FEDERAL UNIVERSITY OF RIO GRANDE DO SUL
ENGINEERING SCHOOL
CONTROL AND AUTOMATION ENGINEERING

GUILHERME RAABE ABITANTE - 00243701

AUTOMATED TEST SYSTEM FOR VEHICULAR CONNECTIVITY SOLUTIONS

Porto Alegre
2021

GUILHERME RAABE ABITANTE - 00243701

AUTOMATED TEST SYSTEM FOR VEHICULAR CONNECTIVITY SOLUTIONS

Graduation Project presented to COMGRAD-CCA of Federal University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of *Control and Automation Engineering*.

ADVISOR:

Prof. Marcelo Götz

Porto Alegre
2021

GUILHERME RAABE ABITANTE - 00243701

AUTOMATED TEST SYSTEM FOR VEHICULAR CONNECTIVITY SOLUTIONS

This Project was considered adequate for obtaining the credits of the course TCC (Diplom Project) of *Control and Automation Engineering* and approved in its final form by the Advisor and the Examination Committee.

Advisor: ______________________________
Prof. Marcelo Götz, UFRGS
Doctor by Universität Paderborn, UPB, Germany

Examination Committee:

Prof. Marcelo Götz, UFRGS
Doctor by Universität Paderborn, UPB, Germany

Prof. Ivan Müller, UFRGS
Doctor by Universidade Federal do Rio Grande do Sul, RS, Brazil

Prof. Renato Ventura Bayan Henriques, UFRGS
Doctor by Universidade Federal de Minas Gerais, MG, Brazil

____
Marcelo Götz
Course Coordinator
Control and Automation Engineering

Porto Alegre, May 2021.

DEDICATION

This work is dedicated to my parents, as they gave their support through all my graduation and during the making of this work. Additionally, to Mauro Zanella and Philipp Hohl, without whom I'd never have had the opportunity to develop it.

ACKNOWLEDGMENTS

To Jan Kaczmarek for giving me pointers on CANoe and wired protocols, to Georgios Charalampopoulos for introducing me to the wonders of V2X and to the whole XAXV3 team for being receptive and helpful throughout the project, I thank you.

ABSTRACT

Verification and testing are essential processes in the development of a product. They not only identify issues, but also assure that the product meets the desired behavior. The present work describes a test system to test an embedded vehicular connectivity unit which has several interfaces, including CAN/CAN-FD, RS232, Bluetooth Low Energy, Wi-Fi, and V2X protocols. The objective of the work is to ease debug of the device under test, testing different features and making the whole validation process more agile for new test cases and less time consuming for routine operations. The test system consists of a software coded in Python that runs in the device under test and a modular test fixtures available in the market. The latter was programmed during this work in a proprietary language, CAPL. The developed test system successfully includes all features initially proposed and was made to be further enhanced. Naturally, new requirements were added as the work progressed. Generating new objectives is essential to keep up the continuous pursuit for excellence and fundamental to the system outside of the scope of this bachelors thesis. So far, the intuitive and friendly interface, the correct management of the system resources, and the implementation of tests, makes the Python software and test fixtures, both combined and independently, reach beyond initial expectations.

Keywords: Vehicular connectivity, protocol validation, test automation.

RESUMO

Verificação e testes são processos essenciais no desenvolvimento de um produto. Não só identificam problemas, mas também asseguram que o produto satisfaz o comportamento desejado. O presente trabalho cria um sistema de testes para um sistema embarcado de conectividade veicular que tem várias interfaces, incluindo os protocolos CAN/CAN-FD, RS232, Bluetooth Low Energy, Wi-Fi e V2X. O objetivo do trabalho é facilitar a depuração do dispositivo, testando diferentes características e tornando todo o processo de validação mais ágil para novos procedimentos de teste e menos demorado em operações rotineiras. O sistema de teste consiste em um software codificado em Python executado no dispositivo e uma jiga de testes modular disponível comercialmente. Este último foi programado durante este trabalho numa linguagem proprietária, CAPL. O sistema de teste desenvolvido inclui, com sucesso, todas as características inicialmente propostas e foi elaborado prevendo futuras melhorias. Naturalmente, novos requisitos foram acrescentados à medida que o trabalho avançava. A geração de novos objetivos é essencial para manter a busca contínua por excelência e fundamental para o sistema fora do âmbito deste trabalho de conclusão de curso. Até agora, a interface intuitiva e amigável, a gestão correta dos recursos do sistema e a implementação de testes, torna o software Python e a jiga de testes, tanto combinados como independentes, acima das expectativas iniciais.

Palavras-chave: Validação, Sistema de Testes, Conectividade Veicular.

CONTENTS

LIST OF FIGURES 15

LIST OF ABBREVIATIONS 17

1 INTRODUCTION 19

2 LITERATURE REVIEW 21

2.1 The Vector Validation Solution 21

2.2 CANoe and CAPL Overview 22

2.3 Python and Generic Coding 22

2.4 V2X Communication 23

3 DEVELOPMENT 25

3.1 Mini Test System 26

3.1.1 Testing CAN 28

3.1.2 Power Supply 29

3.2 Qualification Software 31

3.2.1 Template Scripts 31

3.2.2 QSW Modules 32

3.2.3 GPIO 35

3.3 V2X 35

3.3.1 Intersection Controllers 37

3.3.2 Vehicle Simulation 37

3.3.3 Special Nodes 39

3.3.4 Virtual V2X Environment Interface 40

4 RESULTS 43

5 CONCLUSION 45

APPENDIX A - QSW EXAMPLE USAGE AND COMMANDS 47

APPENDIX B - MTS BEFORE THE QSW 53

APPENDIX C - BLUETOOTH DONGLES WITH THE QSW 55

APPENDIX D - MTS V2X INTERSECTION *.CAN* FILE 57

ANNEX A - FIRST ANNEX TITLE 61

A.1 DENM Cause Codes 61

REFERENCES 63

LIST OF FIGURES

1 ProCV Gen2 - Vehicular Connectivity Unit 19
2 System overview 20
3 Current European V2X standards 24
4 Mini Test System (MTS) with ProCV Gen1 connected 25
5 MTS Simulation Setup 27
6 MTS Measurement Setup 27
7 MTS CAN test interface 28
8 MTS Power Supply test interface 30
9 QSW Python Scripts 34
10 MTS V2X node architecture 36
11 MTS V2X simulated car close-up 39
12 MTS V2X input interface 40
13 MTS V2X message trace 41
14 MTS V2X simulated intersection close-up 42
15 QSW Server raw help text 47
16 QSW Client in commander mode listing the available managers and CAN0 scripts 48
17 QSW Client in commander mode executing *help* 49
18 QSW Client in supervisor mode 50
19 QSW Server executing the start up sequence 51
20 QSW Server executing the stop sequence 52
21 MTS CAN0 sub-network 54
22 *mgmtBT* raw *help* code 56

LIST OF ABBREVIATIONS

DUT	Device Under Test
QSW	Qualification Software
MTS	Mini Test System
V2X	Vehicle-to-everything
GPIO	General Purpose Input Output
IMU	Inertial Measurement Unit
BT	Bluetooth
BLE	Bluetooth Low Energy
DENM	Decentralized Environment Notification Message
CAM	Cooperative Awareness Message
SPaT	Signal Phase and Timing

1 INTRODUCTION

The development process of a product is usually accompanied by constant testing and verification, not only to promptly identify and solve issues in the work, but also to ensure that the product meets the desired behavior. This continuous checking consists of different tasks and may use different strategies. The objective of the present work is to create a system that automatically tests simple features of the target product, identifying commonly occurring problems. It allows professionals who perform these tests to focus on more detailed approaches and to broaden the scope of the whole validation procedure.

The Device Under Test (DUT) is a vehicular connectivity unit with several communication interfaces (FRIEDRICHSHAFEN, n.d.). It was developed to be a telematics central, exchanging information with different peripherals, communicating with external devices, and interfacing with the internet. It operates with a Linux kernel and a firmware developed within the company. The test routines are compatible with different combinations of firmware and hardware versions. The second hardware version may be seen in Figure 1. This device is still under development, by ZF Friedrichshafen AG, and the test system aims to improve the workflow of the validation team assigned to this DUT of this company.

Figure 1: *ProCV Gen2 - Vehicular Connectivity Unit*

Source: Author

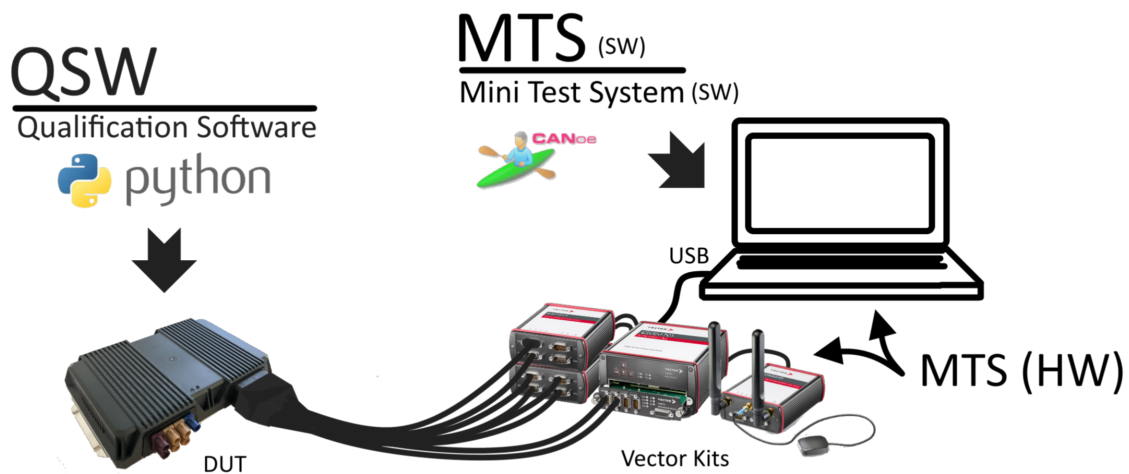
As part of this work, a software that runs on the DUT, the Qualification Software (QSW), was developed from scratch. It manages test procedures, executes internal tests, collects system data for analysis, and logs procedures. The QSW is primarily programmed in Python, but also makes use of libraries coded in C++ for low level access and interface

manipulation. Although most of this software was developed by me, there were instances of paired programming. These are specified in Subsection 3.2.2.

Most of the DUT interfaces are tested by communicating with an external system specifically designed to interact with the DUT, the Mini Test System (MTS). This system may also perform independent tests on DUT features. The MTS hardware selection is not part of the scope of this work, being briefly explained in Section 2.1. The software that operates this test system is called CANoe. This work includes the development of program blocks and interfaces that run inside CANoe to test the DUT according to its requirements, which was done entirely by me.

The predefined hardware of the MTS is a collection of validation kits developed by Vector Venture Capital GmbH, centralized in a server module by the same manufacturer. It communicates through USB with a computer where an instance of the main software that controls the whole Vector system, CANoe, runs. In an attempt to clarify the boundaries of QSW and to provide visual description of the MTS, Figure 2 depicts the interaction of CANoe with the computer that runs it, its interface to Vector Kits, the connection of Vector Kits to the DUT (which may happen in different protocols), and finally the loading of QSW into the DUT.

Figure 2: *System overview*



Source: ZF Friedrichshafen A.G.

2 LITERATURE REVIEW

The first section of the Literature Review (Section 2.1) describes the MTS hardware, protocol configuration, and how to setup the CANoe environment. Although the last two are topics are specific to the DUT and included in the scope of this work, they were done by following a manual given by the manufacturer. Therefore, they are not described in the Implementation chapter, instead, a generic configuration procedure is explained here.

All algorithms developed specifically to test the DUT with the MTS are executed by CANoe and were coded in a proprietary language called CAPL. Section 2.2 describes how it handles protocol interfaces and structures the system. The section also briefs on the language and explains how tests are performed in a Vector environment.

In the context of object oriented programming, class inheritance is a common practice that aids not only in organizing the project and keeping code generic, but also eases most debug procedures. As the QSW uses inheritance in its core, the Section 2.3 gives an overview on how this is dealt in Python.

Section 2.4 gives an overview of Vehicle-to-everything (V2X) communication, giving examples of the most common message types and what kind of information is exchanged. In real use cases, the DUT would be included in a V2X environment and assigned a specific role.

2.1 The Vector Validation Solution

The Vector solution is designed for analysis and testing of DUTs, aiding from development planning to system-level test. A validation setup has a CANoe instance at its center, commanding a number of connected Vector Kits. These pieces of hardware have distinct capabilities and multiple protocol interfaces, supporting different protocol versions depending upon the model. When testing a DUT with multiple interfaces simultaneously, communication is usually centralized in a single "server" kit, so that the computer where CANoe is running has to connect with only one hardware. This approach not only minimizes cabling to the computer to a single USB, but also allows the Vector Kits to communicate internally, to automatically synchronize systems, and to combine a single timestamp among all test interfaces.

Vector products are event driven, which means that all information exchanged internally is carried by events, including messages sent between interfaces and configuration done at code level. As events may be queued or delayed for some reason, they are all loaded with a timestamp when created, guaranteeing proper tracing for most of algorithms.

Once the test system is physically interconnected, the hardware must be mapped in software. All settings and all used algorithms are saved into a single file, called a "CANoe configuration". As part of this configuration, the real hardware is mapped to a

virtual network, elevating the abstraction level and facilitating any test system hardware enhancements. The virtual network interfaces may be assigned individually to a specific port of the real hardware. The created virtual interfaces are all connected to the same virtual network, but they are grouped by protocol into sub-networks, as a way to organizing the test system.

Hardware can be directly configured prior to test execution through the CANoe interface. During test execution, there is the possibility to trigger events that configure the virtual interfaces. These take precedence and will override hardware settings. Depending on the interface type, different settings are available, such as data speed, resistor terminations, checksum, protocol version, master-slave role, and others.

2.2 CANoe and CAPL Overview

In CANoe, any event within the virtual network may be processed by a node, which may also generate events on its own. Nodes are independent code blocks that run in a specific sub-network. Each node executes its algorithm in parallel and may have an associated interface, with which the user can interact while the test is in execution. There is no event routing within the virtual network, which means that it's possible to configure any node to react to any event, even if they belong to different sub-networks.

CANoe offers standard nodes that do not require additional programming beyond some initial setup. Standard nodes execute simple tasks like tracing messages of a given type or generating user configured messages by the click of a button or by a set period. More complex nodes, adapted to the target project, are added just like an ordinary node, but must be programmed by a developer using CAPL, a C based proprietary language that is event driven just as CANoe.

The main program file of a developed node is a .can file. This file may include a number of additional .cin files, which act as function libraries for the node. These included files may include additional .cin files, cascading inclusions, allowing the developer to avoid code repetition, and increasing project flexibility. The events handled in these nodes range from an incoming message to a modification in a system variable. The latter is an information defined in the CANoe context which is used mainly to store information and sync the nodes with either other nodes or their interfaces.

The common approach in programming interfaces is to create a system variable and then associate its value with a configurable field in a GUI. If a programmed node requires an interface, it must also be designed by the developer. CANoe offers a tool for quickly creating a GUI and associating its buttons and fields with a system variable.

2.3 Python and Generic Coding

Python is an interpreted high-level programming language that has been consistently gaining popularity in the last decade (GUTTAG, 2016). The Python community encourages programmers to properly create documentation and to follow best practices, contributing to conventions universally accepted across applications. Python Enhancement Proposals (ROSSUM; WARSAW; COGHLAN, 2001) serve as reference documentation for the community and as a channel for describing the rationale for new features in the programming language.

Python is dynamically-typed, garbage-collected that contains many options for any programmer. Although inside the interpreter Python uses an object oriented structure,

the scripts can also be coded using other programming paradigms, including structured and functional programming. Currently Python offers a wide range of libraries and open source projects that can be easily included in a project to ease development. The number of available libraries continue to expand as a result of the incentive to using modules. Modules allow for sharing open-source libraries among the members of the Python community, making a cumulative collection of reusable and distributable code.

Python does not natively use interfaces, but to ensure that a certain class has the required set of methods to operate, class inheritance can be used. The child class will not just have all the methods and properties of the parent, but it may overwrite any method or property of the parent with new instructions or objects. This could be used in a similar way as an interface if, for example, the parent class is programmed with a method called *connect* that raises an exception when used. The *connect* method could only be used if the child class overwrites it with specific code.

2.4 V2X Communication

Vehicle-to-everything (V2X) is a class of communication that includes any transmission from or to a vehicle. The DUT communicates through the IEEE 802.11p standard using European message types (INSTITUTE, n.d.). The standard gives much freedom to message structure and is generic in the higher layers of the protocol. Therefore, complementary standards were used to further specify how the DUT implements V2X.

When using the IEEE 802.11p standard in Europe, it is possible to select different communication channels and use personalized message types. Still, just the four most used types are tested with the tools developed in this work. These messages are used by a vehicle to communicate to other vehicles or road infrastructure. The message types described below are carried by a Geo-Networking layer that contains the location of the transmitter, its address and its size. If the Geo-Networking message does not carry any content, it is interpreted as a beacon. How the V2X standards are structured in Europe can be seen in Figure 3.

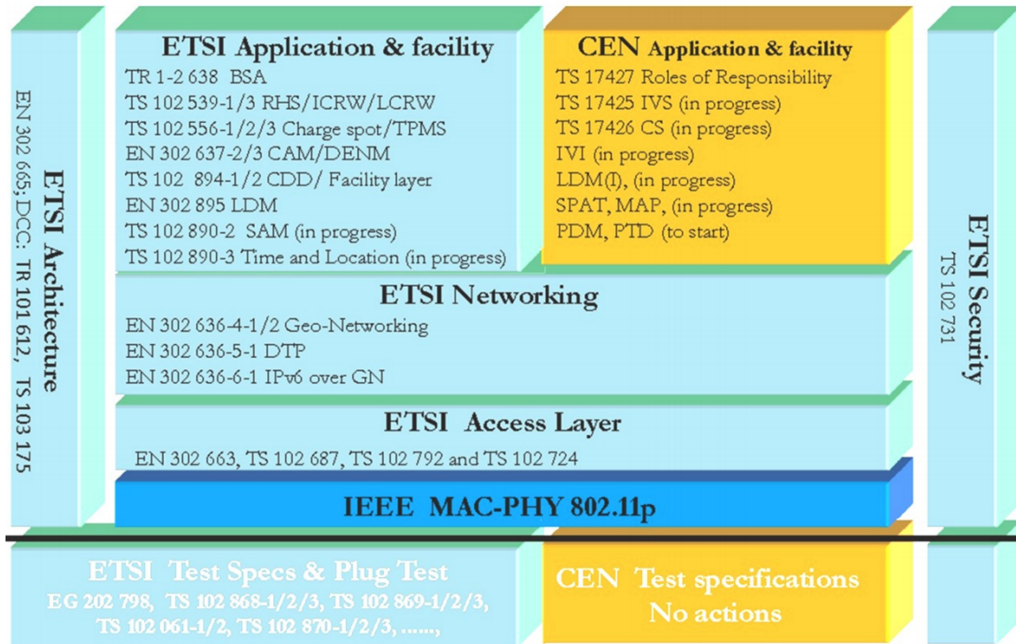
A Cooperative Awareness Message (CAM) is used mainly to broadcast vehicle actions, such as lane changes, future trajectory, and blind spot detection (ETSI, 2019). Yet, they can also be used to forward any type of data within V2X stations, allowing for complex platooning operations. These messages may also contain any type of vehicle information, informing from the hazardous cargo to the steering wheel angle.

The road infrastructure transmits MAP messages to vehicles, defining topological aspects of lanes, intersections, and road segments (ISO, 2019). MAP messages also specify the relationships of all circumstances of the road, allowing vehicles to safely map the traveling environment.

SPaT stands for Signal Phase and Timing, which is a message type that broadcasts cyclical behaviors, like the status of traffic lights or barriers. They are used not only to organize traffic when platooning, but also in vehicle control systems to predict if the car should be slowing down to an incoming change in the traffic signal or if it will have time to safely cross an intersection.

Decentralized Environmental Notification Messages (DENM) are triggered to provide information about a specific driving environment event or traffic event to other stations. The events can be of any kind, as long as it has an impact in traffic. Road construction, for example, could have a V2X capable road side unit informing the restricted areas and a new speed limit while the repairs are ongoing. Another possible usage is to have an ambulance

Figure 3: *Current European V2X standards*



Source: SWARCO, Amsterdam Intertraffic 2014

in route to a hospital transmit DENM messages to inform the special circumstances and ask for right of way.

3 DEVELOPMENT

The QSW and MTS were developed to be counterparts, nonetheless they may independently execute simple routines. The DUT interface test coverage and the test cases are discussed in a separate section for each counterpart. These do not go in depth in the actual test procedures, as that would require lengthy descriptions. Instead, few modules are selected as examples to illustrate generic operation, with emphasis in describing the overall test system structure. The MTS fully connected to a DUT running an instance of the QSW may be seen in Figure 4.

Figure 4: *Mini Test System (MTS) with ProCV Gen1 connected*

Source: ZF Friedrichshafen A.G.

Initially MTS and QSW were testing tools, rather than test case systems. As a result, the user had to create initial parameters based on test requirements and manually input them. After the implementation of this work, the system requires very little input from the user, yet an operator must still check if the test output matches what is expected. With that being said, the MTS and QSW were built to allow a higher level of automation, as test cases may be implemented at any time by the simple addition of a single script or node. Future developments of the QSW and MTS will include these features.

The system tests the following DUT wired interfaces: 2x CAN, 2x CAN-FD, 2x RS232, 1x K-Line, 1x LIN, and 2x Broad-Reach (Eth). The wireless protocols tested are V2X, Wi-Fi, GSM/Cellular, Bluetooth 4.0 / BLE, and GNSS. Further feature tests are done regarding the DUT's 4 analog inputs, 4 digital outputs, power supply, device memory, and CPU. As the product is still in development, current protocol versioning supported and additional interface specifications are classified. When using the QSW, some additional hardware is needed to fully use the system: a GPS antenna for GNSS, an access point for Wi-Fi, an available cellular network for GSM, among others.

3.1 Mini Test System

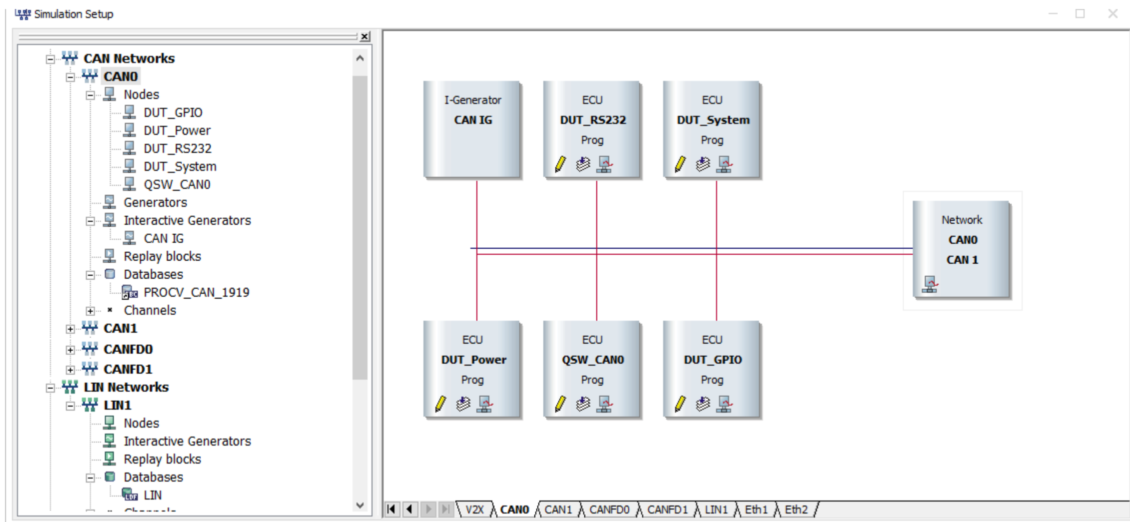
The CANoe configuration of the Mini Test System includes eight sub-networks: CAN0, CAN1, CAN-FD0, CAN-FD1, V2X, LIN1, Eth1 and Eth2. The MTS is also connected with the DUT through the GPIO and RS232 interfaces. These are not treated as virtual interfaces by CANoe, but as "sensors", due to the nature of the hardware used. Communication with sensors is handled in the MTS directly through system variables.

Having no associated virtual interface, the nodes that interact with RS232 and GPIO, as well as the ones that perform system and power tests, do not belong to a specific virtual sub-network. They were added to the CAN0 sub-network because every node must be in a network, but do not handle any CAN messages. Figure 5 shows the CAN0 setup and the project configuration tree-view. All nodes inside the "Nodes" category were programmed for this setup. Generators are also nodes, but are part of CANoe so are categorized differently. Databases are files that map the network, describing possible messages and their fields. Databases define communication parameters with the DUT.

Whenever the DUT communicates in a way that the MTS was not expecting, an error event is triggered. This event carries the message information and the error may be easily seen in CANoe's trace window, as the event would be marked in red. Failures range from wrong electrical levels at the physical layer to a format that does not conform to the database. As this checking is done automatically by CANoe, much can be said of the DUT simply by plugging it to the MTS and exchanging few messages. The databases used in the MTS were configured during this work, yet they are to be provided by the DUT's developers once the product enters its final stages as part of the test requirements. It is a concern of this work to complete, but not overreach, the given specifications.

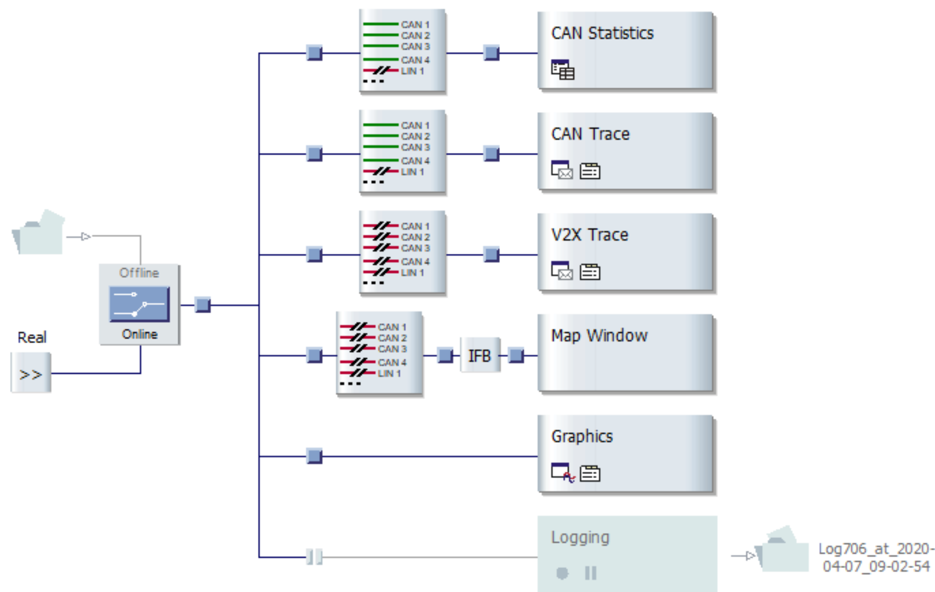
The measurement setup is a CANoe configuration that routes messages and events to specific blocks. The MTS measurement setup can be seen in Figure 6. The leftmost block selects between online and offline operation. Online means communication with the DUT and offline means reading a log file as an input. The offline option may be used to test specific scenarios with the MTS or emulate an interaction with the DUT, which is basically used to verify the test system. This block outputs the events that occur in the simulated network. The events are filtered by the blocks in the middle of the image, according to interface or origin (a node or the DUT), and then delivered to the rightmost blocks.

Figure 5: *MTS Simulation Setup*



Source: Author

Figure 6: *MTS Measurement Setup*

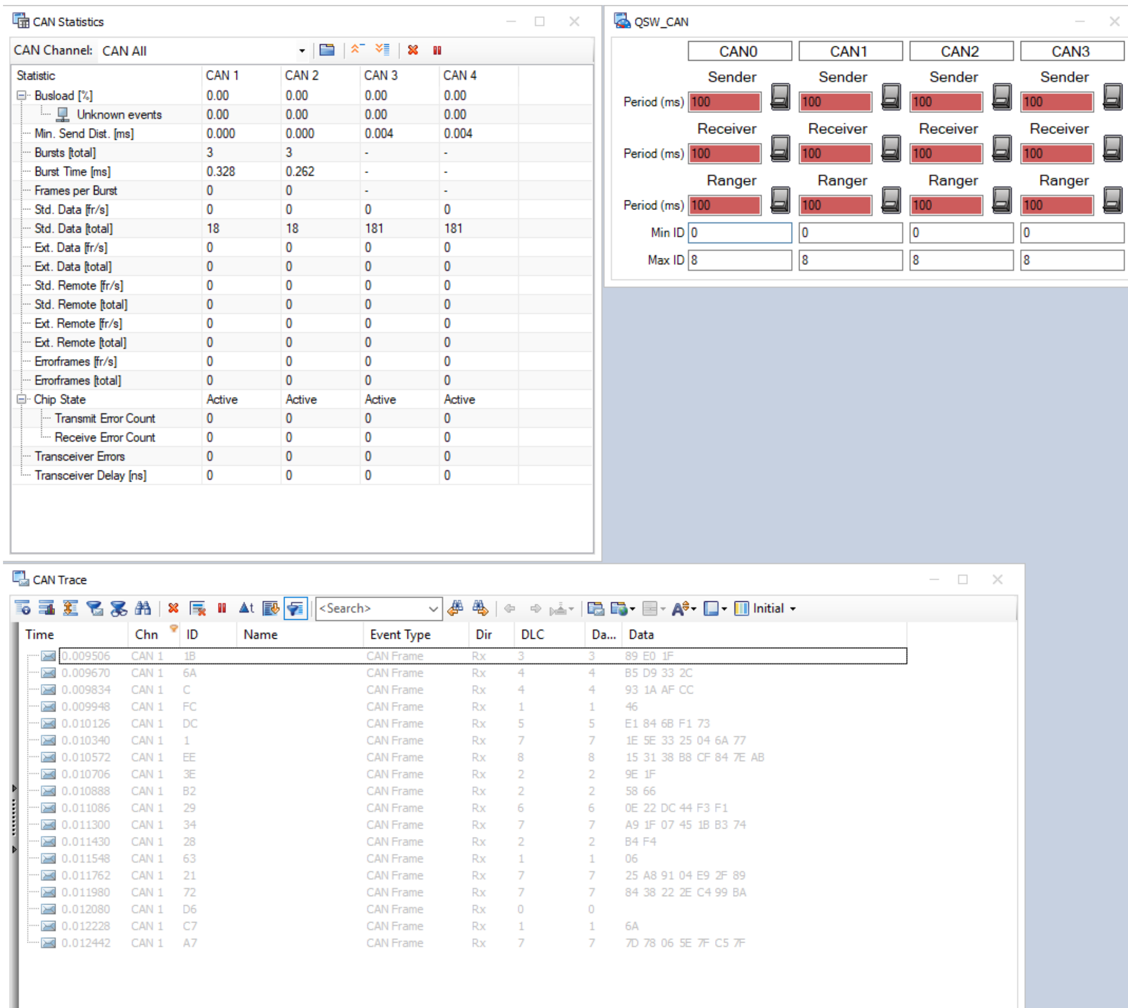


Source: Author

The Statistics and Trace blocks are further explained in the next subsections. The Logging block, disabled in Figure 6, generates a log file every time the test system runs. This logging is not only proof of the DUT's behavior, but may also be used by the MTS in offline mode to evaluate a scenario step by step. Logging is a standard feature applied in any official test procedure and is essential when an issue is encountered.

All blocks shown in Figure 6 were configured during this work, but are native to CANoe and required little adaptation. The nodes displayed in Figure 5, and all other sub-networks, were all programmed during this work, with the only exception being Generator nodes.

Figure 7: *MTS CAN test interface*



Source: Author

Generator nodes can trigger custom messages at demand, but only manually, so they are rarely used in the MTS.

Prior to implementing QSW, MTS controlled the DUT using SSH via RS232 interfaces. A node was developed to authenticate the session, trigger command at the push of a button, parse the command outputs, and gather system data automatically. Although well constructed and of extreme significance at the time, this feature was deprecated once the QSW was created, as it was no longer used for testing. Some of the nodes code was instead reused to pass commands to the QSW, in a higher abstraction level than before. The MTS still carries an interface for the user to input a string over the RS232 interface and receive its output. Appendix B gives and insight on how the system was at the time.

3.1.1 Testing CAN

The first to be implemented, CAN/CAN-FD tests incorporate a simple and effective interface, which will help explain the manual usage of the MTS. As seen in Figure 7, the interface consists of three windows used in parallel for testing.

The window used by the user to control the procedure is at the right hand side of the

image. There, the CAN interfaces are shown, each with its own configuration. When enabling the Sender, the corresponding interface will start sending messages with a random payload length and content. The message ID is also randomized, but must be within the range determined at the bottom of the window. The Ranger similarly sends messages, but instead the message IDs are incremented by one in every new message, looping back to the minimum ID configured once the maximum ID is reached. Whenever the Receiver is enabled, any messages with even IDs will be answered with a message with the next ID and the same payload.

The test approach with CAN follows the same guidelines as the QSW, introducing behaviors to the interface. The message types used by DUT are explicitly defined in the included database, but they are not yet fully defined. Once they are, the new database should be included in the CANoe configuration and allow more proper tests to be performed. In the meantime the behaviors introduced are sufficient for testing.

Whenever a DUT interface mismatches the configuration of the connected MTS interface, CANoe triggers an error event. The automatically generated error log also gives some insight on how the DUT is configured, allowing easy debug. Some other CANoe features are used in tests, mostly to perform stress tests. The introduction of message bursts in the bus is especially useful, which is used to check how the DUT behaves with high bus usage.

The statistics windows is seen at the left hand side of Figure 7. The data can be used to monitor the progression of any ongoing operations. The shown metrics can be used to quickly evaluate the DUTs state.

The message trace window, at the bottom of the interface, gives a detailed overview of what is happening at any given time and may be used to view the communication history. Filters may be applied and messages can be grouped chronologically, by message ID or other classifications. Any error frames appear in red, making it easy to track protocol breaches or wrong message formats. To further enhance error identification, handlers were programmed in the CAN nodes to alert whenever an incoming message is valid by the protocol, but unexpected by the test procedure.

Additionally, there is the Write Window, a window used by all CANoe nodes to write messages. In simpler terms, when a node uses a "print" command, the given string is shown in the Write Window and automatically logged.

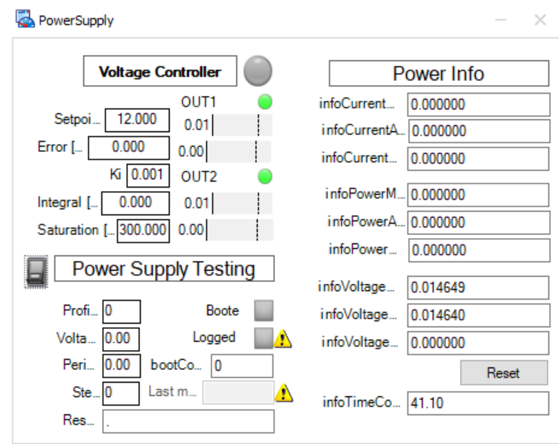
3.1.2 Power Supply

Power to the DUT can be supplied by the MTS, through a configurable internal power source. The tested device is intended to be a vehicular connectivity unit, so the input power includes common car battery voltages, ranging form 9V to a maximum of 36V. Power is regulated internally to 12V and is fed through three wires: OUT1 is the positive terminal to the device, OUT2 connects to the ignition of a real vehicle for signaling power on, and the third wire connects to ground.

Test with the MTS is done by defining a profile in the power supply CANoe interface. Profile 0 applies a square wave to the output with a period set by the user. Profile 1 ramps up the Voltage with a discrete number of steps. More profiles were to be added to the MTS, however development faced an insurmountable obstacle: the current hardware cannot change output voltage faster than $100\frac{V}{s}$.

The max Voltage rate is even lower when the DUT is plugged in the system. This limitation restricts tests that could be done with other profiles (e.g. applying a sinusoidal waveform), therefore development was halted until the MTS power supply module was upgraded. The implemented profiles cannot verify the device behavior in power surges, but

Figure 8: *MTS Power Supply test interface*



Source: Author

perform well for testing maximum and minimum operation voltages. This limitation also distorts the square wave set by Profile 0 into a series of trapezoids and the steps generated by Profile 1 into a series of ramps.

Aiming to follow the correct procedures, the output voltages of the MTS were measured and, once it was verified that the set Voltage differed from the real output, a control loop was implemented to compensate any errors. OUT1 is controlled by a proportional integrative loop, having the integral portion of the controller saturate at a defined level to avoid windup. As the error in OUT2 was similar to OUT1 and given that when following the implemented profiles both outputs are set to the same voltages, OUT2 is controlled in open loop that simply mimics OUT1.

The MTS Power Supply Test Interface may be seen in Figure 8 and through it the controller parameters may be set as desired. The current status of the power supply relays (green circles in the image) and levels of OUT1 and OUT2 may also be seen beside the controller settings. Metrics relevant to most test cases can be seen at the right side of the image and be reset manually through a button at any time. At the bottom left corner of the interface, the automatic test procedure may be configured and started. The right hand side displays whether the DUT has booted and whether the SSH connection could successfully login, followed by the number of times the DUT initiated a power-up sequence since testing began and the timestamp of the last received message in any of the RS232 interfaces. This information is used to implement test cases that generate a printed overview in CANoe once complete. This overview also registers a failure with a timestamp in case a power-down failure occurs during execution.

Unfortunately some interfaces experience a bug that makes the field names be covered by the fields, as seen in Figure 8. This happens when the computer makes a resolution change or the interface is resized. Currently the only workaround is to edit the interface and save it again using the new resolution and size.

3.2 Qualification Software

The Qualification Software (QSW), as mentioned before, is a collection of Python scripts that run in the DUT. Therefore it is interfaced with a command line terminal, same as most embedded systems based in a Linux kernel. Each script has a determined standing in the QSW, following a hierarchy. The top level of the system is a QSW Server with any number of QSW Clients connected. They communicate through internal sockets and use threading to execute tasks in parallel. Under the QSW Server, there are several Interface Managers, one for each interface type of the DUT.

The QSW Server imports and initializes all available Interface Managers at startup. Then it processes given commands to execute tests, start metric logging, parameterize interfaces, and other actions. Before executing any command received from a QSW Client, the QSW Server executes a command sequence defined in a text file. These commands are typically used to configure interfaces or automatically start logging, but could also include testing or additional configuration to be executed when the QSW Server starts.

Upon running an instance of QSW Client, the user is prompted to choose between two modes of operation, one for supervising and another for giving commands. The supervisor mode will periodically update display information on the selected interfaces of the DUT and QSW modules, monitoring the system while in operation. Although the QSW Client in supervisor mode has a fixed refresh rate of *100ms*, the information displayed is usually updated more slowly and asynchronously. The viewed data is primarily generated by the Managers and sent to the QSW Client, each one through a different thread. The Managers could be configured to update data as frequently as the QSW Client refreshes, however this would consume more DUT resources and compromise critical tasks, like maintaining stable communication across the DUT interfaces.

Commanding the QSW through a QSW Client command line was designed to be intuitive to experienced Linux developers, as the syntax is often similar. Whatever may be the QSW state, one could enter "help" to get a list of available commands. Managers, test procedures, and interface behavior configuration also have a detailed description which may be accessed through a "help" command. In all, more than 50 commands are included, some of them being explained in this section. A few usage examples and outputs from "help" commands may be seen in Appendix A. QSW Client connections in command mode may also be logged, including commands and their outputs, by giving the "logSimple" command at QSW Server start up. Command history is essential to determine if the state of the DUT metrics matches what is expected.

The main reason driving the development of the QSW to follow this design was the need to run the software in the background of the DUT operating system while being able to easily control and monitor tests. If the QSW Server were to be opened through a terminal and not be assigned as a background process, the server would terminate in the same instant the terminal closes, interrupting any ongoing tests and truncating metrics logs. With the current approach, the QSW Server may continue to run even if the DUT is left by itself. To make any changes, one would simply connect a computer to the DUT, open a terminal and run the QSW Client.

3.2.1 Template Scripts

The classes defined in template scripts do not become objects in the QSW, instead, they are used as base classes by other scripts. Like so, the templates define only common methods and instances, that all scripts of the respective type must have. A specific Interface

Manager is the top level script of its interface and usually incorporates a configuration script, some message core scripts, and more behavior scripts. Commands given to the manager configure and trigger these included scripts.

The Interface Manager template includes two classes. The first is a class for the manager itself, containing common methods for initializing threads, setting internal objects, and processing commands. This class also implements a standalone mode of operation, in which the Interface Manager can be started independently from the QSW with its own command line. Self-sufficient operation may be useful in stress tests, during which the DUT should not be burdened with maintaining a whole instance of the QSW.

The second standard class implements a supervisor that by default instantiates a thread and uses it to periodically update information. The data is available to other objects as long as those contain the supervisor as a parameter. All Interface Managers have one supervisor, collecting information about system resources, interface configuration, or messaging metrics. Each supervisor is responsible for a different set of information, in a way that even with all supervisors running in a QSW instance, no information is fetched twice.

Configurators are scripts that configure communication interfaces and setup the environment prior to communication. They may also be used to identify the current configuration. Generally the configuration scripts call native DUT commands through the Python *subprocess* library. Alternatively, configuration may be done through file descriptors serving as inputs to an interface API. Configurators may include different methods of configuration, however these are usually encapsulations of the same approach and are mainly included to test the approach itself, and not the configuration. For example, the Bluetooth Low Energy interface may be configured by transmitting the configuration directly to the protocol stack, by using the API, or by manipulating the protocol stack wrapper, but in essence, all of them are using the same internal bus to apply the configuration.

Aiming to conduct communication using different approaches, the QSW always exchanges messages using *cores*. These are low level scripts that exchange messages using imported Python packages, Linux commands, manipulating wrappers, internal buses, among others. All possible means of communication are to be tested, as the DUT's user may opt to use any of them. Cores are made to elevate the code abstraction level, in a way that, no matter the approach, the test script simply imports the desired core and uses *send* or *receive* methods.

As aforementioned, the Interface Managers do not implement formal test procedures, but instead apply behaviors to a desired interface. Behavior scripts may be enabled to, for example, answer a determined type of message with another, or periodically send messages following a predefined pattern.

3.2.2 QSW Modules

Several scripts were developed for the QSW, Figure 9 gives an overview while showing some information on them. The shape of each module shows which template they follow and all templates are at the bottom of the image. Blue colored scripts were entirely made by me, purple ones were made with the help of some team members, while white represents the ones not developed.

Although the managers were all implemented using the same template, their usage varies and many have unique features. While this section does not explain the managers commands in detail, Appendix A contains the raw information of the *help* command for some managers that could be used to better understand the managers capabilities. In the

next subsections, the managers are grouped and described by similarity.

Wired Protocols

As the DUTs firmware is still in development, most of the structure needed to test the BroadR-Reach (BR), LIN, and K-Line interfaces is not ready. Still their managers were implemented and can already be used to configure the interfaces. Although most of the protocol layers cannot be tested, QSW verifies that the protocol stacks are responsive and can exchange messages.

The situation is different for RS232 and CAN interfaces. Some additional features were added to the QSW, implementing the counterparts for the tests described in Section 3.1. Consequently, more elaborate standalone tests can be performed, by connecting interfaces of the same type and enabling different behaviors. The possible test cases could be determined by inspecting the scripts described in Figure 9.

The second hardware version of the DUT brought significant changes, affecting the RS232 interface. The elimination of a Redis service used to control the RS232 crippled *coreRS232redis*, the only core available that could be used to communicate through the RS232 interfaces. As a result, RS232 is not currently tested in the latest DUT hardware version. The SSH connection remained the same thus commanding the DUT through the MTS was preserved. Additional RS232 cores are being specified to solve this issue.

Wireless Protocols

The Wi-Fi and GSM managers work in a very similar way. Both of them use a stack implemented in the firmware to connect, disconnect, completely ignore a network, and configure the interfaces. Both of them do an automatic health check and maintain the device connected. It is possible to directly pass commands to each protocol stack, increasing greatly the usage of these managers and allowing flexible operation. Both of them can use an additional script to ping a selected IP address or URL over the internet, as well as check latency and packet loss.

mgmtGNSS is misleadingly named. First, it supports only GPS and GLONASS systems, so it is not an all-purpose GNSS module. Second, Inertial Measurement Unit (IMU) information comes from the same peripheral as GNSS data, so they were conveniently combined. This manager gathers data through either Redis or internal sockets and evaluates the collected information. Data consistency checks are done automatically, which alerts through the supervisor of any errors detected in the data. Additional scripts were implemented to measure the precision of the positioning system, offset, and IMU slip.

The Bluetooth (BT) section of the QSW is the most complex part of the QSW, as it includes extensive interaction with the protocol stack using the message bus system D-Bus (FREEDESKTOP, n.d.). Any application that uses Bluetooth will have to be developed by the user, since the firmware only includes the BT protocol stack. Therefore most of the code necessary to operate the Bluetooth interface was developed as part of the QSW.

All basic BT 4.2 features are tested by the QSW, including scanning for connections, pairing, usage of different BT services and some security measures like address blocking. Bluetooth Low Energy (BLE) is tested extensively using Bluetooth dongles that were developed specifically to test the DUT with the QSW. They are used to test not only the communication itself, but also if other protocol definitions are followed, like maximum number of devices in a connection/piconet and the special notification messages.

The QSW can configure the Bluetooth interface as either a master or slave and also use a specifically designed protocol to communicate with the dongles and rearrange the network as pleased. The QSW directly commands the dongles and can program behaviors like generating a burst of dummy messages in a selected BT channel, all using BT messages.

Figure 9: *QSW Python Scripts*

Source: Author

How the QSW may interact with the dongles is briefly described in Appendix C.

Most of the V2X routines and features are part of the DUT and its version as a final product, including all V2X protocol layers. Currently, there are no message handles and no way to control the V2X protocol stack. Therefore, the QSW has limited V2X interaction, only collecting internal metrics for comparison with the data the MTS retrieves. This data is collected using the protocol stack API and used to assert system health by the *scriptV2Xmetrics*. The MTS is already geared for handling V2X tests, as seen in Section 3.3, relinquishing the need for greater QSW role in the testing of the DUT.

3.2.3 GPIO

The GPIO manager handles digital output and analog input tests. Most test procedures rely on the readings of the MTS, yet there are some tests that can be done independently by the DUT. One such example is to plug a digital output and analog input together and verify that setting the value of the output will cause a different reading on the input. This test not only checks that the GPIO is responsive to commands, but also verifies its latency.

Once connected to the MTS, the test can be performed with a single channel, allowing more precise and better detailed results. By combining the MTS and QSW, it is possible to assert the configured input sampling rate and to check if the output voltages are correct in different scenarios.

DUT Resource Management and Stressing

The first generation of the DUT was built with an additional Aurix Microcontroller, responsible for some peripherals, for security measures and to handle some of the main processor tasks. The second generation of the DUT is equipped with a more powerful processor and a different system architecture, that does not include the additional microcontroller. To handle this change, the QSW detects at startup the current hardware and firmware of the DUT. This information is used to determine which parts of the QSW should be active and also sets some of the managers behavior. The *aurixCommands* script is used when the first hardware version is detected and it translates input commands for the Aurix Microcontroller as needed.

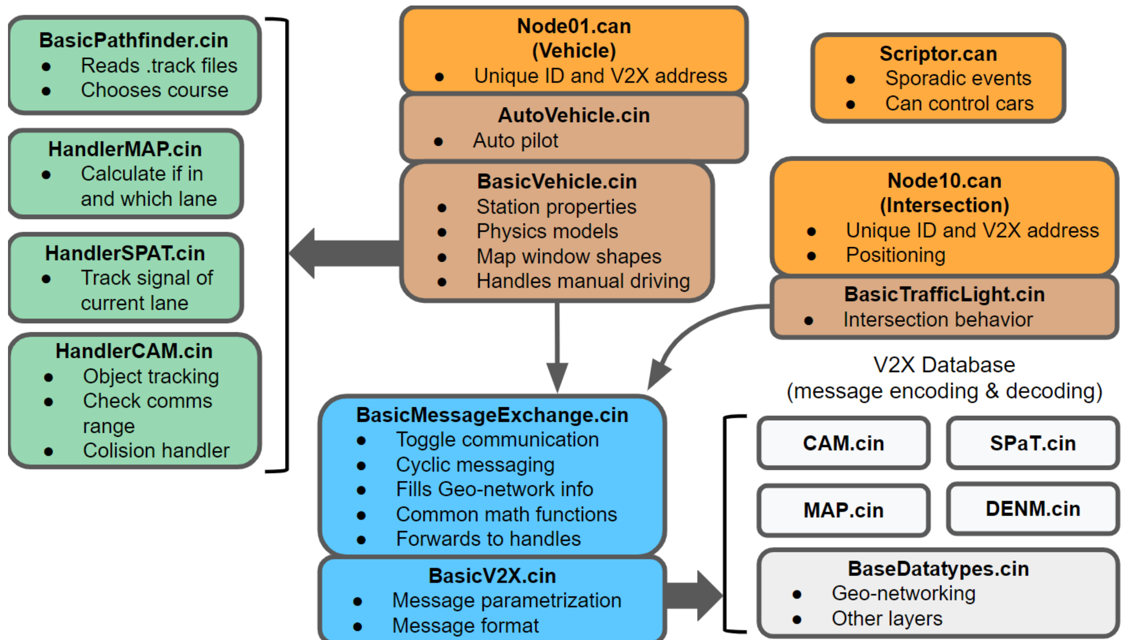
DUT resources are constantly monitored by the QSW through *mgmtSystem*. To make it possible to conduct tests under a defined level of system stress, it can also trigger scripts to consume processor time or increase internal bus usage. This module does not offer as many procedures as other parts of the QSW, instead it enhances the other managers by tracking resource usage and setting the operation conditions of the DUT. Those features are essential to most tests.

3.3 V2X

The Device Under Test implements V2X communication through a protocol stack that starts at boot. It answers according to the European V2X standard message types (INSTITUTE, n.d.) and is configured to generate beacon and CAM messages in a set period. Although the DUT protocol stack is operational, it is still not ready to interact with other processes. As there is no way to send or read messages freely, the implemented test procedures rely in reading protocol stack logs and checking metrics. This is enough to assert that all included message types are being correctly exchanged.

The MTS receives and decodes V2X messages from the DUT according to the included database. CANoe automatically validates the structure of the messages against the same database, alerting if any error is found. The work done in this stage was basically to

Figure 10: *MTS V2X node architecture*



Source: Author

download the international V2X CANoe database and adapt it to both the European Intelligent Transport Systems standard (ETSI, 2010) and the DUT requirements.

The V2X virtual network has the most nodes, with up to 14 depending on the test. These nodes work together to create a virtual environment used to test the DUT in different scenarios, where each node represents an unique V2X station. The objective of this setup is to expose the DUT as closely as possible to a real application situation.

The *.can* files define the unique station ID, and unique V2X address each node executes. It also defines the executed algorithm by importing different *.cin* files. The procedures programmed in a *.cin* define whether the node either behaves as a vehicle, as an intersection controller, or as a generic road-side unit.

Furthermore, each *.cin* file may include additional *.cin* files. This architecture allows multiple nodes to have the same behavior and avoids duplicate code. Figure 10 exemplifies the node setup using three nodes of different types. The MTS is usually configured with one Scriptor node, two intersections, and five cars. These and an example test procedure are covered in the next subsections.

Despite the fact that all nodes sending V2X messages include the file *BasicMessage-Exchange.cin*, the node behavior will vary depending on configured message handles or enabled message types. *BasicMessageExchange.cin* contains the configurable timers that cyclically trigger V2X messages and functions to characterize messages up to the Geo-Networking layer. The specifics of CAM, MAP, SPaT and DENM are configured in *BasicV2X.cin*, which uses the CANoe V2X Database API to structure the messages correctly. The API was altered similarly to the message database, as to fit the European Intelligent Transport Systems standard and the DUT requirements as well.

BasicV2X.cin has data structures that contain the necessary information to fill every used field of the V2X messages, guaranteeing that all required data is available. This

structure defines critical information to the higher layers of the node, like the station latitude, longitude, and dimensions. The initial configuration usually alters the variables defined in this structure.

3.3.1 Intersection Controllers

In a real application scenario, an intersection controller broadcasts MAP and SPaT information about one or more intersections. It directly controls traffic lights and may react to DENM messages by giving preference to an incoming emergency vehicle. In the MTS simulated environment, the intersection controllers do not react to messages, instead they stick to the signaling cycle programmed at startup, periodically sending SPaT and MAP.

Nodes that operate as traffic controllers utilize additional information in their *.can* files to determine their position, timing, and design. Traffic controllers use MAP messages to broadcast their profile to the rest of the system.

As it is not in the interest of the current test procedure to check how the DUT deals with different intersection architectures, the *.can* file does not define all fields of the MAP message directly, instead, it defines a data structure that is later used by a function to calculate and update all of the MAP message fields. This approach limits the possible intersection designs, but makes it fairly easy to position a new intersection in the map while improving code readability. Appendix D demonstrates how the simplest intersection available in the MTS is defined.

3.3.2 Vehicle Simulation

When a node represents a vehicle, its *.can* file imports *AutoVehicle.cin*, which implements an autopilot to guide the car. The purpose of this layer is to mimic a driver and it uses a closed loop control system to guide the vehicle from one set of coordinates to the next. This is done by controlling the steering wheel angle, accelerator pedal, and other values. Originally the vehicle simply followed a given route, but this approach was discarded in favor of simulating a more detailed model, a homogeneous box with four weightless wheels. The detailed model was developed because CAM messages also notify physics information and the drivers behavior. The transmitted information include if the brake pedal is engaged, the lateral acceleration of the vehicle, and the aforementioned values which are used to control the vehicle model.

The physics model is calculated in *BasicVehicle.cin*, which also sets the data structure in *BasicV2X.cin* to values commonly seen in cars. Information like weight, maximum front wheel angle, and length are defined here to a default value, but may be overwritten in the *.can* file if the test requires a different vehicle. This node also handles changes in system variables during test execution. These variables can be used by the user to take manual control over the car and guide it through the virtual environment using a game-like interface. Special messages and other parameters may also be defined, but these will be covered in Subsection 3.3.4.

The route the autopilot follows is generated by *BasicPathfinder.cin*. It uses a coordinate set defined in a *.track* file to inform the autopilot where to go. This coordinate set represents a possible route, normally a segment of a real-world road. Once it is close enough to the destination, the next coordinates are given. At the end of a route, an associated *.choice* file is read, which specifies where the vehicle may go next and which routes are connected to the current route. *BasicPathfinder.cin* then randomly selects one and restarts the process. If the *.choice* file is mapping an intersection with a traffic light, it must guide the vehicle to the correct lane, matching any defined Intersection Controller information. In other words,

all autopilots in the simulated environment follow a virtual map made of road segment coordinates and choices, which include intersection lane specifications.

There are currently 57 road segments with their associated choices in the MTS, mapping a region of Friedrichshafen, Germany. There is no *.track* file without an associated *.choice*, so all routes lead somewhere and the map is closed. If by chance *BasicPathfinder.cin* cannot find the correct files, the vehicle is halted and the MTS operator is alerted.

These *.track* and *.choice* files were made through a special CANoe node that does not belong to the MTS. It uses *BasicVehicle.cin* and all its imports to create a vehicle that is manually driven by the user. The operator sets which road segment or choice it is mapping and clicks on a button to start registering. A coordinate set, speed information and other data is added every few meters. The distance between points in the files is configured at generation, normally ranging from $4m$ to $20m$. This means that all files were generated by manually driving the vehicle through virtual roads.

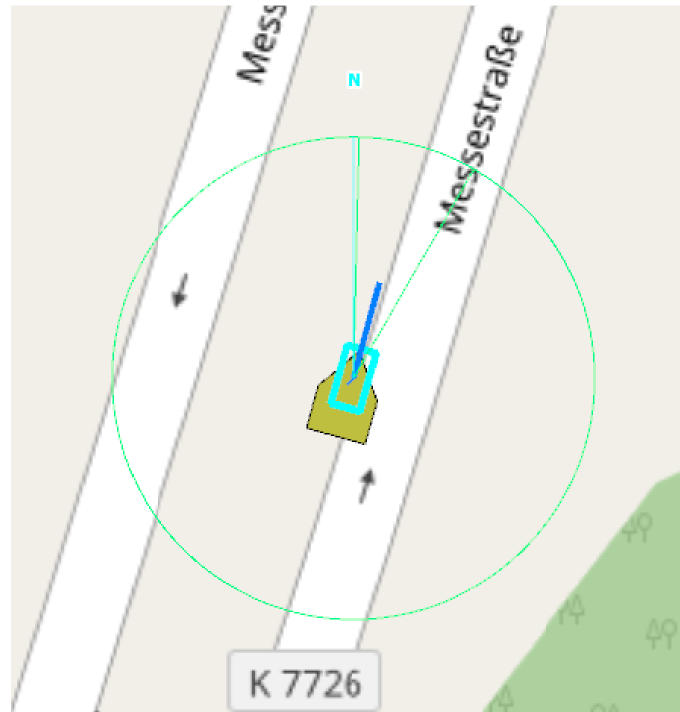
Nodes that represent a vehicle have handlers for three types of incoming messages. *HandlerMAP.cin* uses the information received through MAP messages to determine if the vehicle is in an intersection. If in an intersection, the vehicle registers its traveling lane and which traffic light signal it should follow. When receiving a SPaT message while in an intersection, *HandlerSPAT.cin* uses the registered signal group the vehicle should follow to determine if the vehicle can cross the intersection or not. This information is used directly by the autopilot. Basically, it constantly monitors *HandlerSPAT.cin* for a red light, upon which halts the vehicle and waits for a green light to resume travel.

The third implemented handler is *HandlerCAM.cin*, which constantly monitors all CAM messages received and track stations that are close to the vehicle. If a tracked object is in front of the vehicle, this handler will notify the autopilot, which will then halt the vehicle. Currently the stations that send CAM messages are either simulated vehicles or the DUT.

To avoid a pair of vehicles that are blocking each other to be forever halted, the vehicle with greater station ID will remain halted while the other one maneuvers around it. The maneuvering algorithm is not perfect, so it rarely fails and causes a halt dead-lock. Two vehicles being in a dead-lock state does not significantly impact the V2X environment, but they usually do so in the middle of the road, which means that in a matter of hours all vehicles in the simulation would be queued behind them. To avoid this scenario, the autopilot will drive the vehicle with a fixed speed for a certain time if the vehicles has been halted for too long. This means that the simulated vehicle will be forced to pass through the tracked object blocking its way, which is a preferable approach when all other alternatives are considered.

The MTS V2X interface has a map window that interprets all V2X messages and draws shapes in a map to represent them. The background is a real city map, loaded from *OpenStreetMap*. Additional shapes can be drawn freely by nodes. In case of a vehicle node, the shapes are drawn by *BasicVehicle.cin*, giving a graphic overview of the vehicles operation. Figure 11 shows a car driving through a street. The cyan rectangle represents the boundaries of the car model. A thick blue line is drawn between the center of the vehicle and the autopilot target coordinates. The brownish shape in the background is automatically drawn by the map window for every V2X station and cannot be removed. The green circle shows the object tracking range. If an object is in the slice defined by the green straight lines, it is considered as an object that is blocking the vehicles movement.

Figure 11: *MTS V2X simulated car close-up*



Source: Author

3.3.3 Special Nodes

The interfaces do not interact directly with nodes, but rather change system variables that are used by the nodes. The nodes were made to be configurable through these system variables and much of their behavior can be set by using them. The Scriptor node was implemented to manipulate system variables as if it were an user. This feature can be used to execute routine tests and procedurally control vehicles and synchronize events, generating cyclical or even random scenarios. The Scriptor node includes different scripts, including some to trigger DENM messages, or control cars directly.

Each instance (one per node) of *BasicMessageExchange.cin* constantly monitors all incoming CAM messages that match a system variable called DUT Address. Whichever station has this exact V2X address is considered the DUT of the current test by all the other stations. To simulate a real world communication range, any station that is distant beyond a determined range from the DUT will never transmit V2X messages.

DUT Address must be defined for the simulated environment to work. By default the system assigns the address of N01 as DUT Address, making it possible to run the simulation without the DUT. This behavior is programmed in the *.can* file, along with functions that draw the communication range in the map window, as a blue circle around the DUT even if the DUT is not N01.

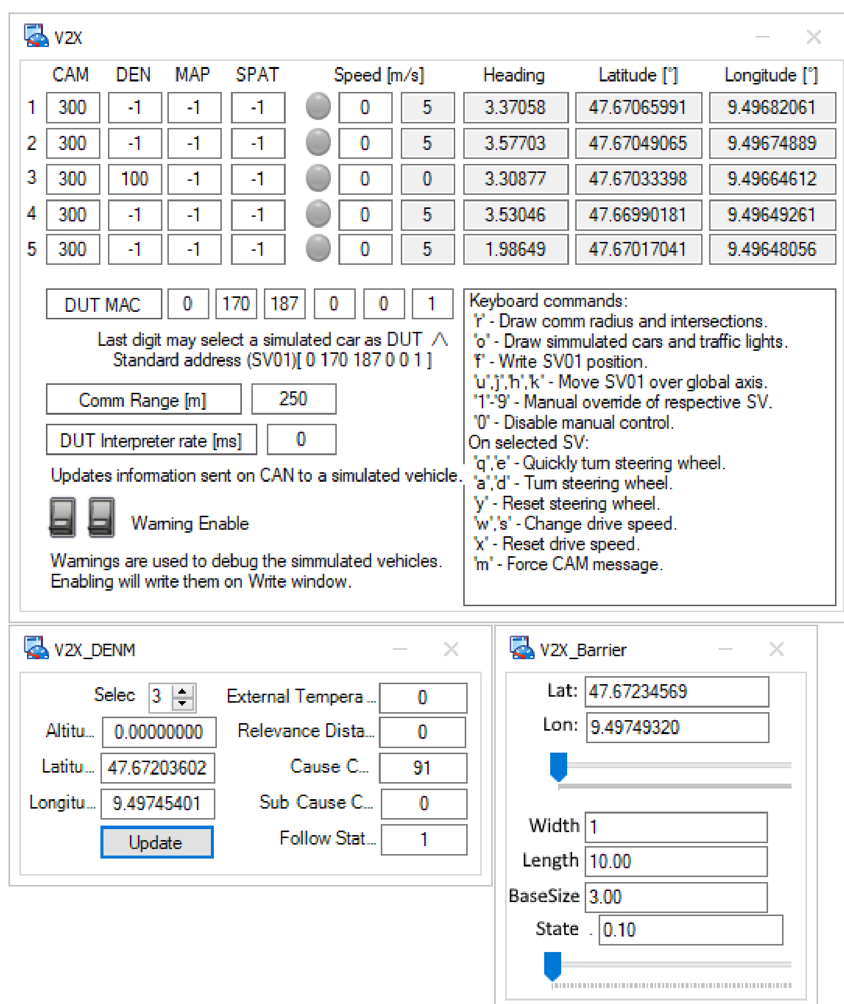
The MTS also includes a prototype Barrier Node, which defines a traffic barrier in the V2X environment. Currently an interface to position and control it is implemented, but the Barrier Node does not have any algorithm to open or close the barrier by its own.

3.3.4 Virtual V2X Environment Interface

The MTS V2X interface has three windows for input, as seen in Figure 12. The bottom-right window controls the prototype Barrier Node explained in the previous subsection.

The window at the bottom-left is used to configure the DENM message to be sent by a node, which is selected by station ID through the field "Selec". Latitude and Longitude can be specified in a DENM message to pinpoint the location of the event. If there is a landslide, for example, the vehicle can send the coordinates of the accident through the DENM message. If the vehicle itself is the source of the event, for example it being a police car asking for passage, the DENM can be set to have the same coordinates as the vehicle by leaving the fields empty. The DENM cause codes defined by the standard can be seen in Annex A.1.

Figure 12: *MTS V2X input interface*



Source: Author

The top window seen in Figure 12 lists all available simulated vehicles, in this case, five. Through this window it is possible to set the period in which each message type will be sent. If the period is zero, no messages will be sent. If the period is negative one, the node will disarm the handler and message timer, freeing computer resources. If a period

is set to the MAP or SPaT message types, the messages will be sent even though a real vehicle would never do so. This feature is used only for testing and the message content is the default defined by the European V2X standard, which carries no information.

The speed that the vehicle travels is defined in the *.track* and *;choice* files, but may be overruled by the speed defined through this interface in the "Speed" column for each vehicle. Through the circular button at the left part of this column it is possible to force vehicle speed. While forcing, the autopilot will ignore red traffic lights and will "run through" any blocking objects.

This same interface window is used to configure the DUT Address and communication range. Additional switches can enable or disable warnings in the Write Window, used to log strings as seen in Section 3.1. These warnings are neither critical to the test nor related to the DUT, they inform of occasional vehicle crashes or when the autopilot has to take a drastic decision such as exceeding speed to overcome a situation. The purpose of these warnings is to debug the autopilot. Some warnings also alert of a bad *.track* or *.choice* files.

Figure 13 shows the V2X message trace during a test, with all four message types and an additional message that has only the Geo-Networking layer, identified as a beacon message. With the shown configuration, messages are grouped by type and address. Once the message has not been received in a while, it fades out, like both messages from N10. It is probably out of the communication range. The "Dir" column shows the message direction in the connected Vector Kit, which interfaces the simulated environment with reality. Messages marked as *Tx* were generated inside the simulation and sent by the Vector Kit. Messages marked as *Rx* were received by the Vector Kit and forwarded to the simulated environment.

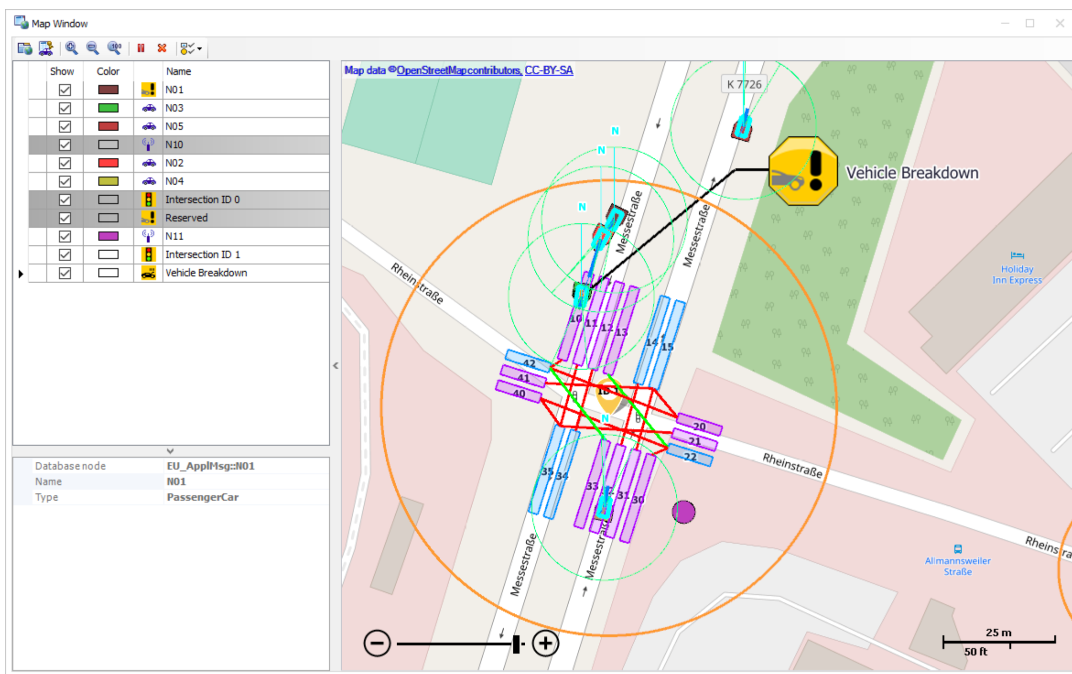
Figure 13: *MTS V2X message trace*

Time	Chn	Dir	Originator Node	Protocol	Event Info	Protocol Info	Source GN Address	Source MAC	Packet Length
82.085582	Ath 1	Tx	N01	CAM		ASN.1 defined	15 06 00:AA:BB:00:00:01	00:AA:BB:00:00:01	136
82.086112	Ath 1	Tx	N03	CAM		ASN.1 defined	15 06 00:AA:BB:00:00:03	00:AA:BB:00:00:03	136
82.085229	Ath 1	Tx	N05	CAM		ASN.1 defined	15 06 00:AA:BB:00:00:05	00:AA:BB:00:00:05	136
37.085568	Ath 1	Tx	N10	SPAT		ASN.1 defined	3D 06 00:AA:BB:00:00:10	00:AA:BB:00:00:10	110
82.084702	Ath 1	Tx	N02	CAM		ASN.1 defined	15 06 00:AA:BB:00:00:02	00:AA:BB:00:00:02	136
82.084321	Ath 1	Tx	N04	CAM		ASN.1 defined	15 06 00:AA:BB:00:00:04	00:AA:BB:00:00:04	136
36.185434	Ath 1	Tx	N10	MAP		ASN.1 defined	3D 06 00:AA:BB:00:00:10	00:AA:BB:00:00:10	182
79.360640	Ath 1	Rx	PassengerCar 4	geo_eh		Beacon	14 00 01:02:03:04:05:06	00:00:00:00:00:02	82
82.187556	Ath 1	Tx	N03	DENM	Vehicle Breakdown	ASN.1 defined	15 06 00:AA:BB:00:00:03	00:AA:BB:00:00:03	135
82.087183	Ath 1	Tx	N11	SPAT		ASN.1 defined	3D 06 00:AA:BB:00:00:11	00:AA:BB:00:00:11	129
81.888718	Ath 1	Tx	N11	MAP		ASN.1 defined	3D 06 00:AA:BB:00:00:11	00:AA:BB:00:00:11	436

Source: Author

The map window shown in Figure 14 and the trace in Figure 13 were captured simultaneously. Figure 14 shows one of the intersections of the simulated environment, where one vehicle is experiencing an engine breakdown, while two others are backed up behind it. South of the intersection, a fourth vehicle waiting at a traffic light, will move again on green. The traffic light status is conveniently indicated by the color of the lines that connect the intersection lanes. Purple lanes are entering, while blue lanes are exiting the intersection. The purple circle at the bottom indicates the location of the intersection controller, which is the origin of the V2X messages. Contrary to the vehicles, the system generates all graphics of intersections in the map window based on the MAP and SPaT network messages.

Figure 14: *MTS V2X simulated intersection close-up*



Source: Author

4 RESULTS

The Qualification Software (QSW) code is easy to understand and expand by other developers, as it uses a generic, well documented, Pythonic approach. This is also due to the usage of templates that divide the software in abstraction layers. By importing a template, a developer already has what is necessary to integrate custom code into the QSW.

In terms of software quality, the QSW has good performance and does not overuse the limited DUT resources. When an external library used by the QSW is updated in a new DUT firmware version, sometimes blocking actions are introduced. They interrupt the execution of the respective QSW thread, preventing the QSW from stopping completely at the end of the test. So far, all blocking cases are dealt by using timeouts or Python's *select* package. In such cases, altering the affected core script resolves the issue. By having the core scripts interfacing QSW's algorithm and the DUT's firmware, it is possible to solve most compatibility issues with a simple and contained fix.

One of the greatest obstacles when coding the QSW was to figure out how to implement the cores. Being a product still in development, the DUT was prone to constant change and the available user manuals were often incomplete or outdated. It was often necessary to investigate the DUT by mapping the firmware or listing the available libraries. It was common for an approach to find a dead end and for the work to be discarded. Initially, multiple cores were specified for each interface, but they were often impossible to implement.

Attention to detail was required while developing the QSW, because it had to not only has to be bug-less, but should execute as many tests as possible while dealing with a bugged DUT. An example of how meticulousness showed impressive results was when it was possible to isolate a bug in internal sockets thanks to the QSW System Manager supervision. It happened that when closing a socket connection, the DUT firmware launched a detached process to close the socket and, after correctly executing its task, it remained as a zombie process. While debugging the QSW through the QSW System Manager it was possible to see this detached process, that was in fact a bug in the DUT firmware.

The MTS uses the Vector solution for testing, which is portable and easily re-configurable. MTS hardware changes and new protocol configurations could be needed, as the DUT is still under development and not all requirements are established. On the down side, as the DUT has several interfaces, managing them all is taxing on the computer that runs CANoe. This fact is specially relevant during V2X tests, where the virtual environment has to be simulated with all its stations and physical models. Originally, the physical models were more complex and gave more information that could be transmitted through V2X. As an example, more of the MTS system had to be enabled in a single test once CAN and V2X integration tests were required, this consequently brought a higher demand of data via the CAN interfaces. As such, vehicle physical models were simplified to free up computer resources.

Most of situations while executing a test routine can be assessed through the CANoe generated metrics and statistics, which are native to the system. Further information for analysis is available via node-generated logs, and via warnings that indicate messages that do not comply with the programmed database. The time taken to program these warnings in CAPL and to correctly set the databases payed off, as in depth analysis was possible even after the test ended.

V2X communication is still being developed in the DUT and some problems with the test procedures originate from the DUT lacking critical features. An example of such is that, if the DUT has no GPS signal, the V2X protocol stack will not run, even when geographical position is not relevant to receiving messages. This prevents all V2X tests to be executed when there is no GPS. The lack of integration between the DUT firmware and V2X protocol stack is also an issue, because although the DUT can be included in the virtual environment, it cannot react with the simulated stations. The inability to neither read nor generate messages limits the number of test cases available.

The autopilot function implemented in the V2X simulation guarantees that the simulated environment will not enter a dead-lock. Still, the approach taken to solve them ignores other vehicles and purposely crashes into objects. This solution is enough to achieve the current requirements, but lacks quality. Enhancing the driving algorithm as to reduce dead-locks from rare occurrences to nonexistent would be the correct way to deal with this problem. The time required to implement this solution currently outweighs the gains for the test system.

The MTS V2X environment and its features, with explicit numbers and operations, meet all initial expectations, while offering the most impressive interface of the system. Unfortunately, the map loaded from *OpenStreetMap* does not always match accurate coordinates and fails to properly represent road width. This causes some vehicles to appear to be driving outside the roads, which could be interpreted as an error by someone who is not familiar with the system.

5 CONCLUSION

The test system is structured to introduce behaviors instead of directly implementing test cases. This approach makes it quicker to implement new test cases, as the programming is done on a higher level. Even so, more complicated procedures require extensive knowledge on the system, which is not something a good test system should demand of its users. This issue could easily be solved in the QSW by sticking to the behavioral approach, but not implementing them as individual object with a thread each, but as methods of an Interface Manager.

This alternative solution could also solve other difficulties. Having multiple threads per Interface Manager present shortcomings to synchronize data transmission. While asynchronous communication packets may prove useful in certain test cases, the majority of applications would suffer from intensified computational verification and processes. While the test cases remain at the level of the current one, having complete control over the QSW is preferable to having more options.

As the QSW interface is a command prompt, it is not as intuitive and appealing as CANoe. It was noted that most users prefer to click a button in the MTS interface than to pass commands to the QSW, even with the color coded outputs. Developing a node in the MTS to pass commands to the QSW through a GUI could improve the systems popularity.

The MTS V2X simulated environment can be used to test the DUT under different scenarios. It implements a simple and effective algorithm to put the DUT under a simulated real application. The actions taken by the vehicle where the DUT would be installed are not programmed in the DUT so far, because the DUT firmware lacks the support to do so. Yet, the MTS is ready to be used to test such algorithms, the DUT being applied to an autonomous vehicle or as a connectivity unit for assisted driving. Once the DUT firmware is completed, it would be possible to implement an algorithm in the QSW to drive the DUT through the simulated environment, testing more features and reacting to V2X messages.

Not only the MTS, but also the QSW, are meant to be further enhanced as the DUT is developed and new test cases are defined. So far, the developed system is useful to the validation team, as they are versatile enough to implement the vast majority of test cases. The system itself does not increment test coverage, but it allows multiple tests to be executed in parallel, so the previous test cases were fused into fewer, broader tests that cover more features. Parallel testing also requires less overall test time.

QSW and MTS usage not only speeds up test procedures, but also helps isolate problems with the DUT, with its logging and replay features. This grants more time to the validation team to the develop new tests. Previously, test procedures required constant monitoring, while using the MTS and QSW requires only occasional attention. In conclusion, the work is a huge gain to the team, proving that the QSW and MTS are indeed useful testing tools.

APPENDIX A - QSW EXAMPLE USAGE AND COMMANDS

This appendix shows usage examples of the QSW and some raw code.

Figure 15: *QSW Server raw help text*

```
"help     ": "Shows this help.",
"exit     ": "Exit manager.",
"list     ": "List available interfaces.",
"[\x1B[33m Note 01     ": "[I] is defined as the target interface.\x1B[m",
"[I] get     ": "Gets properties of interface.",
"[I] set [P] [V]     ": "Sets property [P] the value [V], without configuring.",
"[I] config [M]     ": "Configures interface. 'M' is method number, except on BT.
                        What 'M' means is not included in 'help' yet.",
"[I] status     ": "Shows supervisor information.",
"[I] list     ": "Lists available scripts.",
"[\x1B[33m Note 02     ": "[S] is defined as the target script.\x1B[m",
"[I] [S] start     ": "Runs SCRIPT.",
"[I] [S] stop     ": "Stops SCRIPT.",
"[I] [S] info     ": "Displys SCRIPT info string.",
"[I] [S] period T     ": "Sets cycle time of SCRIPT to T seconds, if supported.",
"[\x1B[33m Note 03     ": "Pay mind that Redis cores are blocking, if selected they may
                           prevent QSWserver from exitting (use them only on Gen1).\x1B[m",
"[I] [S] core [C]     ": "Selects target core [C], if supported. Leave [C] empty for list.",
"[I] [S] [CMD]     ": "Send command [CMD] to target script.",
"[\x1B[33m Note 04     ": "The following are QSW commands.\x1B[m",
"logSimple     ": "Logs server information on default file 'logSimple.txt'.",
"autoVariableSetup     ": "Execute default setup sequence for internal variables.",
"autoConfigure [C]     ": "Configures all CAN interfaces and sets [C] as core.",
"unsupervised     ": "List interfaces whose info is being sent to QSWclient.",
"unsupervise [I]     ": "Stop sending target interface info to QSWclient.",
"supervise [I]     ": "Start sending target interface info to QSWclient.",
"supervise     ": "Start sending all interfaces info to QSWclient.",
"[\x1B[33m Note 04     ": "The following are inherited interface-specific commands.\x1B[m"
```

Source: Author

Figure 16: *QSW Client in commander mode listing the available managers and CAN0 scripts*

```
QSW>> list
Managers/Interface list:
  CAN0
  CAN1
  CAN2
  CAN3
  GPIO
  BT
  GNSS
  WIFI
  GSM
  RS232
  System
Server: done.

QSW>> CAN0 list
CAN0 : sender [  Running  ]
CAN0 : ranger [  Stopped  ]
CAN0 : receiver [  Stopped  ]
Server: done.

QSW>>
```

Source: Author

Figure 17: *QSW Client in commander mode executing help*

```
root@VCU-7C97635040D0:~# python3 QSW/QSWclient.py
Do you want to act like 'commander' or like 'supervisor'?
commander
Client connected and running as commander.
QSW>> help
 help                        Shows this help.
 exit                        Exit manager.
 list                        List available interfaces.
 [ Note 01              [I] is defined as the target interface.
 [I] get                     Gets properties of interface.
 [I] set [P] [V]             Sets property [P] the value [V], without configuring.
 [I] config [M]              Configures interface. 'M' is method number, except on BT. What 'M' means is not included in 'help' y
                            et.
 [I] status                  Shows supervisor information.
 [I] list                    Lists available scripts.
 [ Note 02              [S] is defined as the target script.
 [I] [S] start               Runs SCRIPT.
 [I] [S] stop                Stops SCRIPT.
 [I] [S] info                Displys SCRIPT info string.
 [I] [S] period T            Sets cycle time of SCRIPT to T seconds, if supported.
 [ Note 03              Pay mind that Redis cores are blocking, if selected they may prevent QSWserver from exitting (use th
                            em only on Gen1).
 [I] [S] core [C]            Selects target core [C], if supported. Leave [C] empty for list.
 [I] [S] [CMD]               Send command [CMD] to target script.
 [ Note 04              The following are QSW commands.
 logSimple                   Logs server information on default file 'QSWlogSupervisor.txt'.
 autoVariableSetup           Execute default setup sequence for internal variables.
 autoConfigure [C]           Configures all CAN interfaces and sets [C] as core.
 unsupervised                List interfaces whose info is being sent to QSWclient.
 unsupervise [I]             Stop sending target interface info to QSWclient.
 supervise [I]               Start sending target interface info to QSWclient.
 supervise                   Start sending all interfaces info to QSWclient.
 [ Note 05              The following are inherited interface-specific commands.
 CAN0 Gen2transceiverEnable
                            CAN0 Enables TJA1145 transceiver (valid on Gen2).
 CAN1 Gen2transceiverEnable
                            CAN1 Enables TJA1145 transceiver (valid on Gen2).
 CAN2 Gen2transceiverEnable
                            CAN2 Enables TJA1145 transceiver (valid on Gen2).
 CAN3 Gen2transceiverEnable
                            CAN3 Enables TJA1145 transceiver (valid on Gen2).
 BT core [C]                 BT Sets manager core. Leave [C] empty for options list.
 BT aid                      BT Get a detailed help of BT operation and Nordic Dongle BT04 protocol.
 BT autosetup [N]            BT Execute automatic setup. [N] is name of devices to connect, leve empty to connect to none.
 BT BTlist [T]               BT List system objects. [T] selects 'Connected' or 'Notify', leave empy for all.
 BT scan [E]                 BT If [E]=='on', enable adapter scanning. Else, disable.
 BT BTprotocol               BT Displays the Nordic Dongle BT05 protocol usage.
 BT default [A]              BT Define default target as [A]. [A] empty to get default. [A] in format XX_XX_XX_XX_XX_XX or path.
 BT [ BT Note 01        BT For the following commands, target [A] can be specified or left empty for default:
 BT BTget [PROP] [A]         BT Gets target property [PROP]. [PROP] == 'all' for all properties. [A] == 'adapter' for adapter.
 BT BTset [PROP] [V] [A]
                            BT Sets value [V] to property [PROP] of target. [A] == 'adapter' for adapter. Depending on the prope
                            rty, [V] can be boolean (True/False), integer (int) or a string. Boolean properties appear as a sing
                            le digit (0/1) with 'BTget all'.
 BT connect [A]              BT Connects to target.
 BT connect name [N]         BT Connects to all devices which name starts with [N].
 BT disconnect [A]           BT Disconnects from target. [A] == 'all' to disconnect from all devices.
 BT remove [A]               BT Removes from target. [A] == 'all' to remove all devices.
 BT send [C] [A]             BT Sends char [C] to target.
 BT recv [A]                 BT Receive char [C] from target.
 BT notify [E] [A]           BT If [E]=='on', enable notifications from target. Else, disable.
 BT notifications            BT Shows dictionary with notifications info.
 GNSS supercore [C]          GNSS Define [C] as core for supervisor.
 WIFI connect [N] [P]        WIFI Connects to [N] using password [P]. Paramters assume default if not given.
 WIFI disconnect [N]         WIFI Disconnect from [N]. Disconnect from all if [N] not given.
 WIFI AP [N] [P]             WIFI Set DUT as access point with SSID [N] and password [P].
 WIFI remove [N]             WIFI Removes connection named [N].
 WIFI prop                   WIFI List all configurator properties.
 WIFI prop [A] [B]           WIFI Sets the value [B] to configurator property [A].
 System killall PROCESS      System Kills all linux processes named PROCESS.
 System getGen               System Registers current ProCV GEN information.
Server: done.

QSW>>
```

Source: Author

Figure 18: *QSW Client in supervisor mode*

```
root@VCU-7C97635040D0:~# python3 QSW/QSWclient.py
Do you want to act like 'commander' or like 'supervisor'?
supervisor
Client connected and running as supervisor.
#################################### Qualification SW - 0.2 #####################################
####  CAN0                                                                                      #
# [Running] TX: [ 000000 Err: 000000] RX: [ 000002 Err: 000000]                                 #
# Went to running     1 times. Last Running: 1.414s.                                            #
# Went to not running 0 times. Last Not Running: 0.000s.                                        #
# sender : Received:       0, Sent:     86.                                                     #
# ranger : Core set.                                                                            #
# receiver : Core set.                                                                          #
####  CAN1                                                                                      #
# [Running] TX: [ 000000 Err: 000000] RX: [ 000002 Err: 000000]                                 #
# Went to running     1 times. Last Running: 5.114s.                                            #
# Went to not running 0 times. Last Not Running: 0.000s.                                        #
# sender : Received:       0, Sent:     86.                                                     #
# ranger : Core set.                                                                            #
# receiver : Core set.                                                                          #
####  CAN2                                                                                      #
# [Running] TX: [ 000086 Err: 000000] RX: [ 000086 Err: 000000]                                 #
# Went to running     1 times. Last Running: 1.392s.                                            #
# Went to not running 0 times. Last Not Running: 0.000s.                                        #
# sender : Received:       0, Sent:     86.                                                     #
# ranger : Core set.                                                                            #
# receiver : Core set.                                                                          #
####  CAN3                                                                                      #
# [Running] TX: [ 000085 Err: 000000] RX: [ 000085 Err: 000000]                                 #
# Went to running     1 times. Last Running: 1.314s.                                            #
# Went to not running 0 times. Last Not Running: 0.000s.                                        #
# sender : Received:       0, Sent:     86.                                                     #
# ranger : Core set.                                                                            #
# receiver : Core set.                                                                          #
####  GPIO                                                                                      #
####  BT                                                                                        #
# No connected devices.                                                                         #
####  GNSS                                                                                      #
# Core not defined.                                                                             #
####  WIFI                                                                                      #
# ProCV_AP : True                                                                               #
# AndroidAPBB70 : False                                                                         #
# Freifunk : False                                                                              #
# v2c-pf : False                                                                                #
# wifi-hotspot : False                                                                          #
# b'IN-USE :  SSID       MODE   CHAN  RATE      SIGNAL  BARS  SECURITY                          #
# * :       sProCV_AP  Infra  1     0 Mbit/s  0             WPA1 WPA2                           #
####  GSM                                                                                       #
# state : connected                                                                            ##
# power : on                                                                                    #
# tech : lte                                                                                    #
# signal : 100% (recent)                                                                        #
# connection : gsm          930caf61-fb7f-4503-8f66-01cf35d411ee  gsm       ttyACM1             #
####  RS232                                                                                     #
# Supervisor not implemented.                                                                   #
####  System                                                                                    #
# CPU Usage: [ 40.0 us, 48.6 sy,  0.0 ni,  0.0  idle]                                           #
# Memory: [   1001.2 total,    726.9 free,     91.4 used,    182.9 buff/cache]                  #
# Temperature: [44.656] °C                                                                      #
# Processes:                                                                                    #
#    gpsd: OK.                                                                                  #
#    ublox-gsm-config: Error fetching service info.                                             #
#    vcu-can-adapter@can0: Error fetching service info.                                         #
#    vcu-can-adapter@can1: Error fetching service info.                                         #
#    vcu-can-adapter@vcan0: Error fetching service info.                                        #
#    vcu-can-adapter@vcan1: Error fetching service info.                                        #
#    vcu-gps-adapter@gps0: OK.                                                                  #
#    vcu-gps-adapter@imu0: OK.                                                                  #
#    vcu-gpio-adapter@gpio0: OK.                                                                #
#    vcu-serial-adapter@rs232: OK.                                                              #
#    vcu-serial-adapter@lin: OK.                                                                #
#    vcu-serial-adapter@kline: Error fetching service info.                                     #
#    ModemManager: OK.                                                                          #
#################################################################################################
```

Source: Author

Figure 19: *QSW Server executing the start up sequence*

```
root@VCU-7C97635040D0:~# python3 QSW/QSWserver.py
coreCANsocket imported.
coreCANcan failed.
coreCANshell failed.
coreGNSStelnet failed.
coreGNSSredis imported.
coreGPIOredis imported.
coreRS232redis imported.
coreBTdbus imported.
--- Logging all server messages at QSWlogServerPromt.txt ---
Mon Jun 29 06:32:46 2020
Automatic command sequence started:
Executing: logSimple
Logging thread started.
Executing: autoVariableSetup
Variables set.
Executing: autoConfigure
Interfaces configured.
Executing: CAN0 sender start
CAN0 : CAN Sender started.
Executing: CAN1 sender start
CAN1 : CAN Sender started.
Executing: CAN2 sender start
CAN2 : CAN Sender started.
Executing: CAN3 sender start
CAN3 : CAN Sender started.
Finished importing modules and executting default sequence.
Connect a client to interact
Welcome to ProCVs Qualification Software.
New client connected.
Client request: commander
Commander thread started.
New client connected.
Client request: supervisor
Supervisor thread started.
```

Source: Author

Figure 20: *QSW Server executing the stop sequence*

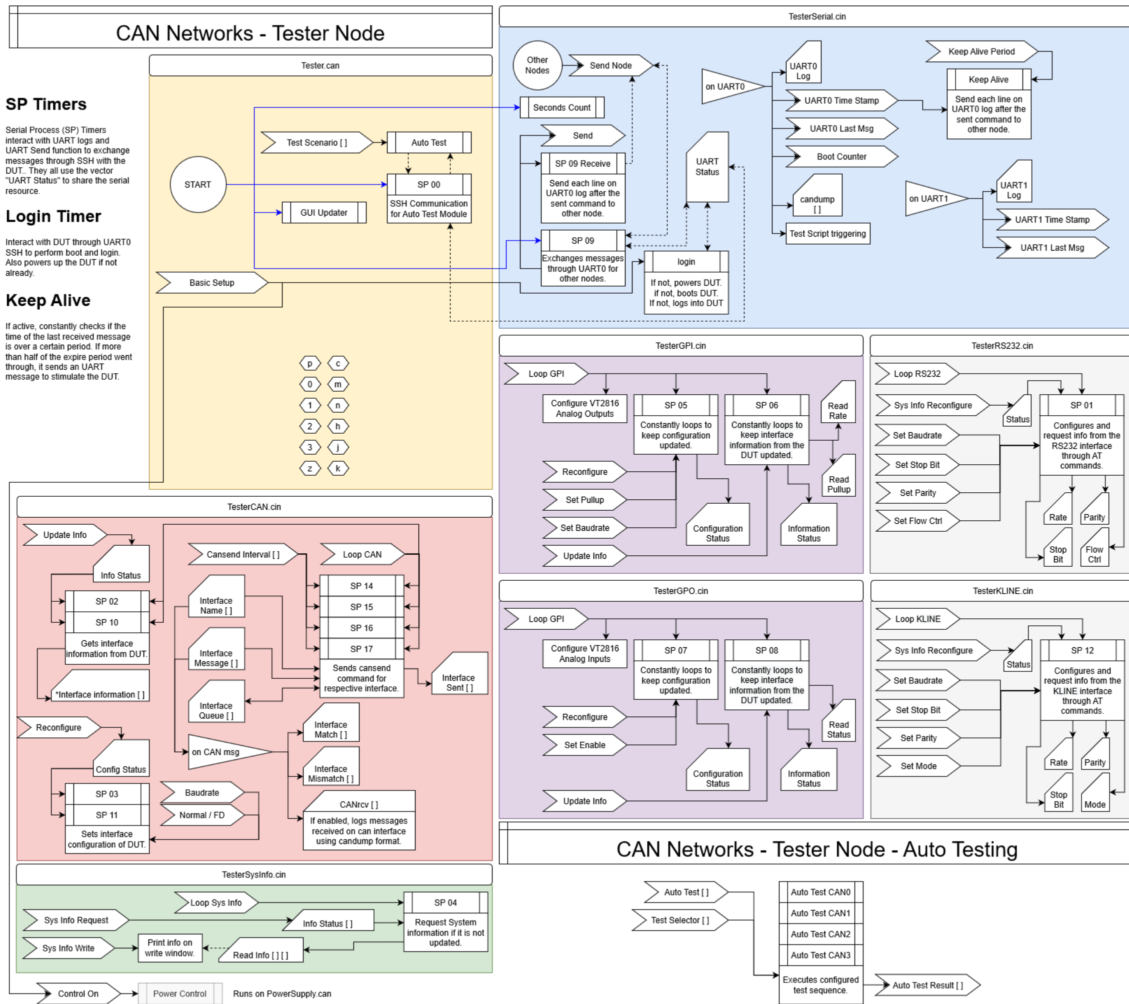
```
##############################
Closing all threads (could take up to a minute).
##############################
GPIO : GPIO sample finished.
GPIO : GPIO loop finished.
GPIO : GPIO pwm finished.
GPIO : GPIO sample finished.
GPIO : GPIO loop finished.
GPIO : GPIO pwm finished.
GPIO : GPIO sample finished.
GPIO : GPIO loop finished.
GPIO : GPIO pwm finished.
GPIO : GPIO sample finished.
GPIO : GPIO loop finished.
GPIO : GPIO pwm finished.
logSimple finishied.
BT : BT Client finished.
GNSS : IMU Static finished.
GNSS : Manager error: 'core' object has no attribute 'thread'
RS232 : RS232 Receiver finished.
Mon Jun 29 06:35:57 2020
Supervisor thread finished.
Standalone finished.
GSM : GSM Pinger finished.
CAN2 : CAN Sender finished.
CAN2 : CAN Ranger finished.
CAN2 : CAN Receiver finished.
CAN1 : CAN Sender finished.
CAN1 : CAN Ranger finished.
CAN1 : CAN Receiver finished.
CAN3 : CAN Sender finished.
CAN3 : CAN Ranger finished.
CAN3 : CAN Receiver finished.
CAN0 : CAN Sender finished.
CAN0 : CAN Ranger finished.
CAN0 : CAN Receiver finished.
System : Yes Script finished.
System : Yes Script finished.
System : Yes Script finished.
System : Yes Script finished.
System : Yes Script finished.
WiFi : WIFI Ping finished.
Finished closing threads.
Client requested: exit
Exitted.
Server: done.
Commander thread finished.
root@VCU-7C97635040D0:~#
```

Source: Author

APPENDIX B - MTS BEFORE THE QSW

Before the creation of QSW, MTS used SSH over RS232 to control the DUT. As many nodes and processes were competing for the same SSH connection, it was necessary to structure access and organize the nodes. Figure 21 gives an overview of the CAN0 sub-network, which contained the RS232 than effectively made the communication and all nodes that did not belong to a sub-network specifically.

Figure 21: *MTS CAN0 sub-network*



Source: Author

APPENDIX C - BLUETOOTH DONGLES WITH THE QSW

Figure 22 shows the raw code behind of the *help* command of *mgmtBT*. It hits on the usage of the manager with the Bluetooth Low Energy dongles developed to test the DUT.

Figure 22: *mgmtBT raw help code*

```
self.cmdHelpSpecific["core [C]   "] = "Sets manager core. Leave [C] empty for options list."
self.cmdHelpSpecific["aid     "] = "Get a detailed help of BT operation and Nordic Dongle BT04 protocol."
self.cmdHelpSpecific["BTlist [T]     "] = "List system objects. [T] selects 'Connected' or 'Notify', leave empy for all."
self.cmdHelpSpecific["scan [E]    "] = "If [E]=='on', enable adapter scanning. Else, disable."

self.cmdHelpSpecific["default [A]     "] = "Define default target as [A]. [A] empty to get default. [A] in format XX_XX_XX_XX_XX_XX or path."
self.cmdHelpSpecific["[\x1B[33m BT Note 01  "] = "For the following commands, target [A] can be specified or left empty for default:\x1B[m"
self.cmdHelpSpecific["BTget [PROP] [A]    "] = "Gets target property [PROP]. [PROP] == 'all' for all properties. [A] == 'adapter' for adapter."
self.cmdHelpSpecific["BTset [PROP] [V] [A]    "] = "Sets value [V] to property [PROP] of target. [A] == 'adapter' for adapter. Depending on the
                                             property, [V] can be boolean (True/False), integer (int) or a string. Boolean properties
                                             appear as a single digit (0/1) with 'BTget all'."
self.cmdHelpSpecific["connect [A]    "] = "Connects to target."
self.cmdHelpSpecific["connect name [N]    "] = "Connects to all devices which name starts with [N]."
self.cmdHelpSpecific["disconnect [A]    "] = "Disconnects from target. [A] == 'all' to disconnect from all devices."
self.cmdHelpSpecific["remove [A]    "] = "Removes from target. [A] == 'all' to remove all devices."
self.cmdHelpSpecific["send [C] [A]    "] = "Sends char [C] to target."
self.cmdHelpSpecific["recv [A]    "] = "Receive char [C] from target."
self.cmdHelpSpecific["notify [E] [A]    "] = "If [E]=='on', enable notifications from target. Else, disable."
self.cmdHelpSpecific["notifications    "] = "Shows dictionary with notifications info."

self.aidDic = {
    " The 'config [M]' method is special in BT interface, where [M] does not coose the method, but scan duration in s. Default 3s." :"",
    " BT testing is made with aid of the Nordic Dongles, their latest revision is BT04. Refer to 'ProCV_Bluetooth_04_Overview.xlsx'." :"",
    " They advertise themselves after power up, acting as servers. The client section of the stack can be enabled through a CHAR command." :"",
    " Commands are anwered if Notify is enabled." :"",
    " If response starts with 'asCli', the client section of the stack triggered the notification." :"",
    " If response starts with 'asSrv', the server section of the stack triggered the notification." :"",
    " If response starts with 'r##', there was a lack of resources error, where '##' is the error number." :"",
    " If response starts with 'c##', previous messages failed, where '##' is the number of fails (probably insufficient MTU)." :"",
    "[\x1B[33m Nordic Dongle BT04 possible char commands to send and their response if Notify is enabled:\x1B[m" : "",
    "0-7: Change LED pattern.": '("asSrv dtRec#", where # is display_mode)',
    "@: Forward remaining data to connected server." : 'No response.',
    "A: Enable client and start scanning for servers." : '("asSrv dtRecScan")',
    "B: Toggle notifyWhen_sendToServer." : '("asCli toggleToTrue") or ("asCli toggleToFalse")',
    "C: Toggle notifyWhen_scanning" : '("asCli toggleToTrue") or ("asCli toggleToFalse")',
    "D: Toggle notifyWhen_buttonEvt" : '("asCli toggleToTrue") or ("asCli toggleToFalse")',
    "E: Toggle notifyWhen_connSrvNtfy" : '("asCli toggleToTrue") or ("asCli toggleToFalse")',
    "F: Variable value check." : '("asCli h", m_conn_handle_to_server)',
    "G: Variable value check." : '("asSrv h", m_conn_handle_to_client)',
    "H: Increment LED pattern, overflowing at 9." : '("asSrv dtRec#", where # is display_mode)',
    "I: Set notifyWhen_buttonEvt as True." : '("asCli buttonToT")',
    "J: Set notifyWhen_buttonEvt as False" : '("asCli buttonToF")',
    "T" : '("sys autoStrUpdated")',
    "U" : '("sys autoCycles:", auto_operation_max_cycles)',
    "V" : '("sys srvOpCycle")',
    "W" : '("sys srvOpAuto")',
    "X" : '("sys srvOpSlave")',
    "Y" : '("sys timerSRVperiod:", num)',
    "Z" : '("sys timerLEDperiod:", num)'
    }
```

Source: Author

APPENDIX D - MTS V2X INTERSECTION *.CAN* FILE

```
/*@!Encoding:1252*/
includes
{
  #include "..\BasicNodes\BasicTrafficLight.cin"
}

variables
{
  int64 stateGreenTime[4] = { 2000,0,0,0 };
  int64 stateYTime[4] = { 3000,0,0,0 };
  int64 stateRYTime[4] = { 2000,0,0,0 };
  int64 allRedTime = 5000;
}

void configureTrafficLight()
{
  char buffer[10];
  int intCounter, sgCounter;
  double intersectionAngle = -0.3;
  double laneAngle;
  double relativeX = 800;
  double relativeY = 300;
  double laneLength = 2000;


  laneAngle = PI/2;
  for(intCounter=0 ; intCounter < elCount(intersec) ; intCounter++ )
    for(sgCounter=0 ; sgCounter < elCount(intersec[intCounter].sg) ;
    sgCounter++ )
      intersec[intCounter].sg[sgCounter].currentState = -1;


  station.Dynamics.latitude = 47.674810;
  station.Dynamics.longitude= 9.498574;
  station.Info.ID = 16;
  thisMAC[5] = station.Info.ID;
  snprintf(buffer,elcount(buffer),"TL%d", station.Info.ID);
```

```
strncpy_off(stationName,0,buffer,10);
station.Info.mobileOrStationaryFlag = 0; // 0 = RSU, 128 = Vehicle
station.Info.stationType = 15;

//================================================================
// Northern Intersection
//================================================================

intersec[0].lat = 47.674710 * 1e7;// Latitude of Traffic Light Center (
  Position)
intersec[0].lon = 9.498840 * 1e7; // Longitude of Traffic Light Center (
  Position)
intersec[0].laneWidth = 350;      // Lane width in cm
intersec[0].speedLimitType[0] = 0;
intersec[0].speedLimitValue[0] = 0;
intersec[0].numberOfLanes = 4;
intersec[0].intersectionIDoffset = 0;

intersec[0].lane[0].X = relativeX*cos(intersectionAngle) + relativeY*sin
  (-intersectionAngle);
intersec[0].lane[0].Y = relativeX*sin(intersectionAngle) + relativeY*cos
  (-intersectionAngle);
intersec[0].lane[0].angle = intersectionAngle + laneAngle;
intersec[0].lane[0].length = laneLength;
strncpy(intersec[0].lane[0].dir,"01",3);
strncpy(intersec[0].lane[0].man,"000",4);
intersec[0].lane[0].id = 10;

intersec[0].lane[1].X = -relativeX*cos(intersectionAngle) + relativeY*sin
  (-intersectionAngle);
intersec[0].lane[1].Y = -relativeX*sin(intersectionAngle) + relativeY*cos
  (-intersectionAngle);
intersec[0].lane[1].angle = intersectionAngle + laneAngle;
intersec[0].lane[1].length = laneLength;
strncpy(intersec[0].lane[1].dir,"10",3);
strncpy(intersec[0].lane[1].man,"100",4);
intersec[0].lane[1].id = 11;
intersec[0].lane[1].con[0].lane = 20;
intersec[0].lane[1].con[0].signalGroup = 1;

intersectionAngle += PI;

intersec[0].lane[2].X = relativeX*cos(intersectionAngle) + relativeY*sin
  (-intersectionAngle);
intersec[0].lane[2].Y = relativeX*sin(intersectionAngle) + relativeY*cos
  (-intersectionAngle);
intersec[0].lane[2].angle = intersectionAngle + laneAngle;
intersec[0].lane[2].length = laneLength;
strncpy(intersec[0].lane[2].dir,"01",3);
strncpy(intersec[0].lane[2].man,"000",4);
intersec[0].lane[2].id = 20;
```

```
  intersec[0].lane[3].X = -relativeX*cos(intersectionAngle) + relativeY*sin
    (-intersectionAngle);
  intersec[0].lane[3].Y = -relativeX*sin(intersectionAngle) + relativeY*cos
    (-intersectionAngle);
  intersec[0].lane[3].angle = intersectionAngle + laneAngle;
  intersec[0].lane[3].length = laneLength;
  strncpy(intersec[0].lane[3].dir,"10",3);
  strncpy(intersec[0].lane[3].man,"100",4);
  intersec[0].lane[3].id = 21;
  intersec[0].lane[3].con[0].lane = 10;
  intersec[0].lane[3].con[0].signalGroup = 1;

  intersec[0].numberOfSG = 1;
}
```

ANNEX A - FIRST ANNEX TITLE

Annex text.

A.1 DENM Cause Codes

Value of the direct cause code of a detected event as defined in ETSI EN 302 637-3 [i.3]. The value is assigned according to the clause 7.1.4 of ETSI EN 302 637-3 [i.3].

The cause codes are described as following:

- reserved (0): the value is reserved for future use,
- trafficCondition (1): the type of event is an abnormal traffic condition,
- accident (2): the type of event is a road accident,
- roadworks (3): the type of event is roadwork,
- value 4: reserved for future usage,
- impassability (5): the type of event is unmanaged road blocking, referring to any blocking of a road, partial or total, which has not been adequately secured and signposted,
- adverseWeatherCondition-Adhesion (6): the type of event is low adhesion,
- aquaplaning (7): danger of aquaplaning on the road,
- value 8: reserved for future usage,
- hazardousLocation-SurfaceCondition (9): the type of event is abnormal road surface condition,
- hazardousLocation-ObstacleOnTheRoad (10): the type of event is obstacle on the road,
- hazardousLocation-AnimalOnTheRoad (11): the type of event is animal on the road,
- humanPresenceOnTheRoad (12): the type of event is human presence on the road,
- value 13: reserved for future usage,
- wrongWayDriving (14): the type of the event is vehicle driving in wrong way,

- rescueAndRecoveryWorkInProgress (15): the type of event is rescue and recovery work for accident or for a road hazard in progress,
- value 16: reserved for future usage,
- adverseWeatherCondition-ExtremeWeatherCondition (17): the type of event is extreme weather condition,
- adverseWeatherCondition-Visibility (18): the type of event is low visibility,
- adverseWeatherCondition-Precipitation (19): the type of event is precipitation,
- value 20-25: reserved for future usage,
- slowVehicle (26): the type of event is slow vehicle driving on the road,
- dangerousEndOfQueue (27): the type of event is dangerous end of vehicle queue,
- Value 28-90: reserved for future usage,
- vehicleBreakdown (91): the type of event is break down vehicle on the road,
- postCrash (92): the type of event is a detected crash,
- humanProblem (93): the type of event is human health problem in vehicles involved in traffic,
- stationaryVehicle (94): the type of event is stationary vehicle,
- emergencyVehicleApproaching (95): the type of event is approaching vehicle operating emergency mission,
- hazardousLocation-DangerousCurve (96): the type of event is dangerous curve,
- collisionRisk (97): the type of event is a collision risk,
- signalViolation (98): the type of event is signal violation,
- dangerousSituation (99): the type of event is dangerous situation in which autonomous safety system in vehicle is activated,
- value 100-255: reserved for future usage.

REFERENCES

ETSI. *ETSI EN 302 637-2: Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 2: Specification of Cooperative Awareness Basic Service*. Sophia Antipolis, France, 2019. p. 45. Available from: <https://www.etsi.org/deliver/etsi_en/302600_302699/30263702/01.04.01_60/en_30263702v010401p.pdf>.

ETSI. *ETSI TS 102 637-1: Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 1: Functional Requirements*. Sophia Antipolis, France, 2010. p. 60. Available from: <https://www.etsi.org/deliver/etsi_ts/102600_102699/10263701/01.01.01_60/ts_10263701v010101p.pdf>.

FREEDESKTOP. *D-Bus Message System*. Available from: <https://www.freedesktop.org/wiki/Software/dbus/>.

FRIEDRICHSHAFEN, Z. *VCU Pro Onboard Unit*. Available from: <https://www.zf.com/products/en/connectivity/products_52100.html>.

GUTTAG, J. V. *Introduction to Computation and Programming Using Python: With Application to Understanding Data.* [S.l.]: MIT Press, 2016. ISBN 978-0-262-52962-4.

INSTITUTE, E. T. S. *ETSI Standards on Vehicular Communication*. Available from: <https://www.etsi.org/deliver/etsi_en/302600_302699/>.

ISO. *ISO/TC 204 Intelligent transport systems*. International Organization for Standardization. 2019. Available from: <https://www.iso.org/committee/54706.html>.

ROSSUM, G. VAN; WARSAW, B.; COGHLAN, N. *Style Guide for Python Code*. 2001. Available from: <https://www.python.org/dev/peps/pep-0008/>.