEMORIES];
   memset(focusShiftListCopy, 0xaa, sizeof(focusShiftListCopy));
   focusShiftList.setFromDid((uint8*) &focusShiftListCopy);
   EXPECT_EQ(expectedFocusShiftListSize, focusShiftList.size());
}

////////////////////////////////////////////////
```

```
/// Test File Handling
////////////////////////////////////////////////

TEST_F(Test_FocusShiftListFile, readFromFile_fileNotExisting)
{
   EXPECT_EQ(focusShiftList.readFromFile(), -1);
}

TEST_F(Test_FocusShiftListFile, readFromFile_fileWrongSize)
{
   // Create empty file and check that it exists
   std::ofstream outFile(FOCUS_SHIFT_LIST_FILE, std::ofstream::binary);
   outFile.close();
   ASSERT_EQ(::access(FOCUS_SHIFT_LIST_FILE, F_OK), 0);

   EXPECT_EQ(focusShiftList.readFromFile(), -1);
}

TEST_F(Test_FocusShiftListFile, readFromFile_correctSizeValidData)
{
   // Create file containing valid data and check that it exists
   std::ofstream outFile(FOCUS_SHIFT_LIST_FILE, std::ofstream::binary);
   outFile.write((char*)listBase, sizeof(listBase));
   outFile.close();
   ASSERT_EQ(::access(FOCUS_SHIFT_LIST_FILE, F_OK), 0);

   EXPECT_EQ(focusShiftList.readFromFile(), 0);

   EXPECT_TRUE(isListEqualToFocusShiftList(listBase,
         0 , NUMBER_OF_DRIVER_MEMORIES - 1));
}

TEST_F(Test_FocusShiftListFile, writeToFile_nonExistingPath)
{
   EXPECT_EQ(rmdir(FOCUS_SHIFT_LIST_PATH), 0);

   EXPECT_EQ(focusShiftList.writeToFile(), -1);

   // Cleanup
   EXPECT_EQ(::mkdir(FOCUS_SHIFT_LIST_PATH, 0755), 0);
}

TEST_F(Test_FocusShiftListFile, writeToFile_clearedList)
{
   EXPECT_EQ(focusShiftList.writeToFile(), 0);

   // Read in file to separate list
   FocusShiftListProperties focusShiftListCopy[NUMBER_OF_DRIVER_MEMORIES];
   size_t fileSize = readFileFocusShiftListBin(
         (char*)focusShiftListCopy, sizeof(focusShiftListCopy));

   EXPECT_EQ(fileSize, expectedFocusShiftListSize);

   for (uint8 i = 0; i < NUMBER_OF_DRIVER_MEMORIES; i++)
   {
      EXPECT_TRUE(areFocusShiftListPropertiesEqual(&itemCleared, &focusShiftListCopy[i]));
   }
}

TEST_F(Test_FocusShiftListFile, writeToFile_filledList)
{
   fillUpList(listBase);

   EXPECT_EQ(focusShiftList.writeToFile(), 0);

   // Read in file to separate list
   FocusShiftListProperties focusShiftListCopy[NUMBER_OF_DRIVER_MEMORIES];
   size_t fileSize = readFileFocusShiftListBin(
         (char*)focusShiftListCopy, sizeof(focusShiftListCopy));

   EXPECT_EQ(fileSize, expectedFocusShiftListSize);

   EXPECT_TRUE(isListEqualToFocusShiftList(listBase,
         0 , NUMBER_OF_DRIVER_MEMORIES - 1));
}
```

A.4.6.2 Test_FocusShiftLogger.cpp

```
#include "gtest/gtest.h"
```

```
#include <string.h>
#include <unistd.h>
#include <iostream>
#include <fstream>

// File under test
#include "../Source/FocusShiftLogger.cpp"

// Function prototypes
void UTCTimestamp_CallbackFunc(uint8 *value);
void PS_VehicleMoving_CallbackFunc(uint8 value);

extern "C"
{
#include "DriverMemory_T.h"
#include "VehicleMode_T.h"
#include "VehicleMovingStatus_T.h"
#include "PushButtonStatus_T.h"
#include "DisplayFocus_T.h"

/*
 *  MOCKS
 */

static uint32 mockVehicleMode;
static uint16 callCount_VehicleMode_GetVehicleMode;

VehicleMode_T VehicleMode_GetVehicleMode(void)
{
   ++callCount_VehicleMode_GetVehicleMode;
   return mockVehicleMode;
}

}


static DriverMemory_T mockDriverMemory;
static uint16 callCount_DriverMemory_GetDriverMemory;

DriverMemory_T DriverMemory_GetDriverMemory(void)
{
   ++callCount_DriverMemory_GetDriverMemory;
   return mockDriverMemory;
}

static bool mockResetFlagFocusShift;
static uint16 callCount_DriverMemory_GetResetFlagFocusShift;
static uint16 callCount_DriverMemory_DeactivateResetFlagFocusShift;

bool DriverMemory_GetResetFlagFocusShift(void)
{
   ++callCount_DriverMemory_GetResetFlagFocusShift;
   return mockResetFlagFocusShift;
}

void DriverMemory_DeactivateResetFlagFocusShift() {
   ++callCount_DriverMemory_DeactivateResetFlagFocusShift;
   mockResetFlagFocusShift = false;
}


using namespace ::testing;


#define FOCUS_SHIFT_LIST_PATH "./test_data/"
#define FOCUS_SHIFT_LIST_FILE FOCUS_SHIFT_LIST_PATH "focus_shift_list.bin"

// Predefined items for the test cases
#define EXPECTED_LIST_SIZE 220u // 11*(4+4+4+4+2+2)

#define MANUAL_FOCUS_SHIFT_TO_IC 1u
#define AUTO_FOCUS_SHIFT_TO_IC 3u
#define AUTO_FOCUS_SHIFT_TO_SID 4u

// Predefined UtcTimestamps for the test cases
static uint8 utcTimestamp1[] = { 34, 04, 114, 12, 57, 154 }; // 2019-04-28 12:57:38.500
static uint8 utcTimestamp2[] = { 34, 04, 114, 14, 28, 126 }; // 2019-04-28 14:28:31.500
static uint8 utcTimestampInvalid[] = { 251, 36, 255, 43, 160, 2 }; // Invalid
static uint8 utcTimestampResetValue[] = { 255, 255, 255, 255, 255, 255 }; // All values upper limit

// Predefined items for the test cases
static FocusShiftListProperties item1 = { 111, 222, 333, 444, 555, 666 };
```

```
class Test_FocusShiftLogger : public ::testing::Test
{
protected:
   Test_FocusShiftLogger() {}

   virtual ~Test_FocusShiftLogger() {}

   static void SetUpTestCase()
   {
      ::mkdir(FOCUS_SHIFT_LIST_PATH, 0755);
      ::remove(FOCUS_SHIFT_LIST_FILE);
   }

   virtual void SetUp()
   {
      mFocusShiftList.clearAllDriverMemories();

      callCount_VehicleMode_GetVehicleMode = 0;
      callCount_DriverMemory_GetDriverMemory = 0;
      callCount_DriverMemory_GetResetFlagFocusShift = 0;
      callCount_DriverMemory_DeactivateResetFlagFocusShift = 0;

      UTCTimestamp_CallbackFunc(utcTimestampResetValue);
      mockVehicleMode = VEHICLE_MODE_UNAVAILABLE;
      mockDriverMemory = DRIVER_MEMORY_NOT_AVAILABLE;
      mockResetFlagFocusShift = false;

      FocusShiftLogger_Init();
   }

   virtual void TearDown()
   {
   }
};


////////////////////////////////////////////////
/// Test Focus Shift Logger
////////////////////////////////////////////////

TEST_F(Test_FocusShiftLogger, Init)
{
   EXPECT_EQ(mVehicleMode_prev, VEHICLE_MODE_UNAVAILABLE);

   EXPECT_EQ(mActiveDisplayFocusItemStat, DISPLAY_FOCUS_NOT_AVAILABLE);
   EXPECT_EQ(mRequestedDisplayFocusItemStat, DISPLAY_FOCUS_NOT_AVAILABLE);

   EXPECT_EQ(mDriverMemoryPrev, DRIVER_MEMORY_NOT_AVAILABLE);
   EXPECT_EQ(mDriverMemory, DRIVER_MEMORY_NOT_AVAILABLE);
}

TEST_F(Test_FocusShiftLogger, Exit)
{
   FocusShiftLogger_Exit();

   FocusShiftListProperties focusShiftListCopy[NUMBER_OF_DRIVER_MEMORIES];
   std::ifstream inFile(FOCUS_SHIFT_LIST_FILE, std::ifstream::binary | std::ifstream::ate);
   size_t size = inFile.tellg();
   inFile.seekg(0, std::ifstream::beg);
   inFile.read((char*)focusShiftListCopy, sizeof(focusShiftListCopy));
   inFile.close();

   EXPECT_EQ(size, EXPECTED_LIST_SIZE);

   for (uint8 i = 0; i < NUMBER_OF_DRIVER_MEMORIES; i++)
   {
      EXPECT_EQ(focusShiftListCopy[i].FocusICTotalTime, 0u);
      EXPECT_EQ(focusShiftListCopy[i].FocusSIDTotalTime, 0u);
      EXPECT_EQ(focusShiftListCopy[i].FocusICCounterButtonPresses, 0u);
      EXPECT_EQ(focusShiftListCopy[i].FocusSIDCounterButtonPresses, 0u);
      EXPECT_EQ(focusShiftListCopy[i].FocusShiftActivationsCounterStandstill, 0u);
      EXPECT_EQ(focusShiftListCopy[i].FocusShiftActivationsCounterMoving, 0u);
   }

   // Cleanup
   ::remove(FOCUS_SHIFT_LIST_FILE);
}

TEST_F(Test_FocusShiftLogger, copyClearedList)
{
   FocusShiftListProperties wList[NUMBER_OF_DRIVER_MEMORIES];

   FocusShiftLogger_CopyList((uint8*)wList);
```

```
   for (uint8 i = 0; i < NUMBER_OF_DRIVER_MEMORIES; i++)
   {
      EXPECT_EQ(wList[i].FocusICTotalTime, 0u);
      EXPECT_EQ(wList[i].FocusSIDTotalTime, 0u);
      EXPECT_EQ(wList[i].FocusICCounterButtonPresses, 0u);
      EXPECT_EQ(wList[i].FocusSIDCounterButtonPresses, 0u);
      EXPECT_EQ(wList[i].FocusShiftActivationsCounterStandstill, 0u);
      EXPECT_EQ(wList[i].FocusShiftActivationsCounterMoving, 0u);
   }
}

TEST_F(Test_FocusShiftLogger, FocusShiftToSID)
{
   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_SID);

   mockVehicleMode = VEHICLE_MODE_RUNNING;
   FocusShiftLogger_MainFunction();

   EXPECT_EQ(mActiveDisplayFocusItemStat, DISPLAY_FOCUS_SID);
   EXPECT_EQ(mRequestedDisplayFocusItemStat, DISPLAY_FOCUS_SID);
}

TEST_F(Test_FocusShiftLogger, FocusShiftToIC)
{
   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_IC);

   mockVehicleMode = VEHICLE_MODE_PRERUNNING;
   FocusShiftLogger_MainFunction();

   EXPECT_EQ(mActiveDisplayFocusItemStat, DISPLAY_FOCUS_IC);
   EXPECT_EQ(mRequestedDisplayFocusItemStat, DISPLAY_FOCUS_IC);
}

TEST_F(Test_FocusShiftLogger, NewDriverRestartsLogging)
{
   FocusShiftListProperties wList[NUMBER_OF_DRIVER_MEMORIES];
   FocusShiftLogger_CopyList((uint8*)wList);

   mockVehicleMode = VEHICLE_MODE_CRANKING;
   mockDriverMemory = DRIVER_MEMORY_2;
   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_SID);

   UTCTimestamp_CallbackFunc(utcTimestamp1);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);

   EXPECT_EQ(wList[DRIVER_MEMORY_2].FocusSIDTotalTime, 0u);

   mockDriverMemory = DRIVER_MEMORY_7;

   UTCTimestamp_CallbackFunc(utcTimestamp2);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);

   EXPECT_EQ(wList[DRIVER_MEMORY_2].FocusSIDTotalTime, 5453u);
}

TEST_F(Test_FocusShiftLogger, TriggerReset)
{
   mockVehicleMode = VEHICLE_MODE_PRERUNNING;
   mockDriverMemory = DRIVER_MEMORY_6;
   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_IC);
   mockResetFlagFocusShift = true;

   EXPECT_EQ(callCount_DriverMemory_DeactivateResetFlagFocusShift, 0);

   FocusShiftLogger_MainFunction();

   EXPECT_EQ(callCount_DriverMemory_DeactivateResetFlagFocusShift, 1);
   EXPECT_EQ(mockResetFlagFocusShift, false);
}

TEST_F(Test_FocusShiftLogger, TriggerResetAllDriverMemories)
{
   FocusShiftListProperties wList[NUMBER_OF_DRIVER_MEMORIES];

   mockVehicleMode = VEHICLE_MODE_PRERUNNING;
   mockDriverMemory = DRIVER_MEMORY_DEFAULT;
```

```
   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_IC);
   mockResetFlagFocusShift = true;
   mFocusShiftList.setFocusShiftListProperties(DRIVER_MEMORY_5, &item1);
   mFocusShiftList.setFocusShiftListProperties(DRIVER_MEMORY_8, &item1);

   FocusShiftLogger_CopyList((uint8*)wList);
   EXPECT_EQ(callCount_DriverMemory_DeactivateResetFlagFocusShift, 0);
   EXPECT_EQ(wList[DRIVER_MEMORY_5].FocusICTotalTime, 111u);
   EXPECT_EQ(wList[DRIVER_MEMORY_8].FocusShiftActivationsCounterMoving, 666u);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);
   EXPECT_EQ(callCount_DriverMemory_DeactivateResetFlagFocusShift, 1);
   EXPECT_EQ(mockResetFlagFocusShift, false);
   EXPECT_EQ(wList[DRIVER_MEMORY_5].FocusICTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_5].FocusSIDTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_5].FocusICCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_5].FocusSIDCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_5].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_5].FocusShiftActivationsCounterMoving, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_5].FocusICTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_8].FocusSIDTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_8].FocusICCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_8].FocusSIDCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_8].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_8].FocusShiftActivationsCounterMoving, 0u);
}

TEST_F(Test_FocusShiftLogger, GetDisplayFocus)
{
   mockVehicleMode = VEHICLE_MODE_CRANKING;
   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_SID);

   FocusShiftLogger_MainFunction();

   EXPECT_EQ(FocusShiftLogger_GetDisplayFocus(), DISPLAY_FOCUS_SID);
}

TEST_F(Test_FocusShiftLogger, VehicleModeLoweredStopLogging)
{
   FocusShiftListProperties wList[NUMBER_OF_DRIVER_MEMORIES];
   FocusShiftLogger_CopyList((uint8*)wList);

   mockVehicleMode = VEHICLE_MODE_PRERUNNING;
   mockDriverMemory = DRIVER_MEMORY_8;
   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_IC);

   UTCTimestamp_CallbackFunc(utcTimestamp1);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);

   EXPECT_EQ(mActiveDisplayFocusItemStat, DISPLAY_FOCUS_IC);
   EXPECT_EQ(mRequestedDisplayFocusItemStat, DISPLAY_FOCUS_IC);
   EXPECT_EQ(wList[DRIVER_MEMORY_8].FocusICTotalTime, 0u);

   UTCTimestamp_CallbackFunc(utcTimestamp2);
   mockVehicleMode = VEHICLE_MODE_ACCESSORY;

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);

   EXPECT_EQ(mActiveDisplayFocusItemStat, DISPLAY_FOCUS_NOT_AVAILABLE);
   EXPECT_EQ(mRequestedDisplayFocusItemStat, DISPLAY_FOCUS_NOT_AVAILABLE);
   EXPECT_EQ(wList[DRIVER_MEMORY_8].FocusICTotalTime, 5453u);
}

TEST_F(Test_FocusShiftLogger, StartExecuteFirstInPreRunning)
{
   mVehicleMode_prev = VEHICLE_MODE_LIVING;
   mockVehicleMode = VEHICLE_MODE_LIVING;

   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_IC);

   FocusShiftLogger_MainFunction();

   EXPECT_EQ(mActiveDisplayFocusItemStat, DISPLAY_FOCUS_NOT_AVAILABLE);
   EXPECT_EQ(mRequestedDisplayFocusItemStat, DISPLAY_FOCUS_IC);

   mockVehicleMode = VEHICLE_MODE_ACCESSORY;
```

```
   FocusShiftLogger_MainFunction();

   EXPECT_EQ(mActiveDisplayFocusItemStat, DISPLAY_FOCUS_NOT_AVAILABLE);
   EXPECT_EQ(mRequestedDisplayFocusItemStat, DISPLAY_FOCUS_IC);

   mockVehicleMode = VEHICLE_MODE_PRERUNNING;

   FocusShiftLogger_MainFunction();

   EXPECT_EQ(mActiveDisplayFocusItemStat, DISPLAY_FOCUS_IC);
   EXPECT_EQ(mRequestedDisplayFocusItemStat, DISPLAY_FOCUS_IC);
}

TEST_F(Test_FocusShiftLogger, CountAndAddTimeFocusIC)
{
   FocusShiftListProperties wList[NUMBER_OF_DRIVER_MEMORIES];
   FocusShiftLogger_CopyList((uint8*)wList);

   UTCTimestamp_CallbackFunc(utcTimestamp1);
   mockVehicleMode = VEHICLE_MODE_PRERUNNING;
   mockDriverMemory = DRIVER_MEMORY_9;
   PS_VehicleMoving_CallbackFunc(VEHICLE_MOVING_STATUS_STANDSTILL);

   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_IC);

   FocusShiftLogger_MainFunction();

   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterMoving, 0u);

   UTCTimestamp_CallbackFunc(utcTimestamp2);

   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_SID);
   FocusShiftActivationTriggerLog_CallbackFunc(MANUAL_FOCUS_SHIFT_TO_SID);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);

   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICTotalTime, 5453u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterStandstill, 1u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterMoving, 0u);
}

TEST_F(Test_FocusShiftLogger, CountAndAddTimeFocusSID)
{
   FocusShiftListProperties wList[NUMBER_OF_DRIVER_MEMORIES];

   UTCTimestamp_CallbackFunc(utcTimestamp1);
   mockVehicleMode = VEHICLE_MODE_PRERUNNING;
   mockDriverMemory = DRIVER_MEMORY_9;
   PS_VehicleMoving_CallbackFunc(VEHICLE_MOVING_STATUS_MOVING);

   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_SID);
   FocusShiftActivationTriggerLog_CallbackFunc(MANUAL_FOCUS_SHIFT_TO_SID);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);

   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterMoving, 1u);

   UTCTimestamp_CallbackFunc(utcTimestamp2);

   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_IC);
   FocusShiftActivationTriggerLog_CallbackFunc(MANUAL_FOCUS_SHIFT_TO_IC);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);
```

```
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDTotalTime, 5453u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterMoving, 1u);
}

TEST_F(Test_FocusShiftLogger, InvalidUTCTimeDoNotAddTime)
{
   FocusShiftListProperties wList[NUMBER_OF_DRIVER_MEMORIES];

   UTCTimestamp_CallbackFunc(utcTimestamp1);
   mockVehicleMode = VEHICLE_MODE_RUNNING;
   mockDriverMemory = DRIVER_MEMORY_9;
   PS_VehicleMoving_CallbackFunc(VEHICLE_MOVING_STATUS_MOVING);

   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_SID);
   FocusShiftActivationTriggerLog_CallbackFunc(MANUAL_FOCUS_SHIFT_TO_SID);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);

   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterMoving, 1u);

   UTCTimestamp_CallbackFunc(utcTimestampInvalid);

   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_IC);
   FocusShiftActivationTriggerLog_CallbackFunc(MANUAL_FOCUS_SHIFT_TO_IC);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);

   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterMoving, 1u);
}

TEST_F(Test_FocusShiftLogger, InvalidDisplayFocusStopsOngoingLogging)
{
   FocusShiftListProperties wList[NUMBER_OF_DRIVER_MEMORIES];

   UTCTimestamp_CallbackFunc(utcTimestamp1);
   mockVehicleMode = VEHICLE_MODE_RUNNING;
   mockDriverMemory = DRIVER_MEMORY_9;
   PS_VehicleMoving_CallbackFunc(VEHICLE_MOVING_STATUS_MOVING);
   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_IC);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICTotalTime, 0u);

   UTCTimestamp_CallbackFunc(utcTimestamp2);
   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_ERROR);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICTotalTime, 5453u);
}

TEST_F(Test_FocusShiftLogger, CountButtonPressesFocusSID)
{
   FocusShiftListProperties wList[NUMBER_OF_DRIVER_MEMORIES];

   mockVehicleMode = VEHICLE_MODE_PRERUNNING;
   mockDriverMemory = DRIVER_MEMORY_9;
   PS_VehicleMoving_CallbackFunc(VEHICLE_MOVING_STATUS_MOVING);

   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_SID);
   FocusShiftActivationTriggerLog_CallbackFunc(MANUAL_FOCUS_SHIFT_TO_SID);
```

```
   SW_Home_ButtonsStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Menu_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Down_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Enter_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Esc_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Left_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Right_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Up_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);

   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterMoving, 1u);

   SW_Home_ButtonsStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);
   SW_Down_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);

   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDCounterButtonPresses, 2u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterMoving, 1u);

   SW_Left_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);
   SW_Right_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);
   SW_Up_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);

   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDCounterButtonPresses, 5u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterMoving, 1u);

   SW_Home_ButtonsStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NOT_AVAILABLE);
   SW_Menu_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);
   SW_Enter_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);
   SW_Esc_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);
   SW_Left_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_ERROR);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);

   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusICCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusSIDCounterButtonPresses, 8u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_9].FocusShiftActivationsCounterMoving, 1u);
}

TEST_F(Test_FocusShiftLogger, CountButtonPressesFocusIC)
{
   FocusShiftListProperties wList[NUMBER_OF_DRIVER_MEMORIES];

   mockVehicleMode = VEHICLE_MODE_PRERUNNING;
   mockDriverMemory = DRIVER_MEMORY_1;
   PS_VehicleMoving_CallbackFunc(VEHICLE_MOVING_STATUS_MOVING);

   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_IC);
   FocusShiftActivationTriggerLog_CallbackFunc(MANUAL_FOCUS_SHIFT_TO_IC);

   SW_Home_ButtonsStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Menu_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Down_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Enter_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
```

```
   SW_Esc_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Left_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Right_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Up_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);

   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusICTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusSIDTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusICCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusSIDCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusShiftActivationsCounterMoving, 0u);

   SW_Down_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);
   SW_Enter_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);
   SW_Esc_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);

   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusICTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusSIDTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusICCounterButtonPresses, 3u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusSIDCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusShiftActivationsCounterMoving, 0u);

   SW_Home_ButtonsStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);
   SW_Menu_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);
   SW_Left_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);
   SW_Right_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);

   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusICTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusSIDTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusICCounterButtonPresses, 7u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusSIDCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusShiftActivationsCounterMoving, 0u);

   SW_Down_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_ERROR);
   SW_Enter_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NOT_AVAILABLE);
   SW_Up_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);

   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusICTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusSIDTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusICCounterButtonPresses, 8u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusSIDCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_1].FocusShiftActivationsCounterMoving, 0u);
}

TEST_F(Test_FocusShiftLogger, SameButtonPushedTwice)
{
   FocusShiftListProperties wList[NUMBER_OF_DRIVER_MEMORIES];

   mockVehicleMode = VEHICLE_MODE_RUNNING;
   mockDriverMemory = DRIVER_MEMORY_4;
   PS_VehicleMoving_CallbackFunc(VEHICLE_MOVING_STATUS_STANDSTILL);

   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_IC);
   FocusShiftActivationTriggerLog_CallbackFunc(MANUAL_FOCUS_SHIFT_TO_IC);

   SW_Home_ButtonsStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Menu_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Down_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Enter_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Esc_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Left_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Right_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
   SW_Up_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
```

```
    FocusShiftLogger_MainFunction();

    FocusShiftLogger_CopyList((uint8*)wList);

    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusICTotalTime, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusSIDTotalTime, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusICCounterButtonPresses, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusSIDCounterButtonPresses, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusShiftActivationsCounterStandstill, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusShiftActivationsCounterMoving, 0u);

    SW_Enter_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);

    FocusShiftLogger_MainFunction();

    FocusShiftLogger_CopyList((uint8*)wList);

    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusICTotalTime, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusSIDTotalTime, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusICCounterButtonPresses, 1u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusSIDCounterButtonPresses, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusShiftActivationsCounterStandstill, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusShiftActivationsCounterMoving, 0u);

    SW_Enter_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);

    FocusShiftLogger_MainFunction();

    FocusShiftLogger_CopyList((uint8*)wList);

    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusICTotalTime, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusSIDTotalTime, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusICCounterButtonPresses, 1u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusSIDCounterButtonPresses, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusShiftActivationsCounterStandstill, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusShiftActivationsCounterMoving, 0u);

    SW_Enter_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);

    FocusShiftLogger_MainFunction();

    FocusShiftLogger_CopyList((uint8*)wList);

    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusICTotalTime, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusSIDTotalTime, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusICCounterButtonPresses, 2u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusSIDCounterButtonPresses, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusShiftActivationsCounterStandstill, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_4].FocusShiftActivationsCounterMoving, 0u);
}

TEST_F(Test_FocusShiftLogger, OnlyCountManualFocusShiftActivationsToSID)
{
    FocusShiftListProperties wList[NUMBER_OF_DRIVER_MEMORIES];

    FocusShiftLogger_CopyList((uint8*)wList);
    EXPECT_EQ(wList[DRIVER_MEMORY_5].FocusShiftActivationsCounterStandstill, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_5].FocusShiftActivationsCounterMoving, 0u);

    mockVehicleMode = VEHICLE_MODE_RUNNING;
    mockDriverMemory = DRIVER_MEMORY_5;
    PS_VehicleMoving_CallbackFunc(VEHICLE_MOVING_STATUS_STANDSTILL);

    DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_IC);
    FocusShiftActivationTriggerLog_CallbackFunc(MANUAL_FOCUS_SHIFT_TO_IC);

    FocusShiftLogger_MainFunction();

    FocusShiftLogger_CopyList((uint8*)wList);
    EXPECT_EQ(wList[DRIVER_MEMORY_5].FocusShiftActivationsCounterStandstill, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_5].FocusShiftActivationsCounterMoving, 0u);

    DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_SID);
    FocusShiftActivationTriggerLog_CallbackFunc(AUTO_FOCUS_SHIFT_TO_SID);

    FocusShiftLogger_MainFunction();

    FocusShiftLogger_CopyList((uint8*)wList);
    EXPECT_EQ(wList[DRIVER_MEMORY_5].FocusShiftActivationsCounterStandstill, 0u);
    EXPECT_EQ(wList[DRIVER_MEMORY_5].FocusShiftActivationsCounterMoving, 0u);

    DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_IC);
    FocusShiftActivationTriggerLog_CallbackFunc(AUTO_FOCUS_SHIFT_TO_IC);
```

```
   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);
   EXPECT_EQ(wList[DRIVER_MEMORY_5].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_5].FocusShiftActivationsCounterMoving, 0u);

   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_SID);
   FocusShiftActivationTriggerLog_CallbackFunc(MANUAL_FOCUS_SHIFT_TO_SID);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);
   EXPECT_EQ(wList[DRIVER_MEMORY_5].FocusShiftActivationsCounterStandstill, 1u);
   EXPECT_EQ(wList[DRIVER_MEMORY_5].FocusShiftActivationsCounterMoving, 0u);
}

TEST_F(Test_FocusShiftLogger, FocusShiftActivationTriggerLogDoesNotHaveToTimeTransitionToBeCounted)
{
   FocusShiftListProperties wList[NUMBER_OF_DRIVER_MEMORIES];

   FocusShiftLogger_CopyList((uint8*)wList);
   EXPECT_EQ(wList[DRIVER_MEMORY_10].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_10].FocusShiftActivationsCounterMoving, 0u);

   mockVehicleMode = VEHICLE_MODE_RUNNING;
   mockDriverMemory = DRIVER_MEMORY_10;
   PS_VehicleMoving_CallbackFunc(VEHICLE_MOVING_STATUS_STANDSTILL);

   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_SID);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);
   EXPECT_EQ(wList[DRIVER_MEMORY_10].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_10].FocusShiftActivationsCounterMoving, 0u);

   FocusShiftActivationTriggerLog_CallbackFunc(MANUAL_FOCUS_SHIFT_TO_SID);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);
   EXPECT_EQ(wList[DRIVER_MEMORY_10].FocusShiftActivationsCounterStandstill, 1u);
   EXPECT_EQ(wList[DRIVER_MEMORY_10].FocusShiftActivationsCounterMoving, 0u);
}

TEST_F(Test_FocusShiftLogger, LockAndSetList)
{
   FocusShiftListProperties wList[NUMBER_OF_DRIVER_MEMORIES];
   FocusShiftLogger_CopyList((uint8*)wList);
   FocusShiftListProperties focusShiftListCopy[NUMBER_OF_DRIVER_MEMORIES];
   memset(focusShiftListCopy, 0xaa, sizeof(focusShiftListCopy));

   FocusShiftLogger_LockAndSetList((uint8*) focusShiftListCopy);

   EXPECT_EQ(EXPECTED_LIST_SIZE, mFocusShiftList.size());

   //FocusShiftLogger_CopyList((uint8*)wList);
   EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusICTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusSIDTotalTime, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusICCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusSIDCounterButtonPresses, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusShiftActivationsCounterStandstill, 0u);
   EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusShiftActivationsCounterMoving, 0u);
   EXPECT_EQ(focusShiftListCopy[DRIVER_MEMORY_6].FocusICTotalTime, 2863311530u);
   EXPECT_EQ(focusShiftListCopy[DRIVER_MEMORY_6].FocusSIDTotalTime, 2863311530u);
   EXPECT_EQ(focusShiftListCopy[DRIVER_MEMORY_6].FocusICCounterButtonPresses, 2863311530u);
   EXPECT_EQ(focusShiftListCopy[DRIVER_MEMORY_6].FocusSIDCounterButtonPresses, 2863311530u);
   EXPECT_EQ(focusShiftListCopy[DRIVER_MEMORY_6].FocusShiftActivationsCounterStandstill, 43690u);
   EXPECT_EQ(focusShiftListCopy[DRIVER_MEMORY_6].FocusShiftActivationsCounterMoving, 43690u);

   mockVehicleMode = VEHICLE_MODE_PRERUNNING;
   mockDriverMemory = DRIVER_MEMORY_6;
   PS_VehicleMoving_CallbackFunc(VEHICLE_MOVING_STATUS_STANDSTILL);

   DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_IC);
   FocusShiftActivationTriggerLog_CallbackFunc(MANUAL_FOCUS_SHIFT_TO_IC);
   UTCTimestamp_CallbackFunc(utcTimestamp1);

   FocusShiftLogger_MainFunction();

   FocusShiftLogger_CopyList((uint8*)wList);
   EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusICTotalTime, 2863311530u);
```

```
  EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusSIDTotalTime, 2863311530u);
  EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusICCounterButtonPresses, 2863311530u);
  EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusSIDCounterButtonPresses, 2863311530u);
  EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusShiftActivationsCounterStandstill, 43690u);
  EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusShiftActivationsCounterMoving, 43690u);

  SW_Home_ButtonsStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);
  SW_Menu_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
  SW_Down_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
  SW_Enter_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
  SW_Esc_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
  SW_Left_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
  SW_Right_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);
  SW_Up_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_NEUTRAL);

  UTCTimestamp_CallbackFunc(utcTimestamp2);
  DisplayFocusItemStat_CallbackFunc(DISPLAY_FOCUS_SID);
  FocusShiftActivationTriggerLog_CallbackFunc(MANUAL_FOCUS_SHIFT_TO_SID);

  FocusShiftLogger_MainFunction();

  // New Focus Shift activation, logged time with Focus IC and putton bush shall not be counted
  FocusShiftLogger_CopyList((uint8*)wList);
  EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusICTotalTime, 2863311530u);
  EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusSIDTotalTime, 2863311530u);
  EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusICCounterButtonPresses, 2863311530u);
  EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusSIDCounterButtonPresses, 2863311530u);
  EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusShiftActivationsCounterStandstill, 43690u);
  EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusShiftActivationsCounterMoving, 43690u);

  // Logging updated again after unlock
  FocusShiftLogger_Unlock();

  SW_Enter_ButtonStatus_6_CallbackFunc(PUSH_BUTTON_STATUS_PUSHED);

  FocusShiftLogger_MainFunction();

  FocusShiftLogger_CopyList((uint8*)wList);
  EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusICTotalTime, 2863311530u);
  EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusSIDTotalTime, 2863311530u);
  EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusICCounterButtonPresses, 2863311530u);
  EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusSIDCounterButtonPresses, 2863311531u);
  EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusShiftActivationsCounterStandstill, 43690u);
  EXPECT_EQ(wList[DRIVER_MEMORY_6].FocusShiftActivationsCounterMoving, 43690u);
}
```