ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE

DEPARTMENT OF

Electrical, Electronic, and Information Engineering "Guglielmo Marconi" - DEI

MASTER OF SCIENCE IN

Advanced Automotive Electronic Engineering

MASTER'S THESIS

in

Test, Diagnosis and Reliability – M

Strategies for Automated Validation of Infotainment for Electric Vehicles

CANDIDATE:

Jaber Nikpouri

SUPERVISOR:

Prof. Cecilia Metra

CO-SUPERVISORS:

Prof. Martin Eugenio Omaña

Gandolfo Mario Librizzi

Paola Conti

Academic Year 2019/2020

Session IV

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I consider myself as a lucky individual due to having the opportunity to spend my internship in Maserati and to be perceived as a big milestone in my career life. Thanks to Maserati, I gained invaluable experience and knowledge in the best possible way in an exciting high-tech atmosphere.

I express my deepest sense of gratitude to Gandolfo Mario Librizzi and Paola Conti as they supported and accompanied me to perform well, and made me feel confident in my abilities although they were extremely busy with their tasks.

I would like to thank my supervisors Professor Cecilia Metra and Professor Martin Eugenio Omana for continuously providing encouragement and undeniable support. It was my honour to be inspired to think outside the box from multiple perspectives in my academic life.

Moreover, I would also like to express my deepest and sincere gratitude to my mother for her unparalleled love, who supported me in every moment of my life and encouraged me to go through my goals. My journey could not be possible without her unlimited support and kind.

Sincerely,

Jaber Nikpouri

Chapter One

1. Introduction

Providing more smartness and comfort to the driver and passengers, vehicle manufacturers pay an exclusive attention on developing an advanced infotainment system and a responsive interface to satisfy the customer. This satisfaction brings into having a precise and error-free system without malfunction. There are many units communicating with each other and make it possible to have different features in infotainment systems of the new vehicles.

The automotive connectivity technology makes it possible for the research and design team to implement individual features using proper communication technology. Nowadays, there are some communication protocols for connectivity purposes between Electronic Control Units (ECUs) like Controller Area Network (CAN), Flex-Ray, Local Interconnect Network (LIN), Media Oriented System Transport (MOST), Ethernet, and so on. Therefore, the research and design teams must take into consideration different requirements to design the network topology of the carline.

The provided features in all subsystems need a complete testing approach to validate if the system works fine in all conditions. To enhance the accuracy and test coverage, manufacturers are going through an automated procedure to lower the faults and increase the efficiency.

Considering the intra-vehicle communication protocols and the functioning time while having an increased data load and security threats, Controller Area

Network-Flexible Data-Rate protocol is becoming common for crucial functionalities according to its benefits. Having all characteristics and advantages of CAN network in parallel, CAN FD network provides larger message packets containing more data and higher data-rate. Thus, it is possible to use the CAN FD network when we prefer the CAN network to have low cost and secure solutions but higher speed and larger data packets.

The industry demands an effort, which puts all the noted points together and provides an automated solution to validate the infotainment system. The automated solution is provided not only to increase the fault coverage percentage to cover as many errors as possible to check, but also to prevent human mistakes and repeating the test sequence with the same situation. Indeed, Stellantis group is developing an internal test tool to help the validation team perform efficiently; therefore, the scope of the project is to make the tool able to test CAN FD network used in the infotainment system of the future products.

Chapter Two

2. Conceptual Framework

Nowadays, the automotive industry is going through electric cars mainly because of energy consumption and pollution. It is undeniable that using electric energy instead of fuels reduces emissions and enhances world health by prohibiting ecological damage. Reacting faster and being responsive while having perfect torque convinces the automotive manufacturers to pay more attention to design and develop electric vehicles. Indeed, the moving parts in a battery electric vehicle (BEV) are less than a diesel vehicle and bring into having less need of servicing, cheaper exhaust systems, injection systems, ventilation system and some parts that are not needed in an electric vehicle.

Moreover, according to the technical progresses, the automated systems are being inclusive to increase the accuracy, speed and comfort. Although each manual and automated procedures have particular pros and cons, the industry is going toward automation and machine-based activities.

As the levels of autonomy in cars are increasing, an advanced interface between the passengers and the vehicle demands high-tech devices available inside the car to provide the driver and passengers a comfortable and reliable travel experience. This is why the vehicles are going toward smartness and connected to all desired interfaces like cell phones, GPS, camera, video streaming, and so on. An advanced set of devices are needed to support all these aspects and make it easy for the passengers to take the advantage of this connectivity. Therefore, the

manufacturers try to design and develop the infotainment system of the vehicles to satisfy these desires and demands.

2.1. Infotainment systems

Automotive infotainment includes the information transmitted and received among the vehicle and passenger involving visual, touch and acoustic senses. This information could be incorporated in the car and used internally, provided by external sources like smartphones and used in the car, and put into an IOT system to be connected by the cloud.

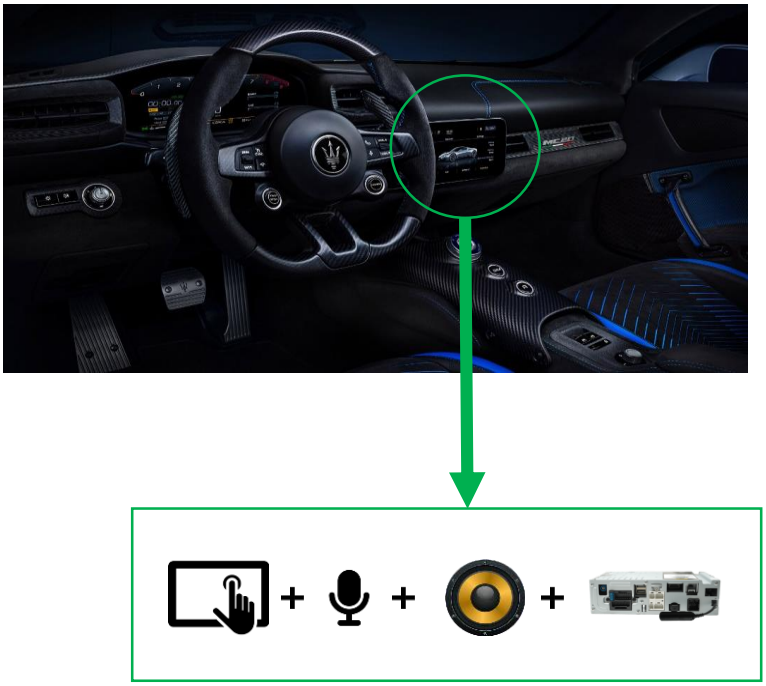


Figure 1 – Infotainment system features [1]

Although the first radio communication was established in 1901, its prototype was applied 17 years later and this technology was first used in 1926. Later, it was developed by other companies for 40 years having radios with FM, solid state

amplifiers and tape. The first traffic messages in radio bands were transmitted in 1975 and a year later, the first electronic instrument cluster was put inside the vehicle. A few years later, the first navigation system, the first car phone and first integration of CD technology and the vehicle radio were applied. The vehicles were equipped with DAB and digital TV 15 years later in around 2000. The first Bluetooth technology and radios with HDD were used in infotainment system in 2007 and then was equipped to DMB/DMB+ technology five years later. Apple Car-play and Android Auto entered the industry in 2014 to enhance the connectivity and intelligence of the car, and double-triple front displays appeared a few years later to increase the user interface and comfort of the passengers.

The infotainment system contains microphone, different types of antennas (Wi-Fi, Bluetooth, FM/AM/DAB/SDARS, and GNSS), speaker, display and silver-box. These commands are connected to the system also using Steering wheel buttons to allow the user to drive without removing hands from the steering wheel and perform the VR.

The software components realize all the features provided by the Infotainment and multimedia system. They are divided in some branches,

- Entertainment services like Radio, USB memory, etc.
- Information services like navigation technology
- Telecommunication services like SMS, voice calls, Remote controls, etc.
- Vehicle functions like adjustment of mirrors and seats, climate control, etc.
- Safety and SOS services like emergency calls and car tracking
- System administration services like software download, channel and resource management, etc.

The infotainment system is assumed as a distributed system providing some benefits to the driver and the passengers via the interaction with many modules.

The customer expects new applications and services in the vehicles that pushes the automotive industry to pay much more attention in this field to gain user satisfaction. Moreover, as new technology is involved, many electrical devices has to cooperate which makes the infotainment part more complex. Usually, interruption by another feature or device, simultaneous functions and consistency might raise some problems in service interaction. Therefore, some new testing methods should be performed as well as functional testing, regression testing and robustness testing to support all provided options like GPS data, different inputs/outputs, ADAS features, and so on. These tests have to consider the simultaneous commands and functionalities to analyse the timing and accurate tasks. The operation of infotainment systems depends on different embedded operating systems capabilities. Therefore, it is crucial to validate all features to ensure all tasks are intact. Moreover, timing and testing coverage are two important challenges that forces all development lines to pay more attention to use efficient testing procedures and enhance the quality.

2.2. Electrical Architecture

The core of the intelligence in the vehicle depends on the electrical architecture. Nowadays, cars contain many electronic components; therefore, the electrical architecture is a key factor for the development of a smart car.

An electrical architecture includes standards that define electrical interfaces, logic distribution and communication protocols that are assigned to hardware and software aspects of the controller. The communication network within the vehicle containing information about the ECU modes and bus types are defined in the network topology. Each car model has its particular network topology describing completely the connecting networks of all ECUs collaborating inside the vehicle.

The methodology used to communicate vehicle specific content to the ECUs are managed in vehicle configuration.

Considering the structure and architecture, the radio software is a custom software, which includes some specific dynamic link library (DLL), and it is necessary to enable communication with projection systems. Therefore, the radio device gets some information from sensors, microphone, buttons, knob, GPS, etc. and it will provide the results using display and speakers according to its configuration and program.

In the automotive industry, the embedded system is implemented on the board of the car used to communicate with the vehicle controls and customer service, and stay connected when a problem is raised. The provided facilities include wireless tracking, diagnostics, and intra-vehicle communication like event-driven information. It is easier to manage fast, reliable and efficient connectivity and processing to support intra-vehicle communication such as the communication among vehicles and cloud, infrastructure and other vehicles using this unit. Moreover, remote door services, navigation and traffic assistance, infotainment features, fleet management and diagnostics are the available services in the automotive industry.

Many cars produced recently provide telematics features using advanced built-in devices. With the recent connectivity technology, these units easily support multimedia services and connection of smartphones to have a single optimal inter-vehicle connection.

Telematics Control Unit is mostly implemented using Controller Area Network, which performs as a bridge among the relevant ECU collection in the vehicle in order to manage data transmission and provide them in a user-friendly way to have a comprehensive control on different features.

In this project, a radio unit is used to transmit and receive data via classic CAN 2.0 and CAN FD protocols. There are some channels allocated with respect to the architecture and protocols.

2.3. Controller Area Network

First microprocessors were used in vehicles in the 1980s. The implemented communication was via point-to-point connections and little real data were transmitted. After some years, a standard for character transmission called ISO 9141 was born and later, the data busses for intra-vehicle communication were introduced and standardized as Controller Area Network (CAN) or ISO 11898 developed by Bosch.

The main reasons of using bus systems are grouped in three topics including lower cost, higher modularity and shorter development cycles. Nowadays, according to the expansion of the modules used in the vehicles, the weight and volume of the wiring and processing system is focused. Moreover, customizability of the vehicles and cooperation with OEMs, reusability of components and error management using test plans using standard protocols are the notable features to convince the industry to take the advantage of bus systems.

CAN bus can be used for driveline (engine and transmission control), active and passive safety (Electronic stability program and airbags), comfort (lighting and automation) and infotainment (navigation system and multimedia).

CAN bus enables On Board Communication including complex control and monitoring tasks, and simplification of wiring and multimedia bus systems. Some well-known sensors connected by CAN bus are Adaptive cruise control, Electronic brake system, sensor cluster, door control unit, seat control, closing velocity sensors, airbag control unit, and so on.

On the other hand, Off Board Communication is also enabled by CAN bus which consists of diagnosis to read out plenty of errors, flashing which means initial installation of firmware on ECUS or adaptation, and debugging to detailed diagnosis of internal status during development.

Low cost, centralized diagnosis and configuration, robustness toward failure of subsystems or interference, efficiency due to the priority management, and flexibility for modifications and additional nodes convinces the designers to use CAN bus in the topologies. Thus, it is mandatory that each ECU attached to the CAN bus follows the CAN interface.

CAN bus includes two wires, CAN High and CAN Low, called twisted pair; therefore, they give noise resistance and resiliency and the ability to survive the connection by CAN low if one of the wires brakes. The bus ends with the impedance of 120 ohms on both sides. The distributed functions being implemented easily permits different configurations without adaptation of hardware and software.

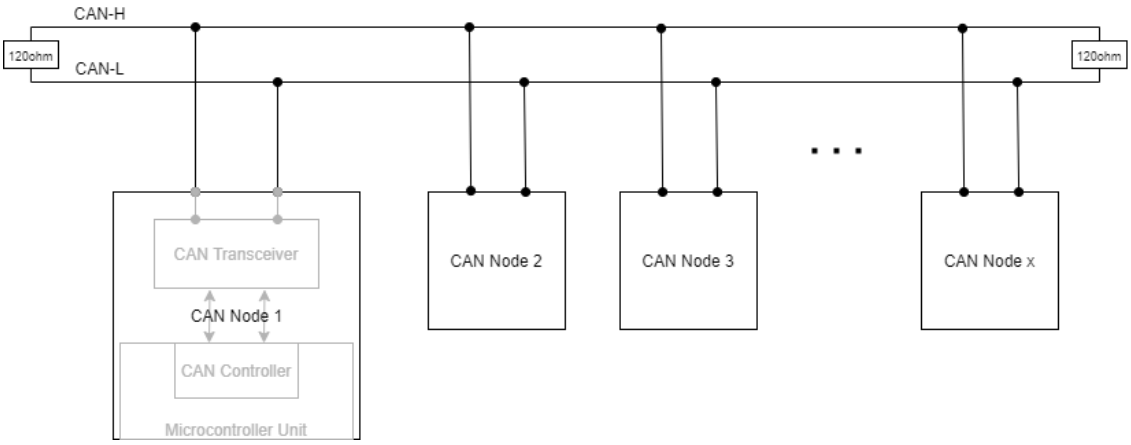


Figure 2 – CAN BUS Diagram (Hardware)

This protocol uses the receiver-selective form of addressing and each module includes an acceptance filtering technique to accept or refuse the messages

because each ECU reads the wire through a buffer and each one could write on the line through a transistor. The base state of a CAN bus is the logical value of one and the transistor would be in a non-conductive state. The ECU turns the transistor conductive that connects the bus to signal ground. Bus level becomes the logical value of zero independently from other ECUs; therefore, the bus will be in a dominant level that is defined as zero.

CAN is an event-driven bus system and waiting for a precise scheduled time slot is not needed, however, it gives rise to collisions as it is a dynamic scheduled system. When an ECU registers an event, it is authorized to access the bus immediately and send data, however, if another ECU is already transmitting data, a list of waiting ECUs is being updated according to the priorities to get the bus free to use. This list is a step to avoid collision because CAN protocol uses Carrier-sense multiple access with collision avoidance. Using Carrier Sense Multiple Access – Collision Avoidance (CSMA-CA) method, when an ECU wants to transmit, it checks if the bus is free and its value is recessive. Finding the bus busy and in dominant level, ECU must wait, and as it is free, it would be possible that multiple ECUs would start to transmit. In order to avoid the collision, the bitwise arbitration method is implemented. In this condition, all ECUs with a transmission request send their message identifiers. Bitwise from the most significant to least significant bit, considering the 0 value prevails over the 1 value on the bus and the lower ID wins. This rule determines if an ECU might continue sending or must stop and wait. Therefore, random, non-destructive and priority-controlled bus access ensures fast and fair bus access [2].

The calculation of Cyclic Redundancy Check (CRC) depends on the arithmetic of module 2 polynomial. It performs the same as polynomial remainder theorem in mathematics and follows like XOR operation in electronics. The generator function divides the message, and the final message is obtained by putting the message and the rest near together. On the other hand, the receiver receives the final message and divides it by the generator function that it already knows. If the

receiver obtains no rest, the transmission was successful; otherwise, it will detect the single bit errors. It is crucial to have a good generator function considering the fact that the rest would be zero if the error is multiple of the generator.

There are two main coding techniques to code bits, Non Return to Zero (NRZ) and Manchester (original variant). To reduce the electromagnetic Interface (EMI), this protocol provides three solutions like adding shielding to wires, using twisted pair wiring and using the coding technique with few rises and falls in signal edges. NRZ is less noisy; on the other hand, Manchester carries the clock with him on every single bit. Using NRZ coding, sending many identical bits does not leave signal edges, which could be used to compensate for clock drift, one of de-synchronization sources. A possible solution is inserting extra bits called stuff bits after a determined number of identical bits.

To start a communication on the bus, at least two nodes are needed because the transmitted message has to be acknowledged by the receiver through the ACK bit considering the fact that the transmitting controller would generate an error flag if the ACK is not properly detected. If a message is not acknowledged, an error is generated with each event until the controller reaches an error limit which is internally set by the protocol and the controller places itself in a bus-off state. This strategy is due to preventing a single node from blocking all the communication.

Considering the physical layer, to lessen the reflected waves happening by the impedance mismatch, the unshielded twisted pair is used ending with two 120 ohms resistors, however, the terminator resistors are not on a node to avoid losing termination if the node is removed. Because of the reflections, the signal in the unterminated cable is obtained by the superimpositions of reflected signals adding to the original one. The maximum bus length is a trade-off with the chosen signalling rate. Actually, the main transmission rate limitation is applied by the cable bandwidth limitations that degrade the signal transition time and introduce inter-symbol interference; however, the cable losses are important only for long distances. The total propagation delay is around five nanoseconds per meter for

twisted-pair cables, however, some reasons like skin effects, electrical and radiation losses, total system delay, proximity to other circuits, etc. should be considered.

When different transmitters send their data using the bus, they listen for the faithful transmission on a bit by bit logic until the dominant bit of another node overwrites the recessive bit which means a higher priority is present and the loser node has to wait and try again during the next opportunity. Therefore, all modules should assert their data within the same bit time before the sampling point; otherwise, data will be falsely received or damaged. This is the main reason to define a time constraint which unfortunately impacts the cabling distance.

The propagation delay consists of input/output delays of CAN controller, transmission and reception delays of the transceiver and cable propagation delay. Thus, it would be possible to find out the maximum allowable cable length of the bus considering the fact that also the bitwise arbitration scheme limits the maximum length. Each bit contains four time segments called synchronization, propagation, phase one and phase two segments. The synchronization happens in the first segment and it is used from recessive to dominant transitions on rising edges and the propagation segment is used for the compensation of the physical delay on the network. In order to manage the sampling point, two programmable segments are available to shorten or lengthen the timing frame of each bit due to re-synchronization and the sample point is between the phase segments. Each segment includes smaller parts called time quantum which could be calculated by dividing Baud-Rate Pre-scalar into the system clock [3].

The galvanic isolation protects the CAN controller and PC from high voltage incidents on the bus. For automotive environments, system robustness could be enhanced by isolating the logic interface to the transceiver. This isolation is between CAN controller and transceiver because the transceiver is the amplifying component that provides the voltage levels for the CAN bus. The signal propagation compensation is mandatory and it is notable that the sum of the

propagation delay time of the controllers, transceivers, bus line and galvanic isolators must be a small fraction of the time of a single bit.

In case of having more nodes like 30 normal nodes on a CAN bus, using transceivers with a high bus-input impedance is highly advised. This problem is due to having many transceiver source or sink current onto or from the bus to sink or source all leakage current on the bus and drive the normal signal voltage levels across the termination resistance. If the demanded current is big, the drive might be driven into the thermal shutdown or damaged. To prevent having the transceiver destroyed, the voltage difference among the reference grounds of the nodes must be held to a minimum which is the common-mode voltage across the system and the cumulative current demand by a lot of devices at a common-mode voltage extreme might be dangerous for the network security although many transceivers are designed in a way to be able to operate over an extended common-mode range. Therefore, to increase this security aspect, many higher layer protocols like Device-Net mention that power and ground wires be carried along with the signalling pair of wires. This is why some cable companies develop 4-wire bundled cables for this kind of application.

CAN protocol was developed to be used in the automotive world and car manufacturers take the advantage of two types including CAN B (ISO 11898-2) and CAN C (ISO 11898-3). The maximum transmission speed for CAN B is up to 125 Kbit per second and it is mostly used for vehicle body functions like seats, window, instrumentation and basic media features such as changing the language of the system. The bus termination in this type depends on each module, considering the fact that the sum will be more than 100 ohms. On the other hand, the maximum transmission speed for CAN C is up to 500 Kbit per second and the bus termination is equal to 120 ohms at the opposite sides providing 60 ohms in total. Therefore, having the higher speed, this type is mostly used for real-time functions like engine management and anti-lock brake operation. CAN FD (ISO

11898-3:2015) is another version of the high speed CAN (CAN C) in which many other advantages are included [4].

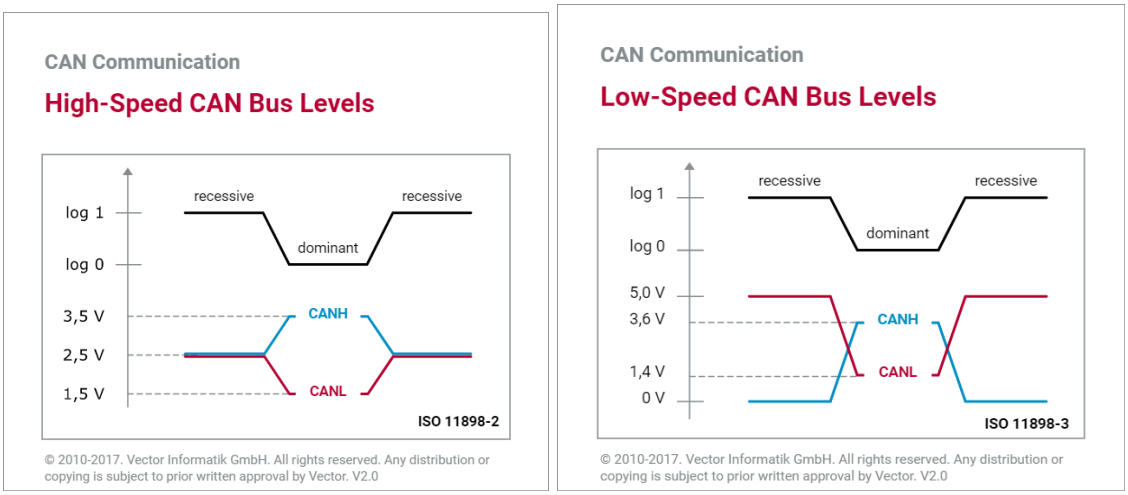


Figure 3 – CAN Bus voltage levels [4]

There are some limitations which need to be considered in classical CAN bus. First, the network speed is limited to 1Mbps (typically less than 500 Kbps). This limitation is forced by physical aspects of the harness because of the response mechanism done by ACK delay including the propagation delay introduced by the transceiver and the wire. Second, CAN messages have more than 50 percent of overhead with respect to other protocols like Ethernet UDP and Flex-Ray.

CAN FD provides shorter CAN frames and increased bitrate, which means both better data throughput due to having less relative overhead, and the simple software when sending large data objects. Additionally, CAN FD has a higher performance CRC algorithm to result in lowering the risk of undetected errors. It is an upgraded CAN protocol in which two invaluable features are considered. CAN FD can support dual bitrates in the message, therefore, we can have an increased bitrate in data part up to 5Mbps, however, the bitrate in arbitration phase remains the same as CAN. Moreover, CAN FD supports larger data length (up to 64Bytes per message). These upgrades could be done while the cost does not

change a lot due to the fact that existing CAN transceivers can be used up to 2 or 5 Mbps. The mentioned cost might include controller, transceiver, crystal, and node interconnection cost.

Considering the mentioned features, we could mix CAN and CAN FD ECUs under certain conditions. If all nodes are CAN FD capable, this mixture can be implemented, however, if some nodes are not CAN FD capable, the communication is done only with classic CAN messages or incapable nodes being switched off. Thus, when we transmit in classical CAN, we could receive both in classical CAN and CAN FD, but when the transmission is done by CAN FD, it is not possible to receive it by the classic CAN receiver.

These points are related to the physical layer, controller, and the system topics. The design has to start with the hardware part that conforms to physical layer and controller requirements. It is notable that the car industry is dealing with an increased EOL programming costs because of the increase in ECU memory requirements. Moreover, the new electronic applications need more ECUs resulting in the decrease in the available bandwidth on CAN bus networks that might lead to add another CAN network to the system. Using CAN FD, it would be easier to program EOL and free up the bandwidth, however, dealing with the new CAN FD system is challenging. CAN FD architectures will use dedicated CAN FD networks and mixed networks. In the first type, all CAN nodes are CAN FD capable, which will result in less effects on the physical layer transceivers and CAN FD usage all the time, however, it increases the cost. Therefore, it would be better to get involved in the second type and mix the network due to the fact that it is easier to migrate to this new technology and there would be no need to change the whole system. On the other hand, physical layer transceivers should support CAN FD filtering method on classical CAN nodes to determine no error frame will be created while CAN FD communication, so it increases the complexity and cost.

Comparing OSI models, the difference between classical CAN and CAN FD are in LLC and MAC sublayers of the data link layer, and PCS and PMA sub-layers of the physical layer. MDI of the physical layer is identical in both protocols. Deeply analysing, the difference in MAC is due to the increase in payload from 8 up to 64 data Bytes. In addition, encryption of CAN FD messages increases the security due to the higher data rate and payload. Moreover, the increase of data rate from 500Kbps up to 2Mbps for nominal operating conditions and up to 5Mbps for diagnostics and EOL programming is done in the PCS part which is counted as another difference between these two protocols.

Using CAN FD, we gain some positive features including avoiding split of data into several frames, avoiding split of networks, decreasing bus load of an existing bus, increasing the number of ECUs on the bus, and faster communication on long bus lines.

Comparing the frames, CAN and CAN FD have the same SOF which is a single dominant bit. They share the same addressing for both standard and extended formats of the messages, but CAN FD removes the RTR bit and RRS bit remains always in dominant level. In control field, CAN FD and CAN share IDE, RES, and DLC bits while CAN FD also adds FDF to distinguish between CAN and CAN FD which is the same as dominant bit called reserved bit or R0 in classical CAN, BRS to separate arbitration and data phase in CAN FD containing the switch of clock rate when BRS is recessive, and ESI to indicate the active or passive error. ESI bit is normally sent dominant, and if the CAN FD frame sender becomes error-passive, this bit will be sent recessive to indicate that the transmitter has a communication problem. In DLC, both of them use four bits and CAN FD is compatible with CAN at data lengths less than 7. Unlike CAN FD, three LSB are ignored by CAN is DLC is equal to 8, and when the length is more than 8, CAN FD uses 8, 12, 16, 20, 24, 32, 48, 64 while in classic CAN it remains 8. If DLC is zero, there is no data field. To have any number of bytes from 9 to 63 would need a 6-bit DLC, and the seventh one would be required to go to 64 bytes.

It was agreed to keep the 4 bit DLC and restrict the number of byte lengths in the CAN FD frames. CAN FD CRC calculation include preceding stuff bits while standard CAN does not use stuff bits in this calculation. The transmission of total number of bits is needed, therefore, it includes the number of dynamic stuff bits into the frame format. In this case, a parity bit is added (even parity), and the stuff bit count is grey-coded. As a result, being CAN or CAN FD and the length of DLC are two factors that makes the size of CRC different, so there would be 15 bits for CAN because all frames have a similar length, 17 bits for CAN FD having data bits less than or equal to 16 bytes, and 21 bits for CAN FD having data bits more than 16 bytes. CAN FD CRC delimiter is always transmitted as 1 bit, however, because of the phase shift between nodes, a transmitter accepts up to 2 bit-time. It should be considered that the data phase in CAN FD ends with the sample point of the first bit of the CRC delimiter. CAN FD nodes recognize up to two bit times as a valid ACK due to having one more bit time allowed to compensate for transceiver phase shift and bus propagation delay because of the switch from a high data phase clock to a low arbitration phase clock. A group of seven recessive bits delimit the frames which is called EOF, and it is the same in both protocols. In Figure 4, the message frames of both classic CAN and CAN FD are compared to have an overview.

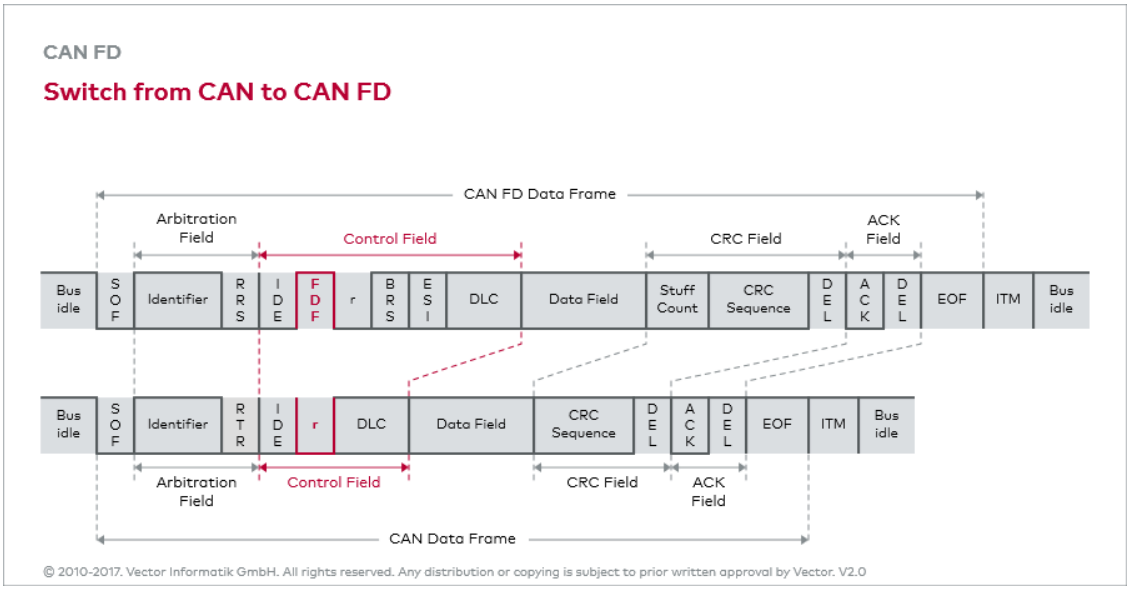


Figure 4 – Classic CAN frame vs. CAN FD frame [4]

Dynamic switching between CAN and CAN FD is allowed by the controller. They differ in having fixed or dual bitrate and being base format or the extended one. Error frame is the same and always sent with arbitration bitrate. Here, the controller switches automatically to arbitration bitrate. Also. Overload frame in CAN FD is identical to CAN and is always sent with arbitration bitrate. Remote frames are not defined in CAN FD format due to the fact that RTR bit is removed from CAN FD bit stream. The higher bitrate does not apply only to the data part of CAN FD, but to ESI, DLC, data, CRC, and CRC delimiter [5].

It is possible to use CAN FD controllers in the system in conjunction with classical CAN controllers, and CAN FD frames and CAN ones could be mixed or use only one of two types when all old classical CAN controllers will be replaced with CAN FD controllers. In case of CAN transceivers, typical bitrates of automotive transceivers for CAN FD are 2Mbps for functional messages and 5Mbps for reprogramming. Transceivers for CAN FD support of 2Mbps within current emission limits. CAN FD average bitrate converges because of the arbitration phase which becomes dominant at a certain baud rate for the data phase. It means that the overall frame length becomes not much smaller through a further increase

of the data phase bitrate. In case of performance analysis, excluding stuff bits, and considering maximum CAN frame with 111 bits and maximum CAN FD frame with 120/572 bits, and 500 Kbps for arbitration bitrate, the frame duration would be 222us in CAN. Considering CAN FD with 64 data bytes, and 3.43Mbps as the average bitrate, the frame duration would be 166.6us for 5Mbps format.

It is possible to improve data transfer efficiency by packing more data in each CAN frame. The efficiency equation assumes the worst case number of stuff bits. Classical CAN has better efficiency compared to CAN FD due to its lower overhead. Increasing the number of bytes in the CAN FD frames from 8 to 64, it would be possible to increase efficiency from 50 to 88 percent. In many cases, long CAN frames with 64 bytes will be used during programming that is normally done when a system is on pause and there are no real-time controls running. Even without real-time demand, it is still preferred to use higher bitrate in order to increase data throughput and lower the download time [6].

Having extra 2 bits in CRC and 4 bits in the stuff counter and the fixed stuff bits make the CAN FD frame longer than a classical CAN frame. Therefore, it is not fair to compare due to having up to 3 stuff bits in the CRC part and 3 more bits in the control part. The extra bits in the CRC part of the CAN FD frames provide better protection to data content which is very important for safety critical systems.

In order to migrate from CAN to CAN FD, the data link layer needs to be made CAN FD compliant. It comprises the ISO Transport Protocol (TP) layer as well as CAN FD device drivers. Therefore, there might be two main steps. At first, the transport layer in CAN FD can transmit or receive longer data length with respect to CAN. Thus, some changes should be done in the transport layer according to the standards mentioned in CAN FD documents to support the 64 data Bytes payload and CAN FD message frames. Moreover, the CAN FD drivers are written with respect to the MCU family for the microcontrollers to transmit and receive CAN FD data frames. The bit stream processor is the component tasked with transmission and reception of CAN FD frames. Separate drivers are required for

each of these components to function. There are certain legacy microcontrollers that do not support CAN FD protocol. To solve this problem, it is possible to replace the microcontroller unit which means re-applying the development, integration and testing. Adding an external CAN FD controller would be a better solution. They function as slave devices and communicate using SPI with the network on behalf of the main control unit.

There are some options in order to make the classical CAN nodes tolerant to CAN FD communication such as partial networking, CAN FD shield, and CAN FD filter. In mixed networks or partial networking containing both classical CAN and CAN FD, it is not possible for CAN nodes to receive CAN FD signals and it will interrupt the communication and error arising. This problem would be solved by partial network functionality which uses the CAN transceiver standard called CAN with selective wake. CAN nodes are passive while CAN FD is communicating, however, they are not inactive and in a low-power mode. As a result, CAN is disconnected from the network during CAN FD communication, and when a valid CAN wake up message appears, the transceiver will wake the CAN nodes up and route the message to them, and this wake up message is transmitted by CAN FD node.

There might be two types of buffers in CAN controllers in which it is either allocated to hold only one special CAN message or the general one that is organized as single buffers, priority queue or FIFO queue [7]. There is another part in the system which is called acceptance filter for receiving message ID. In case of transmission, internal arbitration, a FIFO or the buffer number for transmission sequence are supported. It is possible to set a label to the message using a message marker which is stored in an event FIFO to order the transmitting many messages. The current error state and transmission or reception state information are given in CAN status interface. Only the messages which match acceptance filters are stored in RX message objects or FIFOs. The access to RAM is controlled by the memory interface and message objects are stored in it. RAM

also could store acceptance filter configuration. A register interface is used by microcontroller in order to access the SFR of a CAN FD controller. More RAM would be needed for message storage due to increase in the payload of CAN FD messages.

The CAN FD peripheral could share the system RAM. It dedicates the RAM that is inside an external CAN FD. SFR is accessed by the SPI interface in order to control the CAN FD engine aiming at the configuration of the CAN FD bit times and setting up the receiving filters. Also, the RAM is accessed to load transmissions and read the receptions which occur autonomously via the external CAN FD, and the microcontroller unit is interrupted as this event is successfully done. SFRs decrease the number of SPI transfers which results in having higher bandwidth. This external controller integrates a clock generator in order to supply the clock used in CAN FD. Additionally, it has to meet the identical requirements as an integrated CAN FD controller. Utilizing an SPI with DMA and enhancement in SPI frequency meets the bandwidth requirements. Simulations shows that using an SPI frequency of 10 MHz to 16 MHz in the external CAN FD controller could keep up with a complete loaded CAN FD bus at data bit rates up to 8Mbps [8].

Increased bandwidth demands rapid process of the messages by the controller unit, thus, higher controller clock speed and FIFOs to buffer messages are needed. Classical CAN controllers usually use the 8 MHz clock, but a faster one is needed by CAN FD. It is better to use the identical sample point setting in CAN FD which allows shorter time quanta (TQ) that means higher resolution for setting the sample point, and the same TQ resolution during nominal and data part. So, more TQ per bit would be needed during the nominal bit phase.

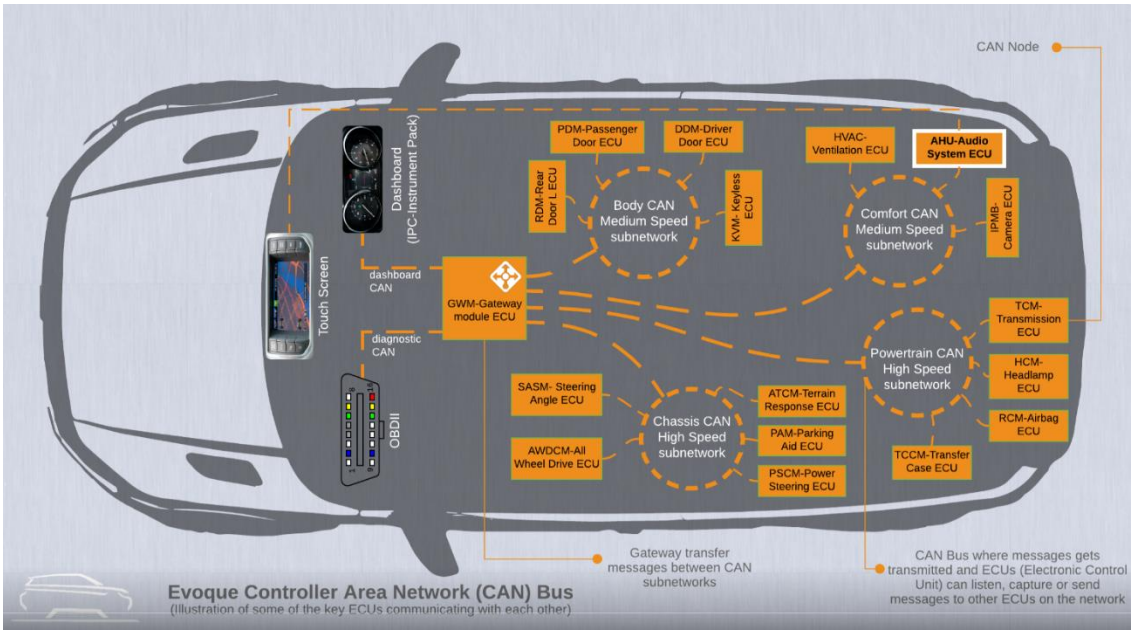


Figure 5 – Vehicle subsystems communicating using CAN protocol [9]

According to figure 8, it is possible to recognize different vehicle systems. Body system uses medium speed sub-network to connect doors and key locks. Comfort system contains heating, ventilation, audio system and camera uses medium speed. Powertrain system is another sub-network including engine functions, headlamp, airbag, and so on, considering this sub-network has to be high speed. Terrain response, steering, and wheel drive need high-speed sub-network and are included in the chassis system. All messages from these systems pass through the gateway to communicate the information to terminal ECUs such as infotainment and instrument panel cluster systems. The gateway includes at least a CPU for real time communication with other peripherals and this CPU is connected to the main CPU via an internal bus. Therefore, all messages are passed through the gateway to connect the main CPU to other modules. There are two types of gateways called direct and indirect gateway. When a message is mapped to the other CAN bus without the value being touched, the gateway is the direct one. On the other hand, if the gateway functions as an interface between different networks or sub-networks to integrate them considering the baud rate and protocol change, the

indirect gateway is used. Indeed, a gateway is used to secure the communication inside the network topology and each manufacturer might have a particular gateway to avoid the system being hacked. Moreover, when a module needs to be changed, the new module must be compatible with the gateway encryption of the producer. Therefore, it is not possible to insert an unlicensed module and gateway should identify the new module to have a successful and secure connection.

Chapter Three

3. Applied Methods

Understanding the preliminary information about the needed scientific background and the actual environment, the scope of the project includes two main approaches. The first approach is to analyse the files and topologies related to the project. Having the details about the interior modules, their functions and connectivity method, the second approach could be implemented. This approach consists of understanding how the current internal automated test bench tool is working with the CAN network, and upgrading it to automatically test the CAN FD network of the carline.

3.1. Testing approaches and validation

One of the most used project management processes is the V-Model which is divided into design and validation phases. Customer and vehicle requirements, E/E systems and component specifications are applied during the design phase. On the other side, the validation phase contains component testing, E/E systems validation and customer prospective validation. The purpose is vehicle production. It must comply with the requirement decided by the carmaker to satisfy the final client.

The car manufacturer splits the requirements of the vehicle, which are written using the combination of Benchmarking, innovation, client satisfaction and considering all subsystems contained in the vehicle, into sub-requirements and sends this to the different suppliers. The phase of validation represents the attention that the development procedure keeps on all systems/subsystems before the production. This phase is scheduled both during the development of the component and after the production during the maintenance of the product such as software updates and over-the-air services.

If the attention is focalized on the infotainment, it is considerable that this system integrates many features on-time and in the future. So as not to affect the production time, the carmaker uses intelligent strategies to validate all subsystems/systems. Figure 6 illustrates the activities followed by the research and development team based on V-model.

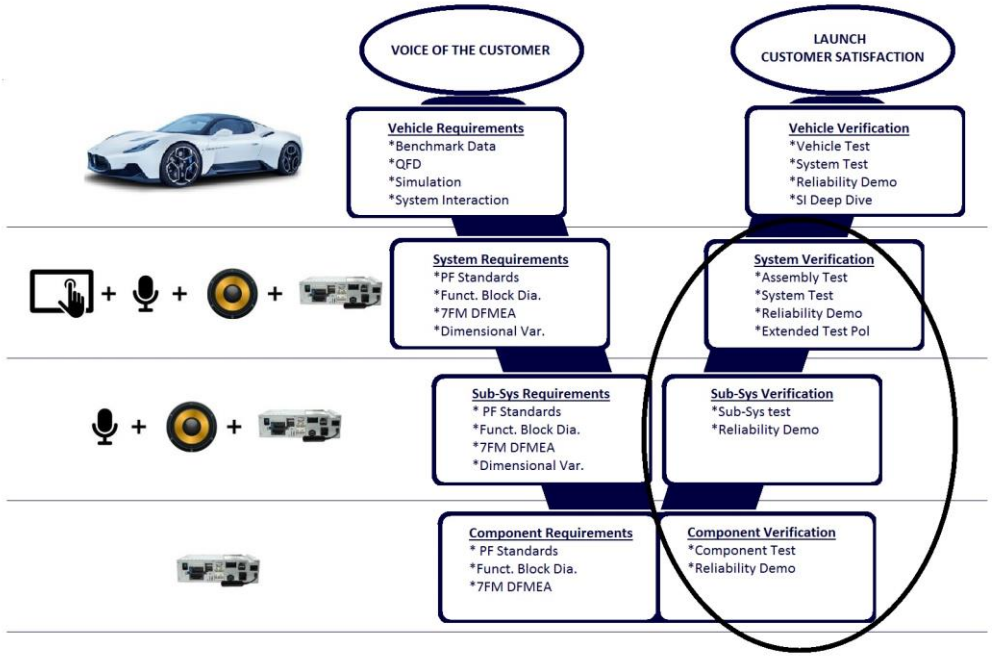


Figure 6 – Specific V-Model diagram focusing on the project scope

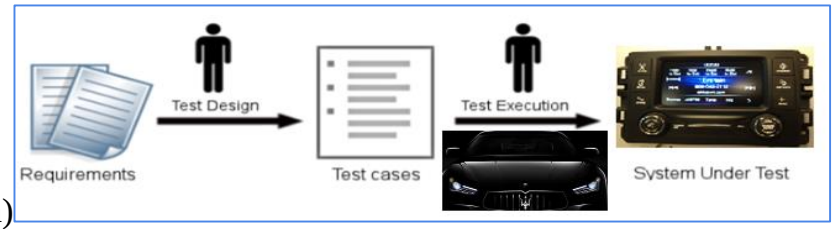
To validate that all subsystems are compliant with the different requirements, there are many testing procedures such as performance testing, functionality testing, quality testing, thermal testing, electrical testing and integration testing. All are managed by internal test norms which are standardized among the market of interest (EMEA, LATAM, NAFTA, APAC, etc.).

Defining the requirements and focusing on the infotainment, the validation team manages the validation activities and usually uses three types of testing approach ,as shown in figure 7, including testing on vehicle, on manual bench and on automated test bench, each having different motivations.

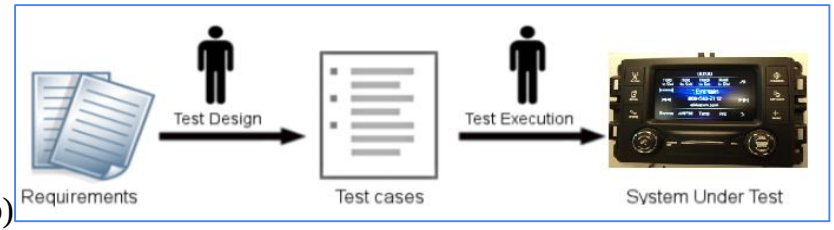
For the testing on the vehicle, for example, the team needs to have different prototypes passed through the infotainment carline under test each including relevant cost on the validation phase. Such activities are also executed on the manual test bench and the automated one.

The test is designed and some test cases are obtained in all mentioned types, however, in testing on vehicle type, the test execution is applied on the system under the test in vehicle environment; therefore, a vehicle is needed to perform all needed test activities by the test engineer. Testing on manual test bench, on the other hand, is performed in a small part of the vehicle which the bench is equipped with, and the test activity is performed on different subsystems of the vehicle individually by the test engineer. The third type is the purpose of this project in which a script is provided after test implementation to a test execution tool, which is replaced by the test engineer, in order to apply the test on the system under the test.

a)



b)



c)

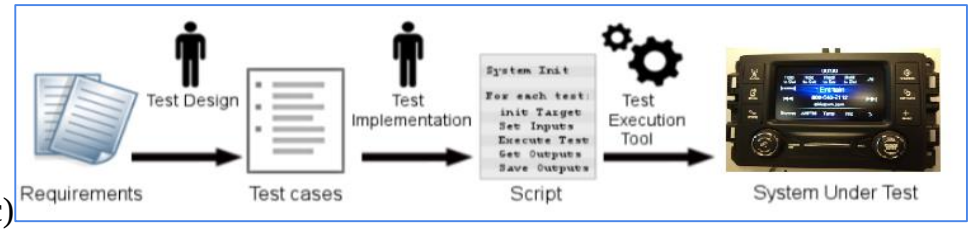


Figure 7 - Testing on a) vehicle b) manual bench c) Automated Bench

As the time goes on and the technological features expand, a precise testing is demanded to check if the product meets the user needs and improves the quality. Automated testing using an internal Automated Test Bench requires a perfectly developed tool to cover all aspects and aims which have to be done during the test phase. This development has to work non-stop for 24 hours to increase the testing speed and lower the business expenses. In addition, the accuracy of testing is enhanced due to the increase in the test coverage and elimination of the human error, in spite of missing the human experience. Reusability of test suite, having the immediate feedback, and the ability to repeat the test with the same timing and condition are the other undeniable advantages to convince the developers for having the automated testing procedure, however, this choice depends on the available time for testing the projects to evaluate which choice is preferred due to the time demanded to write a test script. Furthermore, some features are unfeasible in automated testing because these analyses require a real environment. These

features contain the options in which a human is needed to communicate the system or the navigation feature of the vehicles. Another disadvantage is the fact that the analysis of results would be done by a test engineer to remove the false failed test.

The current automated test bench used by Maserati does not contain this feature yet. Therefore, the scope of this work is upgrading the current automated test bench to manage the CAN FD which results in a well-planned validation activity. Thus, the objective of the project is mainly to expand the functionality of the automated Test Bench on the new generation of carline projects using CAN FD. For example, in the future BEV vehicles, the CAN FD will be integrated in their architecture.

Considering the requirements and the procedures to build an automated test bench, the objective of the project could be defined as creating a script to manage the CAN device to simulate the messages of the ECUs present in the network using CAN FD protocol, and then to integrate this script on the current internal ATB Editor tool after upgrading the tool to be compatible with CAN FD protocol. The current internal tool, ATB Editor, is used to manage and control the automated test bench.

The tool not only controls the bench instruments such as audio analyser, RF generator, power supply etc. but also creates the CAN network to be able to excite the ECU under tests by simulating other ECUs to which the latter is really connected on the vehicle.

Thanks to ATBEditor, the infotainment validation team writes test cases in the form of scripts to create a test sequence loop in compliance with Stellantis standards and requirements. It currently allows its user to manage and monitor the 2.0 type CAN network.

The code created for this purpose will finally be integrated into the ATBEditor tool, to allow the validation team to start writing the test cases for the new infotainment systems.

To achieve these objectives, after having fixed and defined the prototypes of useful functions, it was decided to write the code in Visual Studio IDE with the C# language.

3.2. Database Container (DBC) file and CAN Log file

Working with Controller Area Network (CAN) data is for the most part an exercise in understanding formats and translation. While working with CAN data, it's never long before the subject of the DBC file is introduced, because this is the most common way to handle identification and translation of the data. Specifically, it is being referred to the identification of CAN messages and the translation of the raw CAN data, as transmitted within a CAN frame, to meaningful values and information.

The DBC file type was developed by Vector Informatik GmbH in the 1990s to provide a standard means of storing information described in a CAN network. Used by the automotive industry primarily, Vector database files (.dbc) have since become the de facto standard for exchanging CAN descriptions. Similar standards operate for other bus systems, such as FIBEX database files (.xml) for Flex-Ray and LDF for LIN (.ldf).

The SAE J1939 standard is written and maintained with a complete understanding of the DBC file. SAE J1939 is used in the commercial vehicle area for connection and communication throughout the vehicle, with the physical layer defined in ISO 11898. [10]

Table 1 shows some rules and useful information considering that the CAN network of the car can be extrapolated from the DBC format.

Message definition	BO_ <CAN ID> <Message Name>: <Message Length> <Sending Node>
Signal definition	SG_ <Signal Name> [M|m <Multiplexer Identifier>]: <Start Bit> | <Length> @<Endianness><Signed> (<Factor>, <Offset>) [<Min> | <Max>] "[Unit]" [Receiving Nodes]
Description field	CM_ [<BU_|BO_|SG_> [CAN ID] [Signal Name]] "Description Text"
Value definition for Signals	VAL_ <CAN ID> <Signals Name> <Value Table Name | Value Table Definition>
Sending type of a message definition	BA_DEF BO_ "GenMsgSendType" ENUM "cyclic", "Triggered", "Cyclic if Active", "Cyclic and Triggered", "Cyclic if Active and Triggered", "None"
Cycle time of a message definition	BA_DEF BO_ "GenMsgCycleTime" INT 0 0
Value definition as long as no value is set/received for this signal	BA_DEF SG_ "GenSigStartValue" INT 0 0

Table 1 – information obtained by DBC file

These were used to properly simulate the desired CAN network. According the figure 8, a practical example could be mentioned for clarification.

```
BO_ 200 SENSOR_SONARS: 8 SENSOR
 SG_ SENSOR_SONARS_mux M : 0|4@1+ (1,0) [0|0] "" DRIVER,IO
 SG_ SENSOR_SONARS_err_count : 4|12@1+ (1,0) [0|0] "" DRIVER,IO
 SG_ SENSOR_SONARS_left m0 : 16|12@1+ (0.1,0) [0|0] "" DRIVER,IO
 SG_ SENSOR_SONARS_middle m0 : 28|12@1+ (0.1,0) [0|0] "" DRIVER,IO
 SG_ SENSOR_SONARS_right m0 : 40|12@1+ (0.1,0) [0|0] "" DRIVER,IO
 SG_ SENSOR_SONARS_rear m0 : 52|12@1+ (0.1,0) [0|0] "" DRIVER,IO
 SG_ SENSOR_SONARS_no_filt_left m1 : 16|12@1+ (0.1,0) [0|0] "" DBG
 SG_ SENSOR_SONARS_no_filt_middle m1 : 28|12@1+ (0.1,0) [0|0] "" DBG
 SG_ SENSOR_SONARS_no_filt_right m1 : 40|12@1+ (0.1,0) [0|0] "" DBG
 SG_ SENSOR_SONARS_no_filt_rear m1 : 52|12@1+ (0.1,0) [0|0] "" DBG
```

Figure 8 – Illustration of some information provided by DBC file

This example illustrates how the message “SENSOR_SONARS” with ID “0xC8” is sent by node “SENSOR”. The mapped signals made of 8 bytes of the message data frame are received by nodes “DRIVER”,” IO”, “DBG”, and each signal occupies a precise bit position within the message.

Each message begins with “BO_”, and contains the message ID, name, length and transmitting module. Moreover, each message includes some signals that begin with “SG_” containing its name, start bit to mark the start point in the payload and signal length. In addition, this line describes the byte order separating signed or unsigned value type, scale, offset, minimum and maximum values, and the receiver. This line also indicates if the byte order is Motorola or Intel to give some information about the mode of storing in memory data of higher dimensions to bytes. Actually, these modes do not change the position of the bits in the bytes or characters in the string. It is important in decoding multi/byte strings of characters. Specifically, they could be classified in big-endian and little-endian, considering the transmission or storage starts from the most significant to the least significant byte is applied in big-endian; on the other hand, the transmission or storage in little-endian starts from the least significant to the most significant byte. The first group is known as Motorola byte order and the second one as Intel byte order.

To make the values more user-friendly, two other lines are added to the DBC file. Beginning with “BA_”, depending on if it is marked for messages or signals, the parameters are specified and their values are defined as desired names in another

line starting with “VAL_”. Another line beginning with “CM_” includes comments to add some descriptions and make the file more readable.

In order to match CAN ID and DBC ID, if it is 11-bit identifier, the decimal values of them are the same, otherwise, in case of extended format with 29 bits, a 0x1FFFFFFF mask has to be applied to the 32-bit DBC ID to map extended CAN ID.

Understanding the requirements of the DBC files, the first idea, to be developed, was to define two prototypes of useful functions in the software part consisting of reading the DBC file to understand the format protocol, and filtering the CAN data by focusing on the infotainment system.

In order to work with some of the participating modules, filtering the messages and writing a dictionary to have a collection of messages that the specific module takes part in making the testing plan clearer. The first phase is to check the transmitting duty, thus, according to the format of the DBC files, it is possible to find out which module plays the role of the transmitter or receiver.

Another file that was taken into consideration before the software development was that of the CAN log of a prototype vehicle. When the modules are configured and connected to each other, it is possible to get a log file to test if the modules work and communicate well. This log file is obtained in the field test which is obtained in the vehicle or while checking the devices in the test benches. In fact, the test benches are the hardware in the loop testing in which the engineers try to simulate the system as a real environment. Due to having an analysis about the integrity and performance of the ECUs, a lot of information could be found in the log file including the message names and frames, the timings, allocated channels for each transmission, and so on. Therefore, using the DBC file and log file, all detailed information about the transmission and reception are available.

Indeed, two types of logging format are available, message-based and signal-based. The message-based format contains information about the bus traffic and communication; therefore, all the messages are logged while other statistics and disturbances of the bus take part. Actually, this file is used for offline analysis of the network while the behaviour of the traffic is evaluated. On the other hand, the signal-based one stores signal values of each message transmitted over the network. All information about each message and network communication will not be available in this method and the related DBC file would be needed in order to decode the signal values. The format of the logging files is ASCII and it supports messages of all bus systems, variables, internal events and comments.

Highlighting what happens in the network while the system is running, the DBC and log file should be compared. Having the details about the messages and value description in the DBC file, the message frames are translated.

Focusing on the byte ordering method, starting bit and the length, it is possible to find out which aim the ECU follows in each frame. Considering this type of output file, it was decided to use it initially to create a macro list containing the values to turn the device on/off by the wakeup/sleep patterns. It is applied to validate the CAN FD simulation and to verify if the developed testing system works consistently with what happens at the vehicle level.

3.3. Software implementation

Each network taking part in the topology has its own internal DBC file in which all the information about the signals, messages, bits, values, and the transmitter or receiver roles are available. To allow the ATBEditor tool to read the new DBC files created for electrical vehicles which have CAN FD networks inside their architecture, the first function examined and modified for this purpose was that of

READ_DBC_FILE. Using this routine, the DBC file is loaded to the tool and some functions are coded to extract and save all data thanks to the determined format of these files. Moreover, this made it possible to extrapolate the entire message dictionary with the corresponding signals that make up the desired CAN FD network and the characteristics of the CAN FD such as baud rate, BRS value for all messages, and so on. A filter was then applied to characterize the network which might be useful only for the infotainment system. This filtering algorithm was created considering the infotainment system as a Black box and this idea can be illustrated as Figure 9.

Figure 9 – communication scope focusing on the focused ECU

Only the messages received from the radio (RX messages) and those sent by the radio node (TX messages) were therefore considered. The created message dictionary made it possible to simulate the CAN network in compliance with the vehicle requirements.

The result of this first part of the work made it possible to obtain the useful infotainment data to be sent with the timing of each message once the CAN network is enabled and the baud rate and the setting characteristics of the desired CAN FD network.

In the software side, at first, the software reads the DBC file and gets all information in the network to create a dictionary of CAN FD messages, applying the filter considering the ECU like a black box, and the type of CAN protocol. Second, CAN device ports are set properly and a routine is created to open the port and send or receive the messages on the network. Moreover, the code needs

to be debugged and checked on the infotainment component. Debugging the code to check in on the infotainment components is considered as the last step before integration into the internal tool to upgrade it. As the last step, the upgraded tool is tested to check if it works properly considering the complete networks and the systems. Having a successful procedure without any bugs, the code is integrated into the internal tool called “ATBEditor”.

The hardware setup built to create an active communication based on the CAN FD network, consists of the standard Stellantis cabling connected to a device used to simulate the CAN network including four ports, two of which can be used to manage the CAN FD.

In order to properly set the CAN device ports using the information obtained by the DBC file, the procedure is to open and activate the channel first. This part is implemented by polling the driver and getting the all channel configurations of CAN device by the vector owner’s specified methods.

The application gets the information on the hardware configuration and it is possible to call it any time after having a successful driver opening. If the device is not available, the program exits from the main function; otherwise, it proceeds with the selection of the chosen port and opening the mentioned port considering CAN protocol specifications. Later, the initial access is checked. Multiple CAN applications could use a common channel at the same time. The first port that has the access to a certain channel gets the property initial access for the mentioned channel. If a different application asks for the access on the channel, depending on the bus type, applications could access the channel simultaneously without initial access. Therefore, a specific structure is provided to open a port for the bus to permit access to the different channels selected by Access mask. Opening more ports on a specific channel is possible, however, only the first one gets the initial access.

The driver channels could be identified by an application specific index. To access one or more available channels, a bit mask based on the channel index is required considering that the mask is equal or less than the index.

If there is the initial access, some information including channel parameters, channel bitrate, channel output and channel transceiver need to be set.

Not having the mentioned access permission makes the program exit from the main function. Considering the difference between the classic CAN and CAN FD, it is crucial to configure the channel and communication relevant to the network trying to transfer data. For example, if the channel is configured as CAN, transferring CAN FD messages raises some errors like incorrect packet size, bitrate difference, and sample point choosing, and so on. Therefore, CAN FD properties such as baud rate, arbitration values, receiver/transmitter mode need to be set correctly as this permission is obtained in the initial access level. Later, the channel is activated and clock resetting is performed.

Having all the configurations and settings ready, the application is ready to transmit and receive the messages. The application takes the advantage of using buffers and queues to store the events and act with respect to the priorities. The logic flow described above and implemented is illustrated as follows in Figure 10.

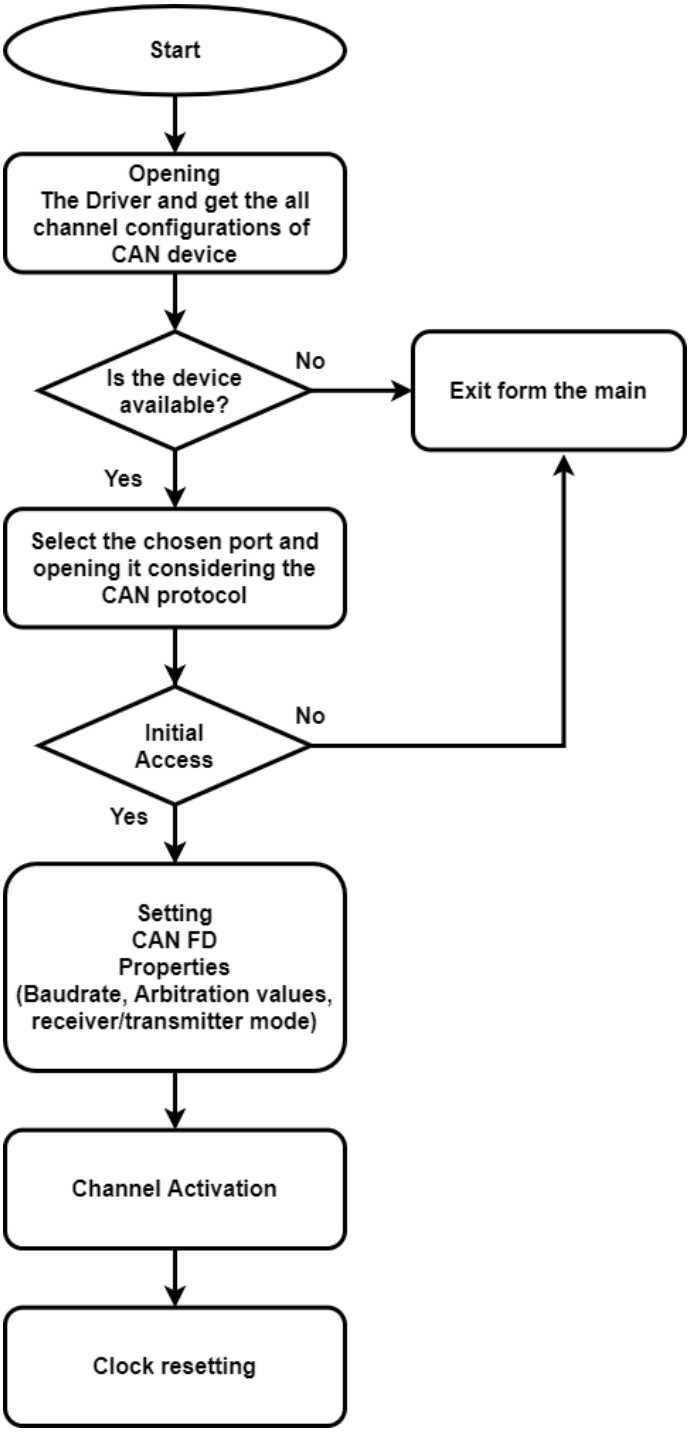


Figure 10 – Block Diagram for initializing the program

Considering figure 10, the function that performs this workflow is called INIT_DRIVER. To send properly the messages of the created dictionary, the

implemented software code contains functions both for transmitting or sending, and for reading or sniffing.

The first function is a subroutine that allows the program to send the message that the infotainment would get as the input and simulate the behaviour of other ECUs connected with the infotainment.

The second function is another subroutine that lets both the program to read the messages that the infotainment system transmits and to verify that the simulated messages have been sent.

Deeply analysing the mentioned functions, input infotainment received messages list, which is based on proper timing of the message puts the information to create the message event collection and configure it. This configuration focuses on the Bitrate Switch bit, flags, and length.

CAN acceptance filter is used to compare the received CAN ID bits with the related bits in the register set which lets the messages pass. There might be different ports with different filters for the channel and if the hardware cannot implement this filter, it will be virtualized by the driver. Implementing these filters does not mean that the application will not receive the blocked message IDs and it depends on the configurations which have many ports. It is notable that the application accepts all messages by default unless the filtering is implemented by the application considering the methods to have a flexible interface for filter configuration.

This method gets the advantage of some acceptance codes and masks to filter IDs and a range to distinguish if the filter is used for the standard IDs or the extended ones using flags. Adding or removing the acceptance filter would need a range determined by the first ID and the last one. Then, the created collection is transmitted and the success flag is checked to start this loop again if the successful status is obtained or to write on the CAN Trace log file if the unsuccessful status is obtained to indicate which type of malfunction or error is detected.

On the other hand, input infotainment transmission message list based on timing of the message provides the information to create the message receive event and wait for the queue. Depending on the port, the queue level is evaluated in bytes or number of events. Meanwhile, as mentioned above, input infotainment received message list is also another information which is considered to create the events and queue.

Some functions would help to block the calling thread until the windows application event reaches the signalled state and reset the event to the non-signalled state as the thread execution continues; therefore, the application would wait for single or multiple objects. Thus, the success flag is again checked to report the malfunction or errors on the CAN Trace log file if it fails, or to update the transmission/reception message dictionary and write on the CAN Trace log file.

Retrieving the description of the event as a text, flushing the transmit and receive queue of the channel, reading the number of events or bytes which are in use in the receive queue of the port to compare the actual queue usage to the specified queue size, and asking the chip state of the CAN controller are done in this stage are applied. Also, all available messages are read to determine for re-enabling the event. The overrun would be returned as a flag to mention the overrun of the receive queue and the description of the error codes are retrieved as a text string.

The two functions created to manage the transmission and reception of CAN messages on the network are called TRANSMIT_MSG_COLLECTION and MSG_ SNIFF.

The workflow showing the connection and the logic of the code implemented for the two functions is shown in figure 11.

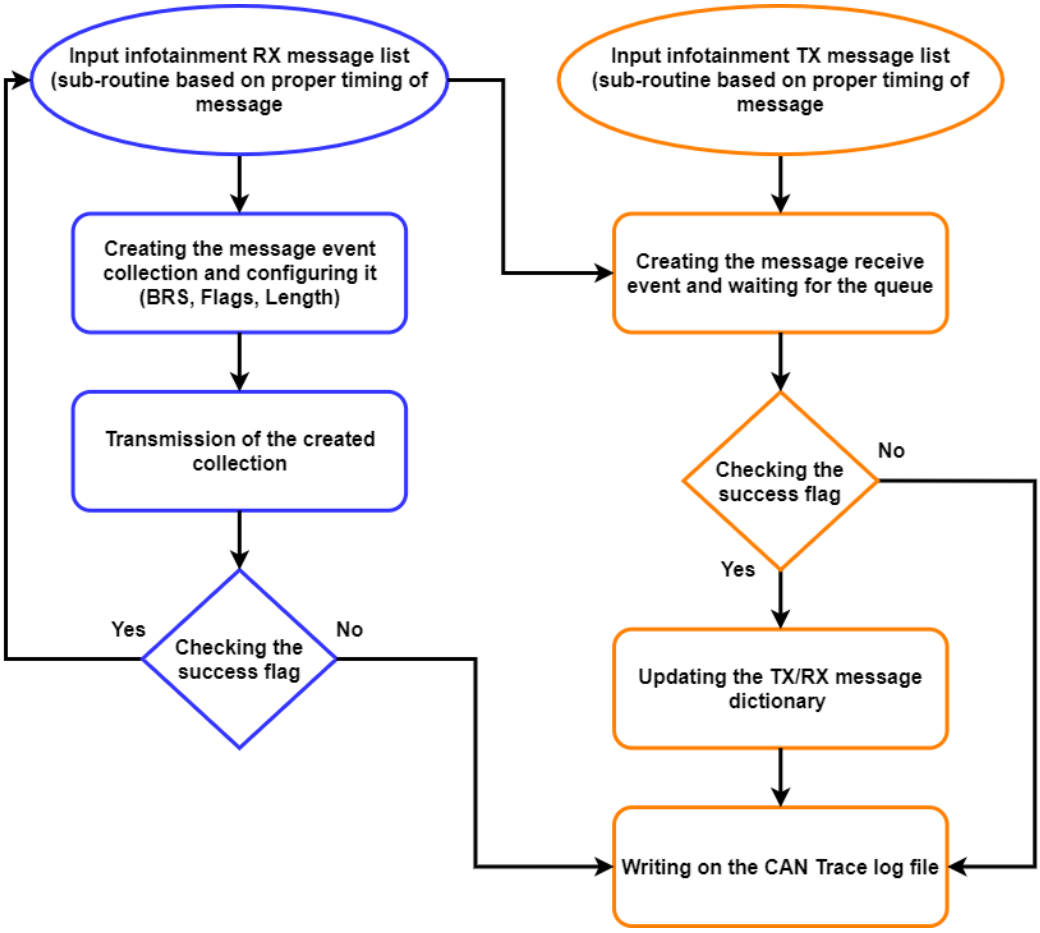


Figure 11 – Block Diagram for transmitting and receiving messages (work logic)

Ultimately, the program terminates the transmission and receiving procedures and the channel deactivation makes the chosen channels go off the bus. Therefore, the program deactivates the channel if there is not any port that activates the channels, and further closes the ports and the instantiated object to manage the driver to finish the workflow of the program according to figure 12.

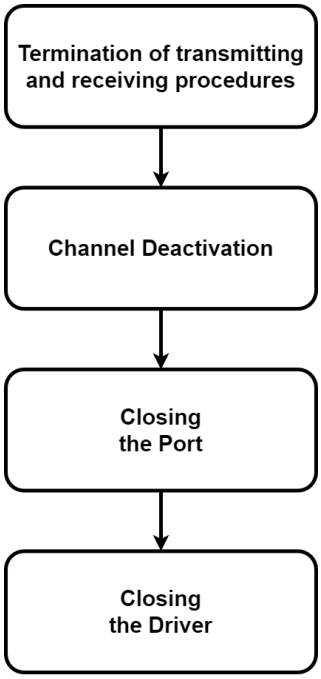


Figure 12 – Block Diagram for terminating the process

The debugging phase makes it possible to verify each single line of the code and check the variable value of interest. Indeed, the Visual Studio environment lets the developer insert the intelligent breakpoints and to use the console to stamp the value of interest. Therefore, it is possible to check the data flow and the progress using the flags and checking for the errors or malfunctions systematically during the debug.

Compared with CAN 2.0, CAN FD functions differ in configuration setting, transmission and reception parts. As these two protocols are different in the arbitration part and data part of the message, the configuration would get some different information for timing and bitrate settings like the interface version and timing calculation details. Thus, it will influence transmitting and receiving events and event threads or strings on the selected port, and consequently the reporting flags.

3.4. Defining a Macro List

Once the software architecture and the main functions for the control and management of the CAN FD network have been created, the activity moved on to the development and definition of the messages used to excite the infotainment system based on the desired functions and to be tested.

This list is obtained by checking the CAN log and the requirements (Vehicle function documents) to understand different functionalities, states and detailed information about the modules; therefore, it includes some important values of the signals in the messages like wakeup/sleep, cranking, ignition on/off and engine on/off, battery status, battery level, etc.

While debugging the code using the information in the macro list, it will be possible not only to verify if the written code is working correctly but also it helps the validation team in writing the test cases by having radio feature blocks already implemented to specification. In addition, it will be possible to verify if those blocks work fine. Indeed, a package of test cases have been created to allow the test engineer to verify the integration phase and implementation, validate the infotainment and have a starting point to create new tests for component validation to help the validation team.

In fact, the ATBEditor tool helps the test engineer to write test cases using and integrating pre-packaged macros which implement the low-level requirements, therefore the same for the CAN.

To complete the porting work for the CAN FD, another critical activity is writing a macro list compliant with the requirements of the architecture of electric vehicles. It includes many benefits such as minimizing time to write the test, reducing human errors while setting the values for the messages, having the requirements linked and so on.

3.5. Software Integration and Design

In the configuration tab of ATBEditor, the test engineer chooses the carline model and the related properties. Then the DBC files are needed to be loaded and according to each DBC, the list of ECUs are available in both “Real CAN Nodes” and “Not Simulated CAN Nodes”. Depending on the test, the user should select the relative ECUs from the Real CAN Nodes list and the selected ECUs are deleted from the Not Simulated CAN Nodes as expected. Furthermore, another interesting feature is performed in this tool to show the transmitting and receiving messages of the related ECU based on the uploaded DBC files and the test engineer is able to check this dictionary by clicking on a single button in front of the corresponding DBC file.

Once the coding part is finished, some graphical changes are applied to the previous version to include all additional features considering hardware settings, DBC files and macro commands. Focusing on the pop-up window for hardware settings, the new version is able to customize the number of demanded channels and once the user specifies the number of channels in each network, the bar is activated to choose which channel port to allocate based on the provided ports of the device. Moreover, as the number of available channels are increased, the tool can get more DBC files as the input depending on the test and ECUs. When the user chooses an ECU to simulate, another feature is taken into account to show which DBC files it refers to. To notify this, a checkbox is put near each field corresponding to the DBC file and the test engineer can understand if the uploaded DBC file is used in that test procedure.

Another integration task was aiming on updating the macro list to be compliant with CAN FD testing. This feature relates to the test manager tab of ATBEditor. While the user adds the macro commands, these commands are called from the DBC files loaded in the configuration part; therefore, the user should consider to

add all DBC files concerning that test, otherwise, the tool will apply the commands partially and the test will fail. To exemplify, some messages are needed to transfer to the display ECU and these messages are defined in a classic CAN network and the display is waiting for some values existing in that message to perform well. Not loading all corresponding DBCs leads the test engineer to have some messages missing in the script, and the test bench would not perform as expected.

3.6. Debugging

To simulate the radio unit, the CAN BH and CAN FD networks are communicating with the rest of the vehicle. In order to have a visualization overview to validate if the radio unit communicates with another ECU, the display unit is also added to the Device-Under-Test (DUT) system. This ECU demands two other CAN BH and CAN FD networks to work. On the other hand, while applying these tests, the test engineer should be aware of the networks and their ECUs and during the test design, many considerations are needed focusing on the network topology and DBC files.

The procedure is to add some commands, according to all analyses applied to the CAN log file and resulted in writing macros. In this case, it is better to check if another signal or message sends another value for the same parameter to avoid value collision. The initial values are noted in the DBC file, therefore, the procedure will be based on those initial values according to the DBC file unless the test engineer changes them or the simulation changes the values due to the events. It is also possible to enumerate the values using ATBEditor easily by selecting the corresponding DBC file, message, and the signal due to having all information previously saved using the DBC loading routine of the program. Moreover, some messages pass through the security gateway of the vehicle. This gateway translates the messages following its algorithms and functionality,

however, the message ID and the values in the input of this module is the same as the output as mentioned before.

In the hardware part of the project, ATBEditor is connected to the CAN device and depending on the system, the number of ECUs and the configuration of the test, this device is connected through a car cable to the aiming ECUs, specifically the radio unit and display. Another cable should connect the radio unit to display that is called Low Voltage Differential Signalling (LVDS) cable.

To finalize the simulation and start testing, writing a complete and precise macro plays an important role due to the fact that some messages might be received by the target ECU from another ECUs in the network. This event raises some errors and malfunctions due to the probability of message repetition, information diversity, periods of the messages and so on. A simulated ECU in the DBC file might send a relevant message to state the system is in ignition-off, however, the macro commands are written to act in ignition-on status. In this situation, the same ECU receives two different information and hops between these two statuses according to the periods of the concerning signals. Moreover, some signals received by the display contain some information for the display intensity, day/night status, etc. which are not macro commands to change the status of the ECU for testing purposes, but might decrease the timing for solving the reason why the system does not perform as expected. All mentioned notes are achieved only if the test engineer has sufficient knowledge about the network and the functionalities the testing procedure takes into account. It highlights the importance of having good knowledge about the details of the network topology used in the subsystem.

In ATBEditor, it is possible to check the values of the signals in each step and write some signals or messages individually from the macros if needed. These two options are crucial to find some errors or diversity regarding the information the macro puts in the bus.

Once the power supply, CAN device, PC, radio unit and display are connected, the test engineer runs ATBEditor to configure the system. The configuration includes loading the relevant DBC files, carline and writing some descriptions for it. This configuration is saved and the user sets the hardware options to add hardware properties. In hardware option pop-up, it is possible to choose the number of needed channels in each CAN network like B/BH, C, FD, and other features like webcam options if needed. Later, the test engineer should apply some commands and the test sequence. In the test manager tab, the commands of macro, reading/writing signals and messages, delays and user requests (pop-ups) could be inserted. Thus, the test is run when it is saved and the user can check the test execution status to analyse if the test is passed or failed. ATBEditor saves the raw text file of what happened during the test as a report and it is possible to check every message value applied by the macro or any checking method was considered. This is a notable validation activity because the test engineer can check every single value or save it in the test report for further analyses.

Another option in this tool is provided for the HMI manager to turn the display on/off without writing any test case. The needed macros and messages are applied in the code to provide a button key to the user. Therefore, it is a good method to check the Human-Machine Interface (HMI) feature of the subsystem without applying a complete test script.

It is also possible to turn the webcam on if it is configured and take screenshots from the environment for further plans concerning image processing. Regarding webcam, ATBEditor is capable of setting the camera options and adjusting the vision features to calibrate the quality and take multiple screenshots in each step of testing from the display for further reports. It is also possible to take the advantage of having the camera by taking some screenshots to have a complete overview on the status of the subsystem. Furthermore, it is possible to define some patterns using these screenshots and save them to apply some image processing methods. The test engineer is able to use these image processing functions to

validate the status of the display. For example, some screenshots can be taken by the webcam in the desired moments of the subsystem and the related functions can be applied in the test script as image processing macros. Later, to automate the validation of the taken screenshot, the defined patterns are searched using the image processing algorithms and the result is inserted in the test report provided by ATBEditor.

The following figures are the screenshots of the ATBEditor environment used to start test procedures.

When a test engineer runs this tool after connecting the CAN device, ECUs, and the computer to each other, figure 13 will be the first window to configure the test. In this part, the user can specify the model, write the description, load the needed DBC files considering their network type, and choose the target ECUs participating in this procedure.

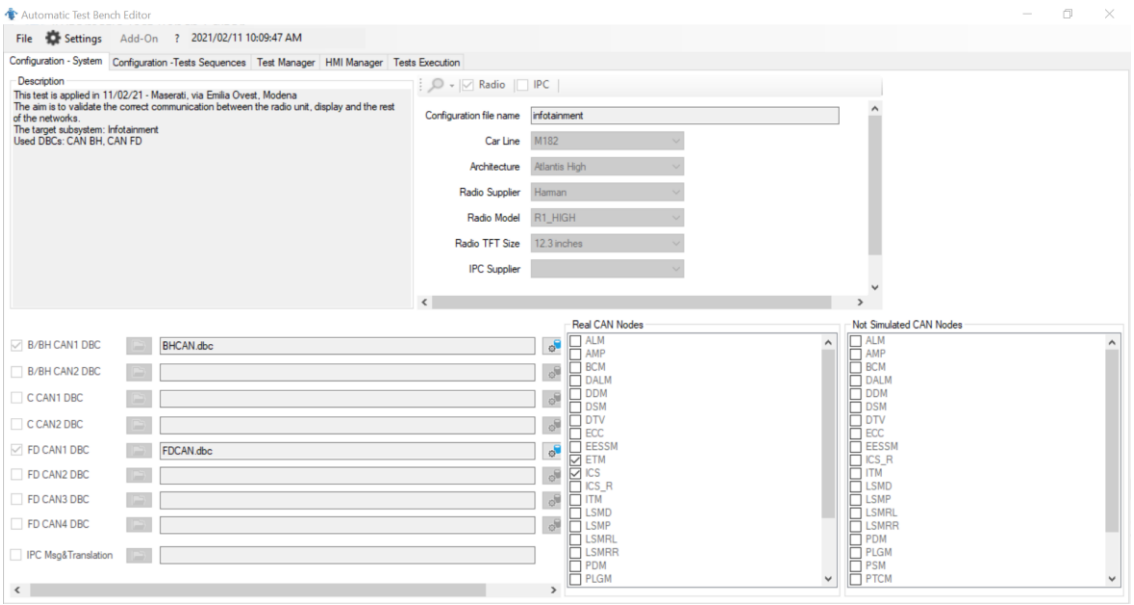


Figure 13 - The first interface of ATBEditor

Then, from settings, the test engineer must configure the hardware options. In this window, the devices and channels are needed to be configured depending on the networks. It is possible to apply the number of channels communicating using the CAN device. Figure 14 is dedicated to B/BH CAN, considering classic CAN 2.0 network of the project scope, and figure 15 is the CAN FD part of the project applied in this tool.

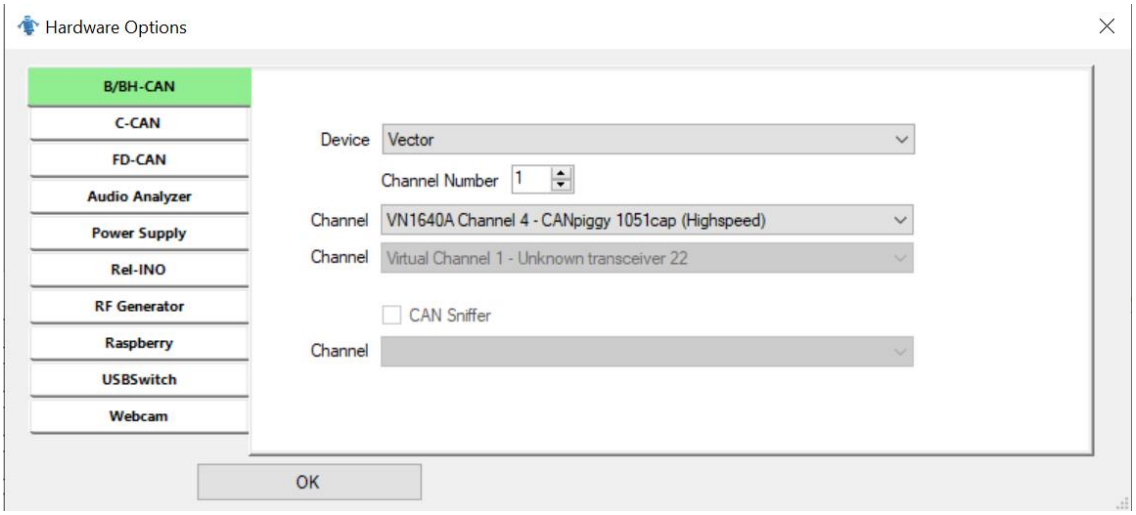


Figure 14 – B/BH CAN part of Hardware options menu in settings

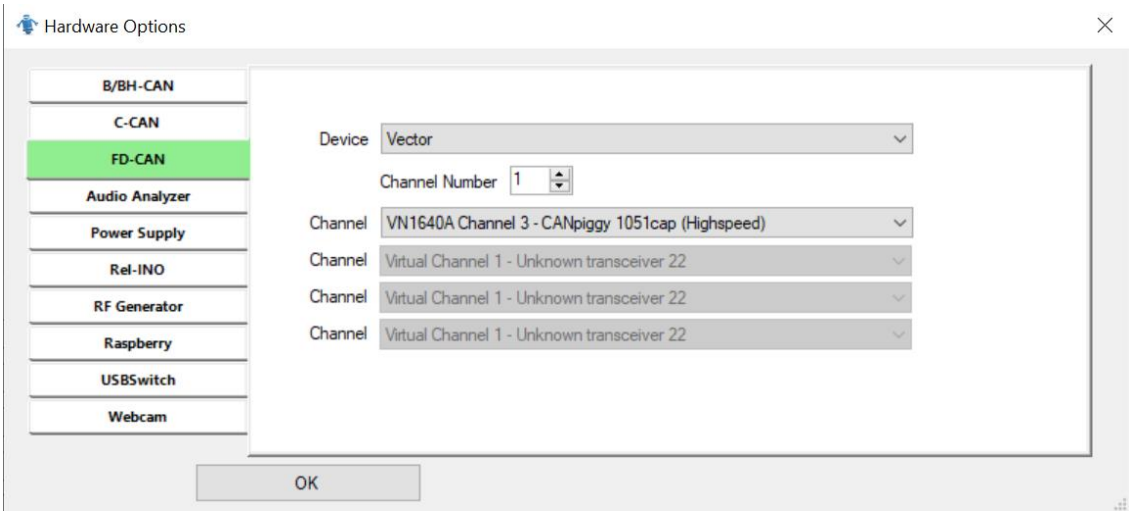


Figure 15 – FD CAN part of Hardware options menu in settings

During the test procedure, it is possible to add some image processing commands like taking screenshots, pattern recognition, and so on. To be able to use these features of the tool, it is also possible to connect a webcam and configure it in the webcam tab of hardware options window; therefore, the information about the connected webcam device appears in figure 16 if it is connected to the computer.

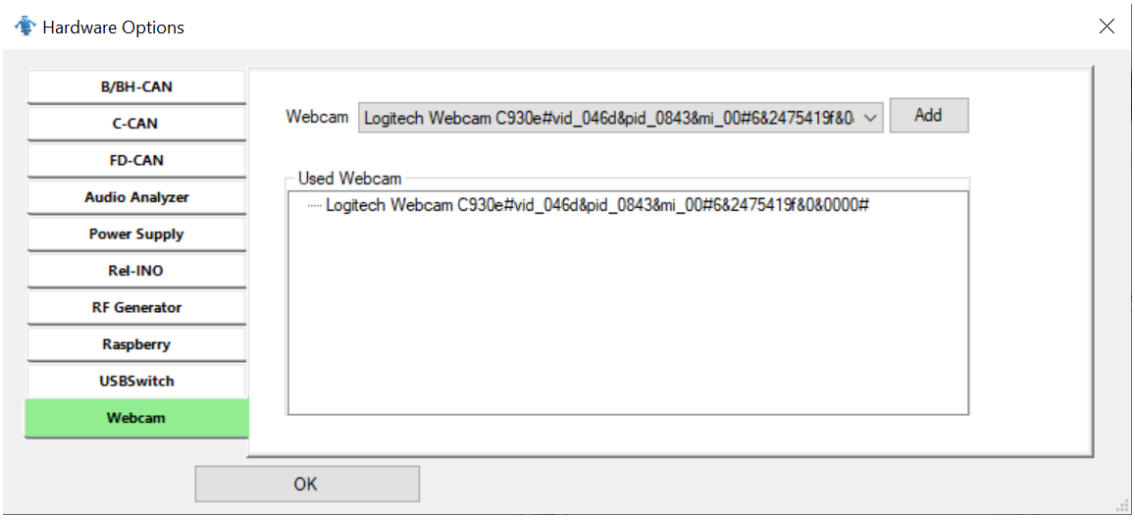


Figure 16 – Webcam part of Hardware options menu in settings

Having all the configurations applied, the user must write the test script and apply the commands in the "test manager" part of the tool shown in figure 17. Thus, all mentioned commands of the macro list are available in this page to add and create the particular test depending on the test scope. As the test engineer adds the commands in the "Add command" menu, the macro command is shown in the bottom part of the window including the names of all messages called from the loaded DBC files. It is possible to specialize the test script by manually adding and reading messages and signals available in those DBC files. Furthermore, by adding the image processing commands, it is possible to capture images for deep analyzes, or inserting pattern recognition commands to validate the captured

screenshots. These patterns must be defined in advance before starting the test procedure. Therefore, the user chooses the predefined patterns by the pattern list to insert the pattern recognition command in the test script.

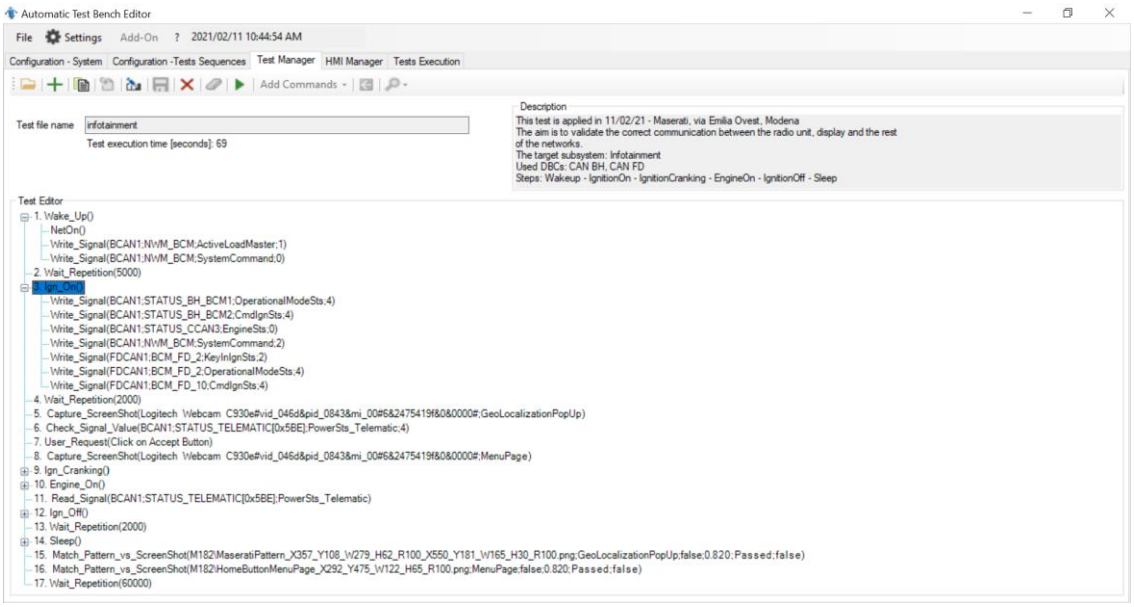


Figure 17 – Test Manager Tab including the test script

Having the test script ready, the user executes the test to receive the test result like figure 18. This test is able to be executed many times to validate the system by applying the same script in the same condition. This is one of the benefits that athe applied scope has added to the procedure. The tool reports the passed or failed test by a green or red flag in the “Test Execution” tab. Moreover, a test log including a test report and screenshots are saved in the directory of the tool for further analyses.

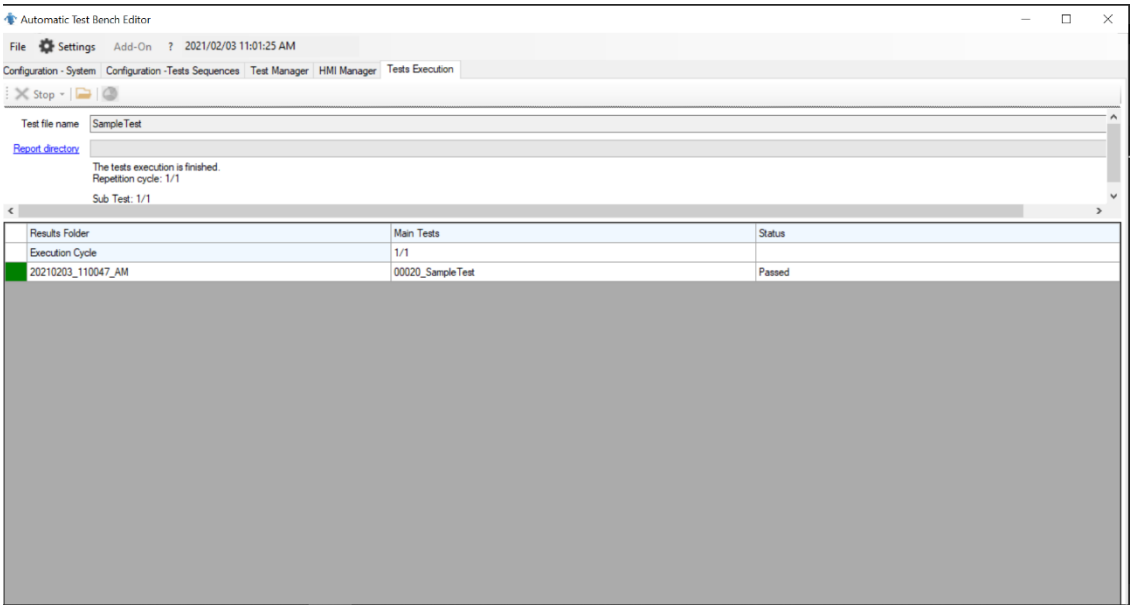


Figure 18 – Test Execution Tab including the Result and directory of the report

Chapter Four

4. Test Case

Once the new version is released, it is able to test the subsystem similar to Hardware-in-the-Loop (HiL). Thus, the test engineer aims on the ECU and simulates all other ECUs communicating with it, and it is applied by sending all signals and messages compliant with the provided details in the DBC file such as the default values, timing periods, size, message IDs, etc. The focus is on the validation of the infotainment system of the vehicle, so depending on the network topology of the carline, the networks and their DBC file are taken into account.

To create a test case, the applied procedure contains waking the system up, performing the modes such as ignition on/off, turning the engine on, and sleep. Therefore, according to the macro list defined before, it is possible to add some commands, and apply some features and messages if needed.

Considering a scenario, the customer opens the door of the vehicle which means a wakeup command to the system. This procedure might take a few seconds applied to the test case as a delay command. Later, the ignition-on command is simulated using the macro list as the driver inserts the key to turn the ignition on. The video frame could be captured from the display and it is expected to show the logo of Maserati as a requirement of the start. Passing these steps, the radio unit should transmit some signals to inform full operation mode to the devices. As the procedure is to test the functionality, a check signal command is put in the test case to assure if the radio unit works fine and transmits this important signal. The cranking procedure begins as the driver wants to turn on the engine for driving aims and consequently, the engine starts working. To determine the performance and test the system functionality regarding the engine, the relevant signals are

checked to assure if the values are as same as the predefined values in the macro list. Then, when the driver turns the vehicle off and removes the key, the ignition off command is applied and the signals can be checked. At last, the sleep command is applied and the system goes to the sleep mode.

4.1. Report of the Test Case

The applied test script is saved and it is possible to analyse each message and signal used in the testing procedure. Figure 19, a screenshot of the test log provided by the tool, includes all steps and commands applied on the system under test. The wakeup command is applied as the first command to the system and all predefined macro commands are performed. A delay is predicted to have a realistic approximation for the time between entering the driver inside the vehicle and pushing the ignition key. The “Ignition On” command applies all needed conditions when the ignition key is pushed. By considering a delay for this step, the display is turned on, therefore, the webcam can capture a screenshot to record that moment to check if the display answers these commands as expected. Furthermore, there is a specific signal to check the performance of the display unit and understand if it communicates with the radio unit. To validate the system precisely, a signal checking command is put in the test script to find out if the connection is fine because this communication is crucial in the project scope. Then, the cranking and engine related commands are applied to the system under test. A reading signal command is also written in the test script to read a specific signal value and validate the radio status by checking if this signal value is as expected in this step. The tool, later, applies the commands related to simulating the ignition off condition. There is a delay approximating the time between applying the ignition off and the sleep mode of the system. To validate the captured screenshots during the test procedure, a pattern recognition and validation is used to check if the system status were as they had to be.

```
2021/02/10 11:56:08 AM: [TEST]00375_infotainment
2021/02/10 11:56:08 AM: Wake_Up()-NetOn()
2021/02/10 11:56:09 AM: Wake_Up()-Write_Signal(BCAN1;NWM_BCM;ActiveLoadMaster;1)
2021/02/10 11:56:10 AM: Wake_Up()-Write_Signal(BCAN1;NWM_BCM;SystemCommand;0)
2021/02/10 11:56:15 AM: Wait_Repetition(5000)
2021/02/10 11:56:15 AM: Ign_On()-Write_Signal(BCAN1;STATUS_BH_BCM1;OperationalModeSts;4)
2021/02/10 11:56:15 AM: Ign_On()-Write_Signal(BCAN1;STATUS_BH_BCM2;CmdIgnSts;4)
2021/02/10 11:56:15 AM: Ign_On()-Write_Signal(BCAN1;STATUS_CCAN3;EngineSts;0)
2021/02/10 11:56:16 AM: Ign_On()-Write_Signal(BCAN1;NWM_BCM;SystemCommand;2)
2021/02/10 11:56:16 AM: Ign_On()-Write_Signal(FDCAN1;BCM_FD_2;KeyInIgnSts;2)
2021/02/10 11:56:16 AM: Ign_On()-Write_Signal(FDCAN1;BCM_FD_2;OperationalModeSts;4)
2021/02/10 11:56:17 AM: Ign_On()-Write_Signal(FDCAN1;BCM_FD_10;CmdIgnSts;4)
2021/02/10 11:56:19 AM: Wait_Repetition(2000)
2021/02/10 11:56:19 AM: Capture_ScreenShot(Logitech Webcam C930e#vid_046d&pid_0843&mi_00#6&2475419f&0&0000#;GeoLocalizationPopUp) the camera works at 30.00 fps
2021/02/10 11:56:19 AM: Check_Signal_Value(BCAN1;STATUS_TELEMATIC[0x5BE];PowerSts_Telematic;4) -> Passed -> BCAN1.STATUS_TELEMATIC[0x5BE].PowerSts_Telematic=4
2021/02/10 11:56:35 AM: User_Request(Click on Accept Button) -> User pushed OK.
2021/02/10 11:56:35 AM: Capture_ScreenShot(Logitech Webcam C930e#vid_046d&pid_0843&mi_00#6&2475419f&0&0000#;MenuPage) the camera works at 30.00 fps
2021/02/10 11:56:35 AM: Ign_Cranking()-Write_Signal(BCAN1;STATUS_BH_BCM1;OperationalModeSts;6)
2021/02/10 11:56:36 AM: Ign_Cranking()-Write_Signal(BCAN1;STATUS_BH_BCM1;OperationalModeSts;7)
2021/02/10 11:56:36 AM: Ign_Cranking()-Write_Signal(BCAN1;STATUS_BH_BCM2;CmdIgnSts;5)
2021/02/10 11:56:36 AM: Ign_Cranking()-Write_Signal(BCAN1;STATUS_CCAN3;EngineSts;1)
2021/02/10 11:56:36 AM: Ign_Cranking()-Write_Signal(FDCAN1;BCM_FD_2;OperationalModeSts;6)
2021/02/10 11:56:36 AM: Ign_Cranking()-Write_Signal(FDCAN1;BCM_FD_2;OperationalModeSts;7)
2021/02/10 11:56:37 AM: Ign_Cranking()-Write_Signal(FDCAN1;BCM_FD_10;CmdIgnSts;5)
2021/02/10 11:56:37 AM: Engine_On()-Write_Signal(BCAN1;STATUS_BH_BCM1;OperationalModeSts;8)
2021/02/10 11:56:37 AM: Engine_On()-Write_Signal(BCAN1;STATUS_BH_BCM2;CmdIgnSts;4)
2021/02/10 11:56:38 AM: Engine_On()-Write_Signal(BCAN1;STATUS_CCAN3;EngineSts;2)
2021/02/10 11:56:38 AM: Engine_On()-Write_Signal(FDCAN1;BCM_FD_2;OperationalModeSts;8)
2021/02/10 11:56:38 AM: Engine_On()-Write_Signal(FDCAN1;BCM_FD_10;CmdIgnSts;4)
2021/02/10 11:56:38 AM: Read_Signal(BCAN1;STATUS_TELEMATIC[0x5BE];PowerSts_Telematic) -> BCAN1.STATUS_TELEMATIC[0x5BE].PowerSts_Telematic=4
2021/02/10 11:56:38 AM: Ign_Off()-Write_Signal(BCAN1;STATUS_BH_BCM1;OperationalModeSts;2)
2021/02/10 11:56:39 AM: Ign_Off()-Write_Signal(BCAN1;STATUS_BH_BCM2;CmdIgnSts;1)
2021/02/10 11:56:39 AM: Ign_Off()-Write_Signal(BCAN1;STATUS_CCAN3;EngineSts;0)
2021/02/10 11:56:39 AM: Ign_Off()-Write_Signal(FDCAN1;BCM_FD_2;OperationalModeSts;2)
2021/02/10 11:56:39 AM: Ign_Off()-Write_Signal(FDCAN1;BCM_FD_10;CmdIgnSts;1)
2021/02/10 11:56:41 AM: Wait_Repetition(2000)
2021/02/10 11:56:42 AM: Sleep()-Write_Signal(BCAN1;NWM_BCM;ActiveLoadMaster;1)
2021/02/10 11:56:43 AM: Sleep()-Write_Signal(BCAN1;NWM_BCM;SystemCommand;3)
2021/02/10 11:56:43 AM: Sleep()-NetOff()
2021/02/10 11:56:43 AM: Match_Pattern_vs_ScreenShot(M182\MaseratiPattern_X357_Y108_W279_H62_R100.png;GeoLocalizationPopUp;false;0.820;Passed;false) -> Matching Pattern Passed
2021/02/10 11:56:43 AM: Match_Pattern_vs_ScreenShot(M182\HomeButtonMenuPage_X27_Y572_W91_H72_R100.png;MenuPage;false;0.820;Passed;false) -> Matching Pattern Passed
2021/02/10 11:56:19 AM: Wait_Repetition(60000)
```

Figure 19 - Test Log provided by the tool

One of the captured screenshots during this test is the in ignition on mode as shown in figure 20. It is the first pop-up the driver can see in the display when the ignition key is pushed to put the vehicle in ignition on status.

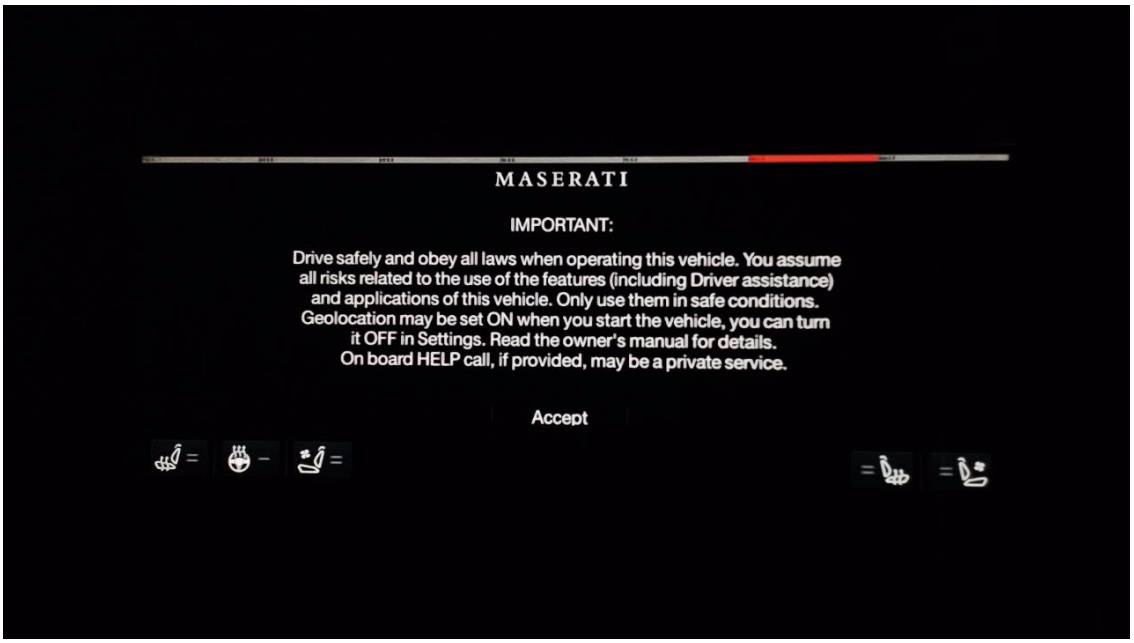


Figure 20 – Screenshot from the first pop-up taken by the webcam

After the first pop-up, the display shows the main menu containing all features provided by the vehicle as figure 21. This pages stays as the front page unless the driver or the passenger touches a button to go through another provided feature in the display of the infotainment system of the car.

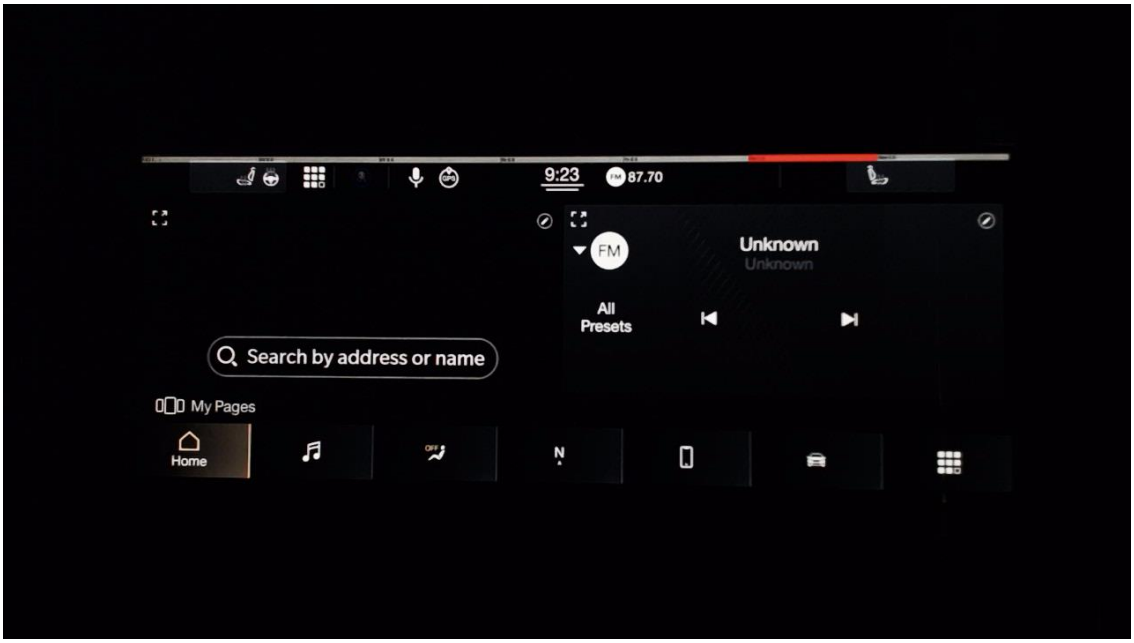


Figure 21 – Screenshot from the display menu taken by the webcam

Having these screenshots captured by the written commands in the test script, the pattern recognition commands validate them by recognizing the predefined patterns inside those screenshots. The first screenshot is the first pop-up in which there is a text and Maserati brand. Therefore, the test engineer had defined the brand as a pattern before starting the test by knowing the display statuses. Pattern recognition validation is applied as shown in figure 22 for the first screenshot.

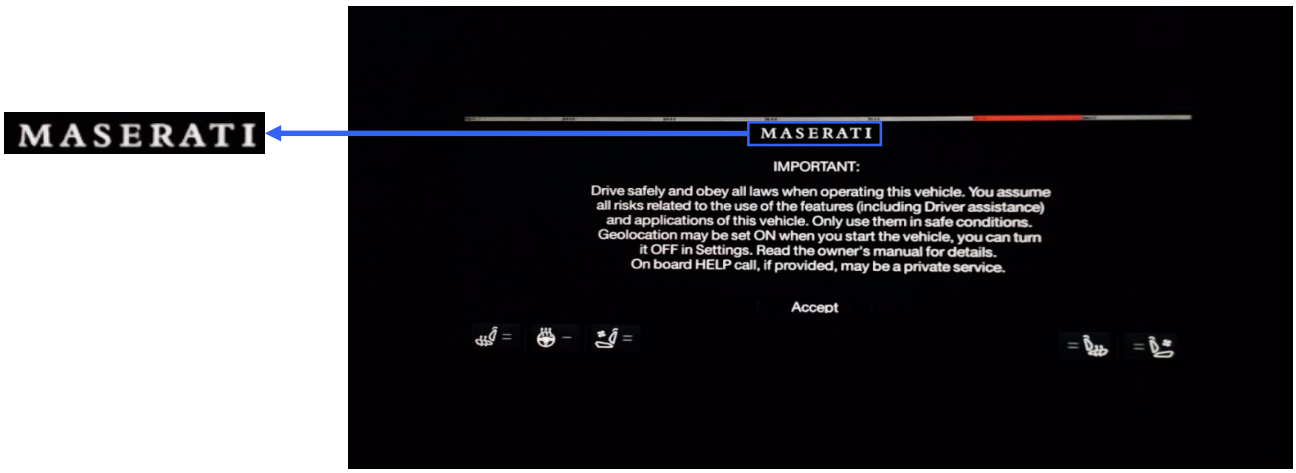


Figure 22 – Maserati Pattern recognition to validate the screenshot and the status of the subsystem

The second Pattern recognition is applied in the main menu screenshot. The main menu is shown in the normal use of the vehicle until the driver changes it or touches it. Therefore, according to figure 23, the menu button is another predefined pattern by the test engineer to validate if the display has passed the first pop-up to the normal operation of the display.

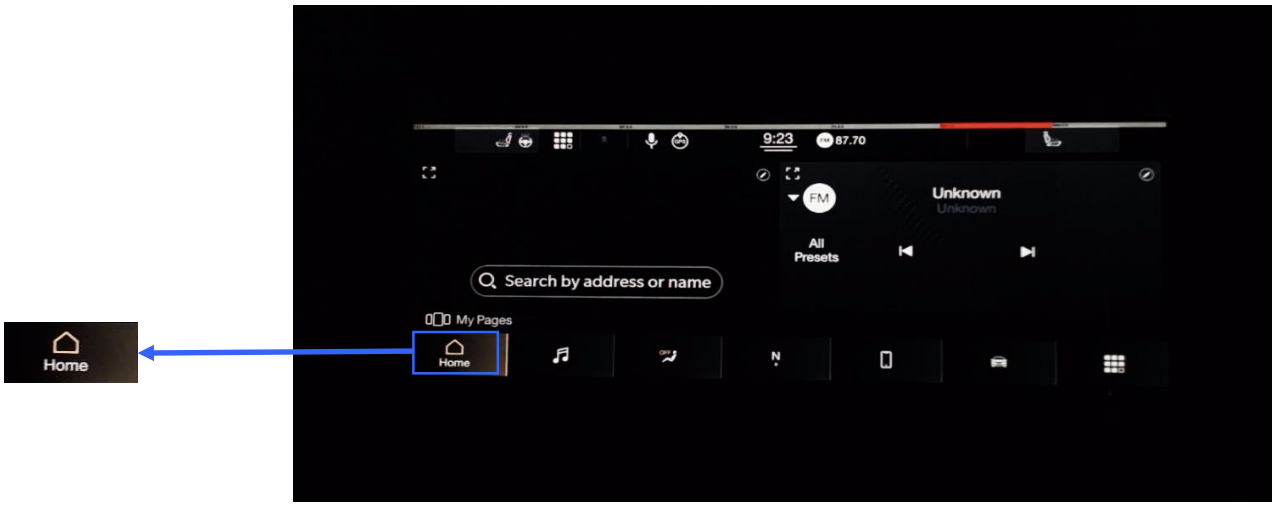


Figure 23 – Menu button Pattern recognition to validate the screenshot and the status of the subsystem

4.2. Discussion

According to the applied tests and checking the functionality of the tool for the infotainment system, it is possible to check CAN and CAN FD network and the regarding messages to check if the ECU communicates well following the specifications. The validation team is now able to use ATBEditor to automatically test and validate the systems by defining the desired test case and related macro.

4.3. Result Analysis

The obtained results exhibit that the radio transmits and receives messages as planned with the ECUs and the automated testing was successful. The display unit is used as an example of ECU which should receive some information and show it to the driver and passengers inside the vehicle.

During the simulation, the whole network topology must be analysed once the test procedure is planned. Every hardware ECU inserted in the device-under-test needs to communicate properly to avoid malfunction of the testing procedure. Basically, the ECUs receive an initial value as the

Using ATBEditor, it is possible to test if the radio head unit communicates fine by CAN and CAN FD networks according to the network topology. This tool can test all subsystems of the vehicle based on CAN and CAN FD if the macro commands are defined based on the related messages and signals available in the DBC, CAN log of that system and the needed test case.

4.4. Future work

As this project is applied for the research and design section of a new car line which will be produced a few years later, it is possible to implement the automated test bench with similar strategies for another ECUs, protocols, and also the whole connectivity system. It will help the validation team test most of the features automatically and concentrate the human resource on the features which could be tested only by the human. Furthermore, it is possible to apply this strategy on different car lines and models in order to facilitate the testing approach in many models produced by Maserati.

CONCLUSION

According to the scope of the project, the previous version of the internal tool ATBEditor is upgraded and modified to make it able to perform the automated testing for CAN and CAN FD network simultaneously thanks to changing and upgrading the engine code and the graphical user interface. Furthermore, the macro list is also updated to include the macro commands of both networks focusing on the vehicle functions and network topology. The target system in this scope is the Infotainment system of the new generation Maserati vehicle, MC20, however, the engine code is able to work with all subsystems and the only change is related to defining the corresponding macro list. As well as having a comprehensive result inside the tool, the detailed result of this automated test is saved in the directory for deeper analysis. The new version is released and applied in Maserati carline as well as Stellantis carline.

REFERENCES

[1] https://www.maserati.com/international/en/models/mc20

[2] Bosch GmbH, “CAN with Flexible Data-Rate - CAN in Automation (CiA)”, Specification Version 1.0 (released April 17th, 2012

[3] Florian Hartwich, Robert Bosch GmbH, “Bit Time Requirements for CAN FD”, 14th international CAN Conference (iCC), 2013

[4] https://www.vector.com/int/en/know-how/technologies/networks/can

[5] Karl Henrik Johansson, Martin T¨orngren, Lars Nielsen, “Vehicle Applications of Controller Area Network”, Handbook of Networked and Embedded Control Systems: Springer, pp 741-765

[6] https://www.kvaser.com/wp-content/uploads/2016/10/comparing-can-fd-with-classical-can.pdf

[7] Uwe Koppe, Johann Tiderko, MicroControl GmbH & Co. KG, “CAN driver API - migration from Classical CAN to CAN FD”, 16th international CAN Conference (iCC), 2017

[8] Orlando Esparza, Wilhelm Leichtfried, Fernando González, Microchip Technology Inc., “Transitioning applications from CAN 2.0 to CAN FD”, 15th international CAN Conference (iCC), 2015

[9] https://www.earth2.digital/blog/what-is-vehicle-can-bus-ecu-evoque-adam-ali.html

[10] SAE. (2013). Data Base Container (DBC) file format (Standard No. J1939). Retrieved from https://www.sae.org/publications/collections/content/j1939_dl