POLITECNICO DI TORINO
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

# Automotive Architecture Framework for Rapid ECU Prototyping in C++

**Relatore**
Prof. Fulvio Risso

**Candidato**
Giovanni Musto

**Supervisore aziendale**
**Italdesign-Giugiaro S.p.A.**
Ing. Domenico Perroni

Anno accademico 2022-2023

**Abstract**

The software development in the automotive industry follows very tight guidelines, focused mainly on the safety of passengers and pedestrians. One of the most used developing lifecycles is the V-model, a very rigorous and therefore very slow and expensive process. It is very effective for series production, but inconvenient for prototyping development.

The aim of this thesis work is to build a new development workflow specifically for low-series production, show-cars, and prototypes.

The first part of my work consisted in modelling some test components in Vector PREEvision and exporting the resulting ARXML files.

Then a Python script and a C++ library have been developed; the aim of the script is to parse the ARXML files and export all the needed information, while the C++ library provides the programmer with ease of access to the CAN network and to the messages and signals described in said files.

Finally, a demo program has been written and run on a development board, making use of the library to control a series production headlamp, by mimicking its control ECU.

In conclusion, it has been shown that it is feasible to develop and deploy an ECU in C++ with limited effort, hopefully paving the way to easier, cheaper and faster car prototyping.

# Acknowledgements

First of all, I would like to thank all of my family, for the immense support given to me during my studying period; in particular I would like to thank my parents, my brother Lorenzo, and my grandparents.

Then, a big thank you goes to all my friends; without them, these years would have been much, much worse. Among everyone, I would like to thank Ivan, Mattia, Domenica and Ciro, that have been my flatmates for a forcefully too short period.

I would also like to acknowledge Prof. Fulvio Risso and Italdesign, for giving me the opportunity to do this master's thesis.

Last, but not least, I would like to acknowledge Domenico Perroni, for following me in this work, alongside Massimiliano Raspante, Paolo Picciafoco and Luca Valentini.

# Contents


**List of Tables** V

**List of Figures** VI

**List of Listings** VII

**1 Introduction** 1

1.1 The V-model . . . 1

1.2 AUTOSAR . . . 2

1.2.1 The ARXML file format . . . 3

1.2.2 The E2E Protocol . . . 4

1.3 The CAN bus . . . 5

1.3.1 CAN frame format . . . 5

1.3.2 CAN FD . . . 7

1.3.3 SocketCAN . . . 8

**2 System Design** 9

2.1 Vector PREEvision . . . 9

2.2 System and software design . . . 10

2.2.1 Software design . . . 10

2.2.2 Hardware design, mapping and routing . . . 15

2.2.3 ARXML export . . . 17

2.3 Software development and integration . . . 17

2.3.1 Problems of the traditional approach . . . 17

**3 V-model alternative** 19

3.1 Python ARXML parser . . . 20

3.2 C++ library . . . 22

3.2.1 CanMap . . . 22

3.2.2 CanLib . . . 25

3.3 Other components . . . 27

3.3.1 Demo applications . . . 27

3.3.2 CMake files . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.4 Deployment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

**4 Real world demo** 30

**5 Conclusions and future developments** 32

**Bibliography** 34

# List of Tables

1.1 CAN frame format explanation . . . . . . . . . . . . . . . . . . . . . 6

# List of Figures


1.1 The V-model . . . 2
1.2 E2E Protocol overview . . . 4
1.3 CAN base and extended frames . . . 5
1.4 CAN and CAN FD comparison . . . 7
2.1 Vector PREEvision features . . . 10
2.2 PREEvision system and software design . . . 10
2.3 Lights Connections model . . . 11
2.4 Lights model . . . 12
2.5 Front Lights model . . . 13
2.6 Rear Lights model . . . 14
2.7 Conversion method example . . . 15
2.8 Lights hardware model . . . 16
2.9 Lights CAN frame . . . 17
3.1 S32G development board . . . 29
4.1 2020 Audi A3 . . . 30

# List of Listings

1.1 Test ARXML extract . . . 3
3.1 Python ARXML parser extract . . . 20
3.2 CanMap class header . . . 24
3.3 CanLib class header . . . 26

# Chapter 1

# Introduction

The history of electronics in the vehicles go hand in hand with the development of technology itself. The first electronic engine control module is now almost 50 years old [1]. Nowadays, the number of Electronic Control Units (ECUs) found in a vehicle can be as high as 150.

As the systems became more and more complex, many coding standards and guidelines were developed, such as MISRA, that covers C [2] and C++ [3].

While an automotive architecture is usually developed using the V-model, the scope of this work is to find an alternative model to specifically use in prototyping and low-series production. Before that, however, some concepts need to be introduced.

## 1.1 The V-model

One of the most used development models in the automotive industry is the V-model. Also called the V-cycle, it is a very robust model, consisting of two distinct phases; the left side represents the project definition phase, while the right side corresponds to the verification phase. Every industry subdivides the two sides in slightly different ways.

The automotive industry usually divides the left side in three parts: design, development and integration. Each of these steps starts from a formal document, and ends in another one, allowing different parties to carry out each step with minimal overlap.

The right side, instead, deals with the verification of the corresponding step on the left side. At each step, the result is validated against the specification, going back to the corresponding phase if needed.

During the design phase, the software and network architecture is defined. All the software components and their interactions are defined, and the number of ECUs and their network topology is chosen; finally, how the software components

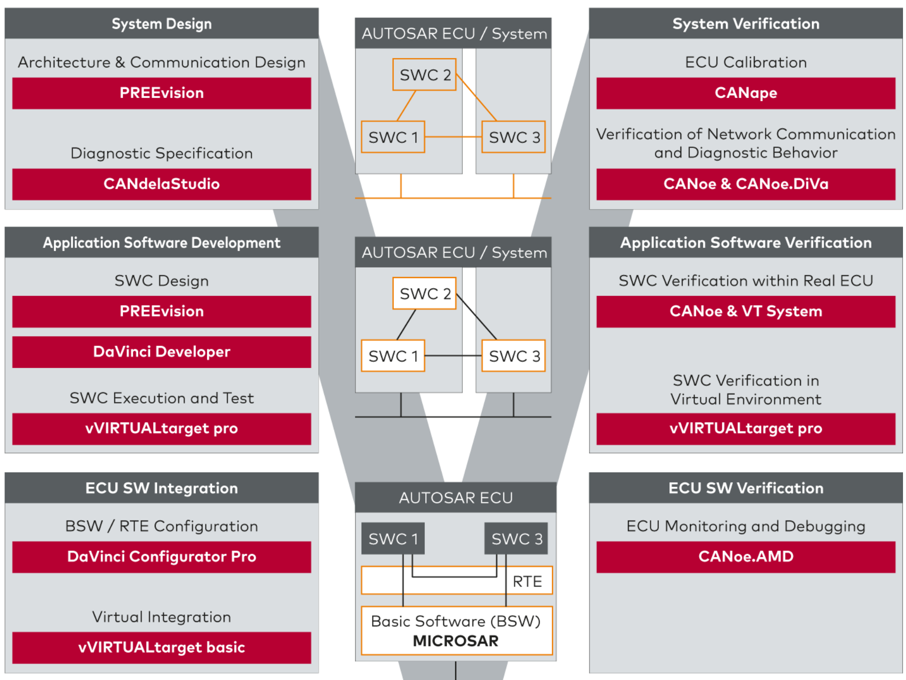


Figure 1.1: The V-model as explained by Vector in terms of its own software.

are distributed on the ECUs is decided.

The development phase consists in the design and implementation of all the ECU software. This is usually done, like the other steps, following the AUTOSAR guidelines, which will be explained later.

Finally, the integration phase consists in putting together all software pieces and deploying them to the physical ECUs.

One thing to keep in mind about the V-model is that the verification is not only done on the right side, but throughout the development life-cycle.

When dealing with series production vehicles, these steps are often outsourced to different entities. The robustness of the model itself guarantees that the final result corresponds to the requirements.

## 1.2 AUTOSAR

*AUTOSAR (AUTomotive Open System ARchitecture) is a global partnership of leading companies in the automotive and software industry to develop and establish the standardized software framework and open E/E system architecture for intelligent mobility* [4].

AUTOSAR aims to standardize the software modules and the application interfaces, making easier to reuse components, even across different manufacturers.

Many AUTOSAR standards, however, arise from already established OEM practices, therefore they allow multiple variants, in order to accommodate what manufactures were already doing, thus limiting interoperability. One such example is the *E2E Protocol*, which will be explained in section 1.2.2 on the following page.

One thing to note is that AUTOSAR only provides the specification of each software module, while the implementation is left to the manufacturer, although open source implementations of some components do exist [5–8].

### 1.2.1 The ARXML file format

AUTOSAR also defines a format to exchange architecture artefacts between compatible software, called ARXML [9]. An ARXML file is nothing more than an XML file with a specific syntax.

The root element is always called *AUTOSAR* and includes zero or more packages.

Packages are containers for elements of the same type and can be nested. Every package and element has a nested *SHORT-NAME* tag that carries its name.

Nodes can contain references to other nodes, but unlike conventional XML, the hierarchy is not explored by tag name, but by the *SHORT-NAME* inner text. The root reference is / and deeper nodes are referenced adding a / and the corresponding *SHORT-NAME* to its parent reference.

```xml
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <AUTOSAR xmlns="http://autosar.org/schema/r4.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
     ↪ schemaLocation="http://autosar.org/schema/r4.0 AUTOSAR_00046.xsd">
3    <ADMIN-DATA>
4      <LANGUAGE>EN</LANGUAGE>
5      <USED-LANGUAGES>
6        <L-10 L="EN" xml:space="default"/>
7      </USED-LANGUAGES>
8    </ADMIN-DATA>
9    <AR-PACKAGES>
10     <AR-PACKAGE UUID="0a9b246c9e373afb9330a953844b3b6e">
11       <SHORT-NAME>Communication</SHORT-NAME>
12       <AR-PACKAGES>
13         <AR-PACKAGE UUID="644b31cca3f63b8aa8f86bf533b94a9a">
14           <SHORT-NAME>Frames</SHORT-NAME>
15           <ELEMENTS>
16             <CAN-FRAME UUID="0a8fde40b186da0c63295fd48X0a8fde40b186da0c63295fd4300">
17               <SHORT-NAME>Test_CANFrame</SHORT-NAME>
18               <FRAME-LENGTH>4</FRAME-LENGTH>
19               <PDU-TO-FRAME-MAPPINGS>
20                 <PDU-TO-FRAME-MAPPING UUID="0a8fde40b186da0c63295fd48x0a8fde40b186da0c63295fd4200">
21                   <SHORT-NAME>new_PDUFrameAssignment</SHORT-NAME>
22                   <PACKING-BYTE-ORDER>MOST-SIGNIFICANT-BYTE-LAST</PACKING-BYTE-ORDER>
23                   <PDU-REF DEST="I-SIGNAL-I-PDU">/Communication/PDUs/Test_PDU</PDU-REF>
24                   <START-POSITION>0</START-POSITION>
25                 </PDU-TO-FRAME-MAPPING>
26               </PDU-TO-FRAME-MAPPINGS>
27             </CAN-FRAME>
28           </ELEMENTS>
29         </AR-PACKAGE>
30         <AR-PACKAGE UUID="8ea7d5f648373dcb81a5df496ef28b08">
31           <SHORT-NAME>PDUs</SHORT-NAME>
32           <ELEMENTS>
33             <I-SIGNAL-I-PDU UUID="0a8fde40b186da0c63295fd47X0a8fde40b186da0c63295fd4400">
34               <SHORT-NAME>Test_PDU</SHORT-NAME>
35               <LENGTH>4</LENGTH>
```

Listing 1.1: An extract from a test ARXML file.

For example, on line 23 of listing 1.1 on the previous page, we can find a reference to the element starting on line 33, while the reference for the element on line 16 would be */Communication/Frames/Test_CANFrame*.

The names of the packages are not standardized, although some of them are common between tools. This means that everything that can be modelled can be serialized, but not every tool can deal with all the information contained in an ARXML file.

### 1.2.2 The E2E Protocol

One of the many standards defined by AUTOSAR is the E2E Protocol Specification [10]. It is a an additional communication layer which sits between the upper and lower communication layers (see fig. 1.2), whose goal is to add end-to-end protection (hence the name) to critical messages, so that common network problems, such as message corruption, repetition or loss can be detected.

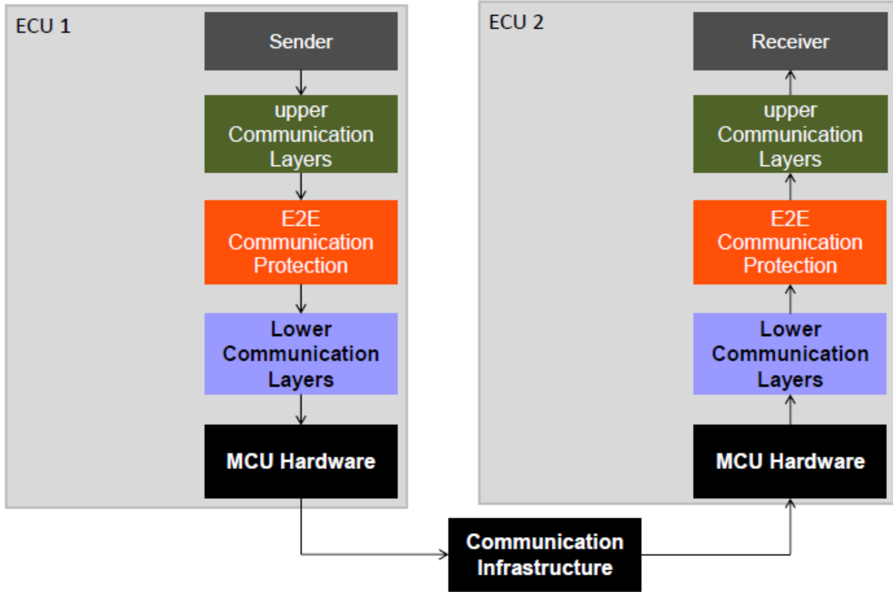


Figure 1.2: Overview of E2E communication protection between a sender and a receiver.

There are, at the time of writing, 14 different profiles described by the standard. Every profile has different use cases with respect to the level of protection that's needed, and to the underlining communication network, but each of them implements a counter, to detect repeated, missing, or out-of-order messages, and a Cyclic Redundancy Check (CRC) to detect corrupted messages, regardless of any protection already built in the lower communication layers.

## 1.3 The CAN bus

The Controller Area Network (also called CAN bus) is one of the most used vehicle communication networks. The development started in 1983, and it has since been standardized by the International Organization for Standardization (ISO) [11–14]. Modern vehicles usually include at least two or three CAN networks, but the number can be much higher [15].

The strengths of CAN include the robustness, the ability to get relatively high data rates, and the low wire count. Even if other other automotive communication networks, such as FlexRay and Ethernet, are being employed nowadays, CAN still remains ubiquitous in the industry. The disadvantages include the small payload (up to 8 bytes) and the lower speeds when compared with newer alternatives.

CAN uses a differential twisted pair of wires, called CAN high (CANH) and CAN low (CANL), making it very tolerant to common mode noise, and facilitating the detection of bus problems. It is terminated with a 120 Ω resistor at each end, and features a dominant state (actively driven by devices) and a recessive state (pulled by resistors).

The recessive state represents a logical 1, while the dominant one a logical 0. There's no bus arbitration: while transmitting, a device is also listening, and aborts the transmission in the event that the bit being received is different from the one being transmitted. Since all the devices are synchronized and sample each bit concurrently, this means that in case of collision on one bit, the device transmitting a dominant bit can continue transmitting, while the others have to stop; this is exploited to establish a message priority, with messages with lower IDs taking precedence in case of collision.

The standard does not set a specific bitrate for CAN, allowing speeds from 125 kbps up to 1 Mbps, with lower speeds being less susceptible to interferences and thus allowing longer network distances.

### 1.3.1 CAN frame format

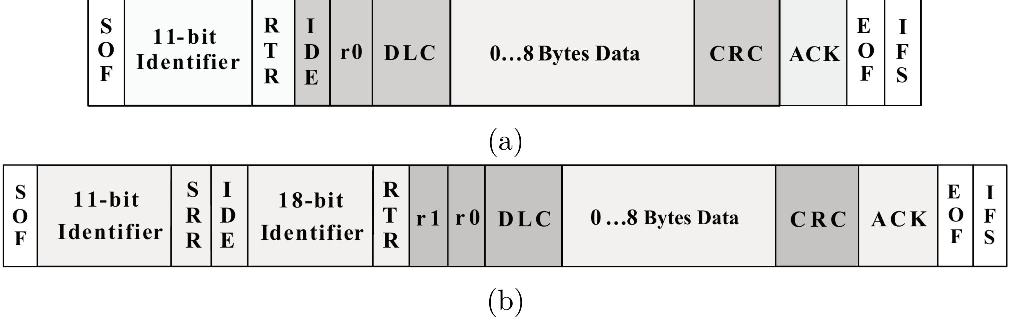


Figure 1.3: CAN base (a) and extended (b) frame formats.

CAN does not identify the sender (nor the receiver), but every message has an ID (that, as discussed before, doubles as its priority). The ID can be 11 bits long (in the CAN base frame format) or 29 bits long (in the CAN extended frame format), and the two formats can coexist on the same bus.

Figure 1.3 on the previous page shows the two formats, while table 1.1 explains the meaning of every field.

Table 1.1: Explanation of the fields of CAN base and extended frame formats.

| **Field name** | **Length (bits)** | **Description** |
|---|---|---|
| Start of frame (SOF) | 1 | Dominant (0) bit, used to signal the start of a frame transmission |
| 11-bit Identifier | 11 | The unique identifier of the message in case of base frames, or the first part of the 29-bit ID in case of extended frames |
| Remote transmission request (RTR)[a] | 1 | Dominant (0) for data frames and recessive (1) for remote request frames; its position is different between base and extended frames |
| Substitute remote request (SRR) | 1 | In extended frames, its located where the RTR bit would be, and is always recessive (1) |
| Identifier extension bit (IDE) | 1 | Dominant (0) for base frames and recessive (1) for extended frames |
| 18-bit Identifier | 18 | In extended frames, the second part of the 29-bit ID |
| Reserved bit 1 (r1) | 1 | Reserved bit, only present in extended frames, must be transmitted as dominant (0) |
| Reserved bit 0 (r0) | 1 | Reserved bit, must be transmitted as dominant (0) |
| Data length code (DLC) | 4 | Size of data in bytes (only values 0-8 are valid) |
| Data | 0–64 | Payload of the message, from 0 to 8 bytes according to the DLC |

Continued on next page

Table 1.1: Explanation of the fields of CAN base and extended frame formats. (Continued)

| **Field name** | **Length (bits)** | **Description** |
|---|---|---|
| CRC | 16 | 15 bits are the CRC-15-CAN calculated over the bits from SOF up to and including data (15 bits), while the last bit is the CRC delimiter, always recessive (1) |
| ACK | 2 | The first bit is the ACK slot, transmitted recessive (1) and asserted dominant (0) by the receiver, while the second bit is the ACK delimiter, always recessive (1) |
| End of frame (EOF) | 7 | Used to signal the end of the frame, always recessive (1) |
| Inter-frame spacing (IFS) | 3 | At least 3 recessive (1) bits, used to separate consecutive messages |

[a] A remote request frame is a special frame with 0 length data, and is used to ask other nodes to send the data frame with the corresponding ID.

*Note*: Fields in grey are only present in the extended frame format.

### 1.3.2 CAN FD

The CAN standard specifies a maximum bitrate of 1 Mbps, but CAN FD (Flexible Data-Rate) has also been specified, allowing speeds up to 5 Mbps.

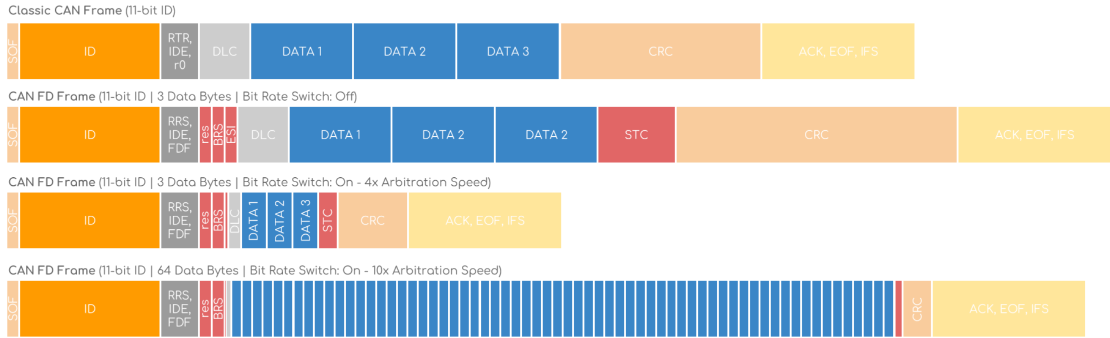


Figure 1.4: Comparison of a classical CAN frame and three CAN FD frames of different sizes and at different data bit rates.

CAN and CAN FD can coexist on the same bus, as long as they use a different set of IDs. Classical CAN devices can tolerate CAN FD transmissions because the same bitrate as classical CAN is used for the arbitration phase, switching to a

faster rate only when transmitting the data and the CRC. On the contrary, CAN FD devices must be capable of receiving classical CAN frames.

The FD mode is signalled by sending the *r0* bit as recessive (logical 1). New bits are added to support all the features of CAN FD, but the principles are mostly the same. The maximum payload has also been increased to 64 bytes, versus the mere 8 of classical CAN. CAN FD messages can still use both the standard and extended addressing.

### 1.3.3 SocketCAN

SocketCAN is an implementation of the CAN protocols in Linux [16]. It is the de-facto standard when dealing with CAN in Linux, since it superseded all previous implementations, that were often specific to some CAN hardware. SocketCAN abstracts the hardware, providing a socket interface, much like TCP/IP sockets. Of course, the two types of sockets are not identical, since CAN has some quirks and peculiarities that TCP/IP hasn't, and vice versa, but their usage is as similar as possible, making use of the same system calls, such as *bind*, *read*, *write* and so on. SocketCAN consists of many components, such as the socket in itself, the configuration layer, the user space applications, and the virtual CAN driver.

The socket layer is the core of the implementation. It abstracts the hardware and provides the application with APIs to send and receive messages, filter them, get the arrival timestamp, and so on.

The configuration layer, instead, deals with the physical configuration of the CAN hardware, allowing to set the speed, CAN FD support, echo of sent messages, and so on. The configuration layer can be accessed with both user space applications, such as *ip*, and specific APIs by the application.

SocketCAN also provides user space applications such as *cansend*, that allows the user to send a CAN frame from the command line, *candump*, that prints to console all the frames received on a CAN interface, and many more. These tools are very useful during development and testing to make sure everything is working correctly.

Finally, SocketCAN also provides a virtual CAN driver, called *vcan*, that allows the user to emulate in software one or more CAN interfaces, in order to simulate and test CAN communications without the need of real hardware. This feature has been used a lot during development, in conjunction with the user space applications, to test everything locally, before deploying the software on real hardware.

# Chapter 2

# System Design

The system design phase, in automotive development, is a very heterogeneous process. The E/E (Electric/Electronic) architecture requires the design of everything related to the electric world, from the wiring harnesses to the high-level functionalities of the single ECUs. With that in mind, it's easy to understand why the architecture design is a process that involves many figures with different skill sets, and why the vehicle is usually divided into *domains*, each one being a set of related components of the car.

The design of an automotive network architecture was already been explored in another thesis work, by Luca Valentini [17], but in order to proceed further, I needed to understand the design process myself, focusing on the software and network design.

## 2.1 Vector PREEvision

*PREEvision is the premier tool for model-based development of distributed, embedded systems in the automotive industry and related fields. This engineering environment supports the entire technical development process in a single integrated application* [18].

PREEvision in an all-in-one solution for E/E architecture modelling. It has full support for the AUTOSAR methodology, and can import and export data via ARXML and many more file formats, allowing migration from other software and/or integration into an existing workflow.

Figure 2.1 on the next page shows the various features of the PREEvision software, as advertised by Vector, but only three of the components were explored for this work: the software and hardware architectures, and the communication layer.

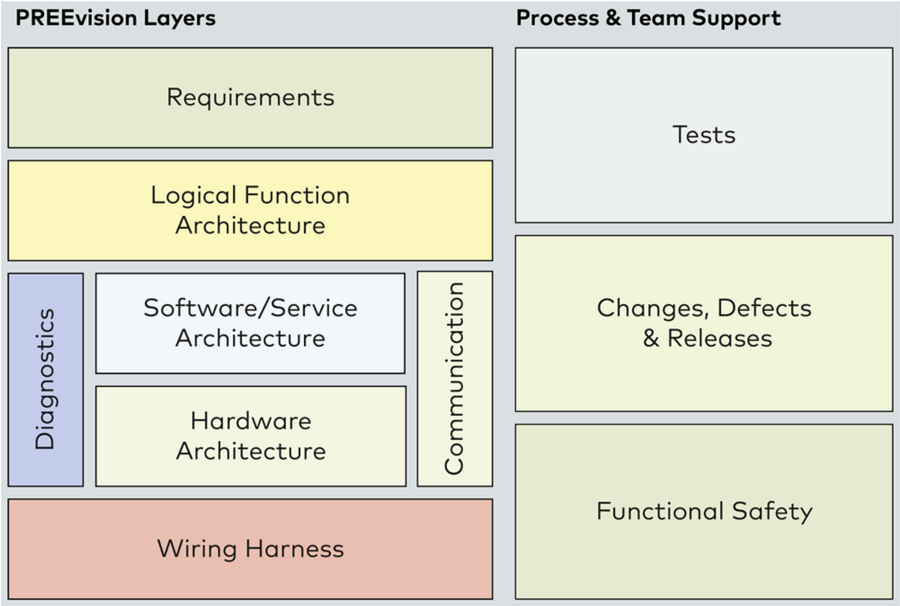


Figure 2.1: Features of the Vector PREEvision software.

## 2.2 System and software design

Figure 2.2 shows the process for the system and software design phase of the V-model. Of course not all steps are mandatory: in my work I started from scratch and not from an ARXML import, and only focused on CAN networks.

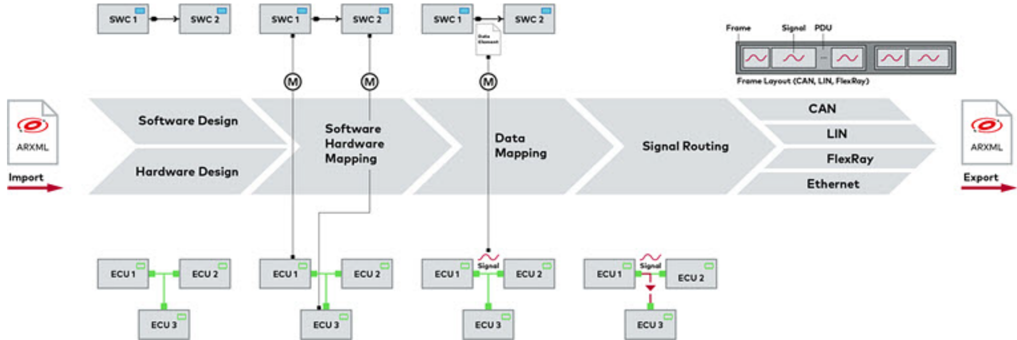


Figure 2.2: System and software design process in PREEvision.

### 2.2.1 Software design

In this phase, software components and their interactions are modelled, independently from the physical model. Software artefacts can be grouped into compositions, and have input and output ports, each one of them usually corresponding

to a signal. Everything is done graphically, by placing artefacts on a canvas and drawing lines to connect them.

Figures 2.3 to 2.6 on pages 11–14 show a simple software model for the exterior lights of a vehicle.

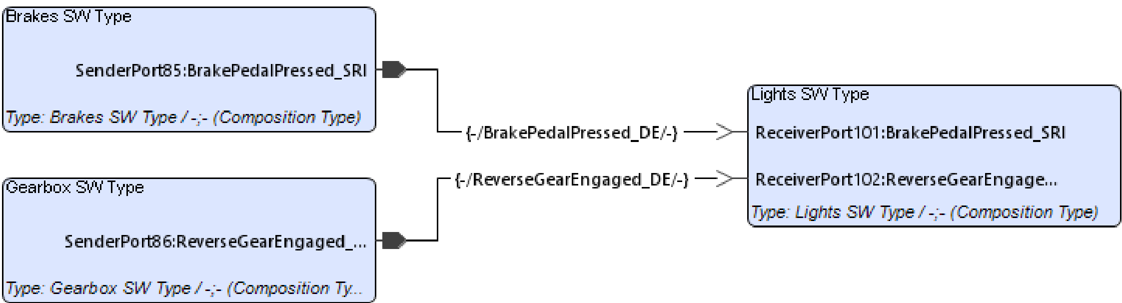


Figure 2.3: Lights Connections composition PREEvision model.

Figure 2.3 shows the general composition for the lights domain. The *Brakes SW Type* and *Gearbox SW Type* were not entirely modelled, while *Lights SW Type* is shown in detail in fig. 2.4 on the following page, alongside its components *Front Lights SW Type* in fig. 2.5 on page 13 and *Rear Lights SW Type* in fig. 2.6 on page 14.

The symbol represents a sensor and/or actuator, while ■ is a SW component. There's no real difference between a sensor and an actuator, and in fact an artefact can be both. A sensor usually only has sender ports ( ), while an actuator only has receiver ports ( ).

Another important step in this phase is the data types creation. Every connection has to have a *data element* assigned to it, and each data element needs a *data type*. *Application data types* are used to differentiate different types of value at the logical level; for example, one can define both *Brake_Pedal_Position_DT* and *Accelerator_Pedal_Position_DT*, and map them to an integer *implementation data type*, but they will represent different types of data at the logical level.

A data type can also have a *computation method*, a way to convert from the internal representation of the value to the physical one, or vice versa. They can be textual (just a description of each possible value, or range of values) or numerical (a mathematical function to apply to each value, or range of values). Figure 2.7 on page 15 shows an example of conversion method, with a piecewise numerical conversion and a textual conversions for two of the possible ranges of values.

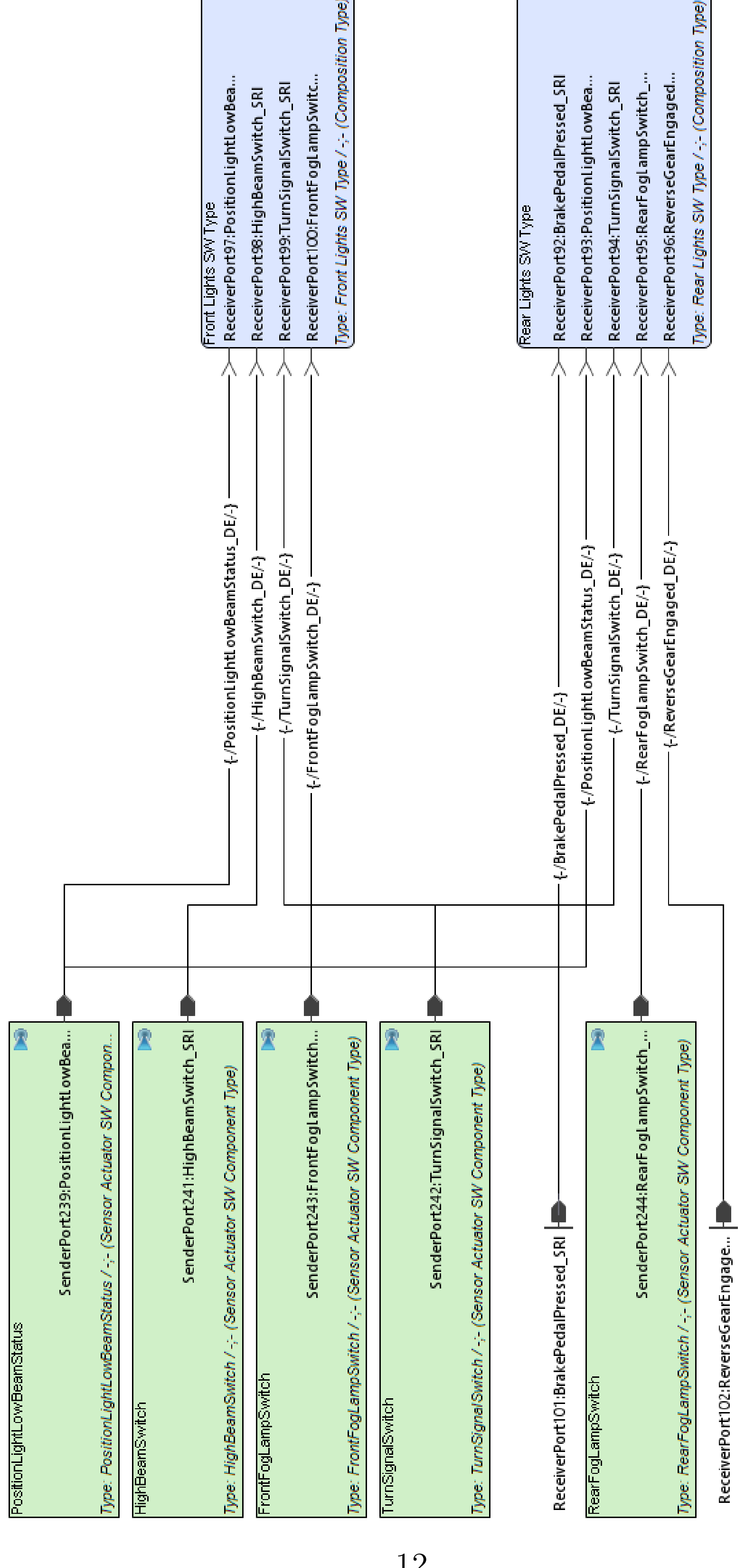


Figure 2.4: Lights composition PREEvision model.

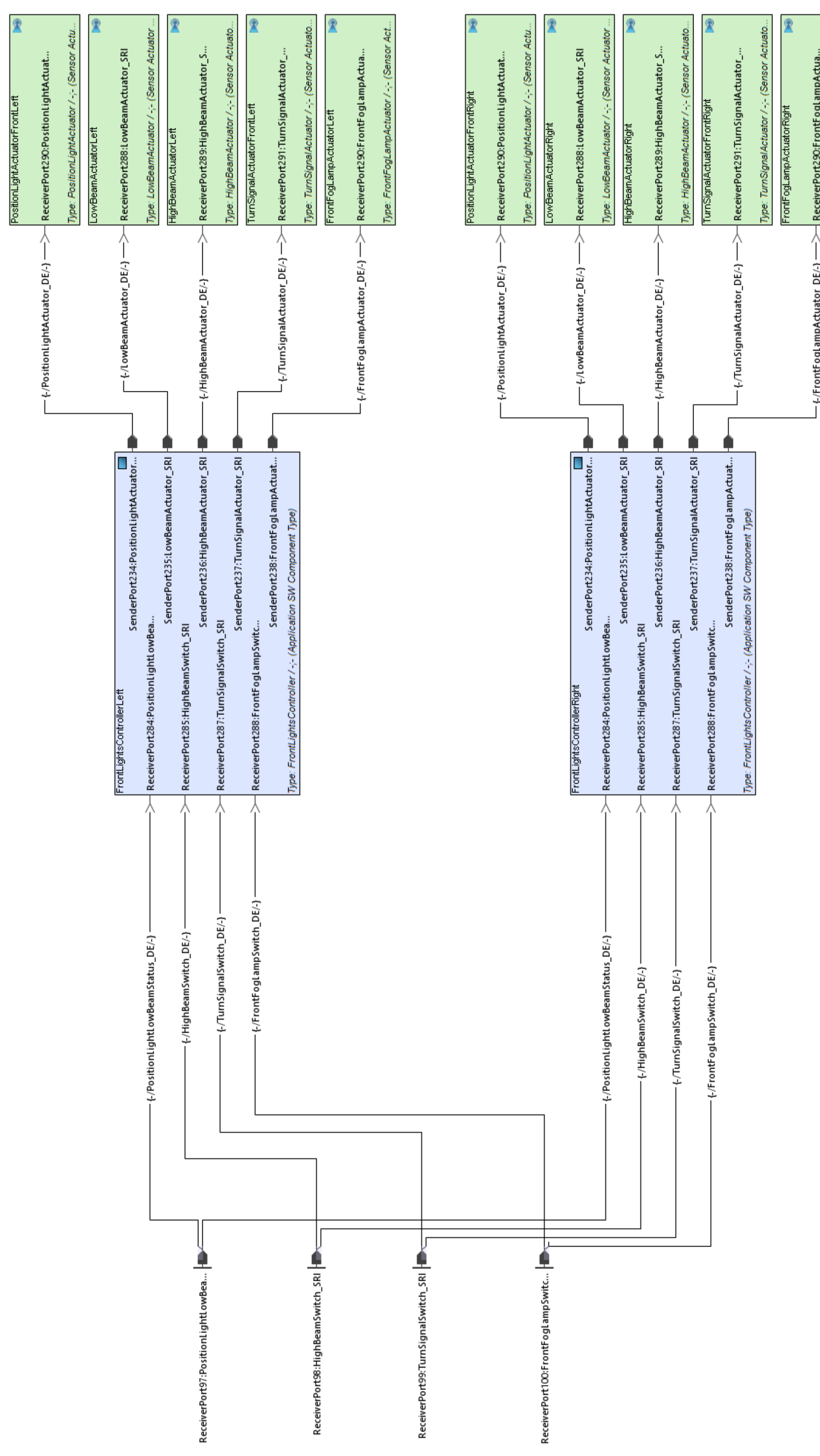


Figure 2.5: Front Lights PREEvision model.

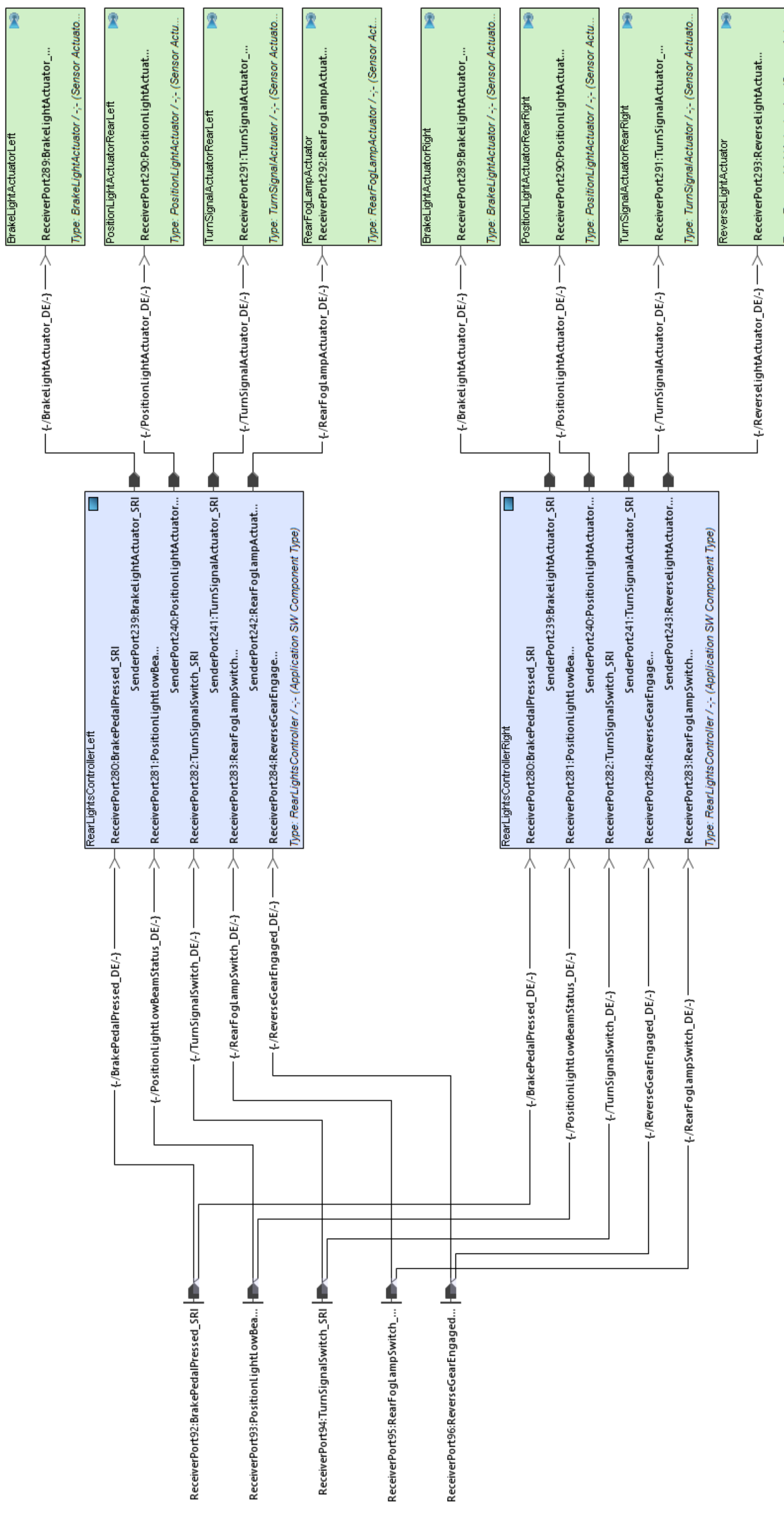


Figure 2.6: Rear Lights PREEvision model.

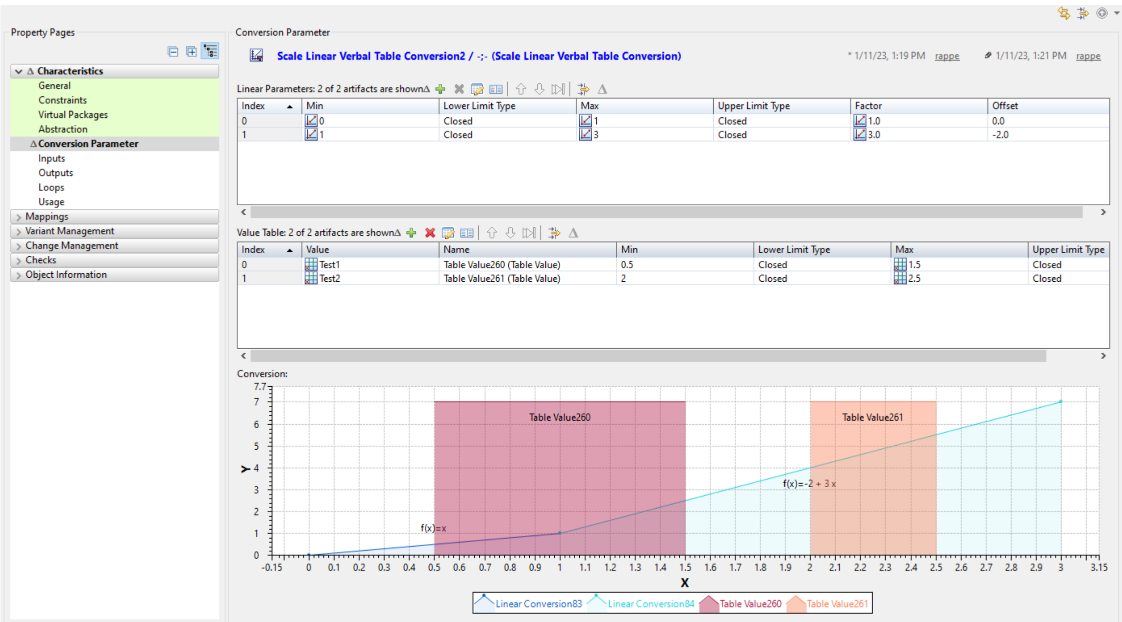


Figure 2.7: Example of a conversion method.

### 2.2.2 Hardware design, mapping and routing

The next step after the software design is the modelling of all the ECUs, and the networks that connect them, defining their types and number. This phase is similar to the previous one, but concerns the physical components, instead of the logical ones. In the end, software components will be mapped to physical components. This is almost never a 1:1 map, since in practice the same HW component can fulfil the role of multiple SW components. The separation between SW and HW components means it's easier to move functionalities between ECUs without having to model everything from scratch.

Figure 2.8 on the following page shows the hardware model for the exterior lights domain modelled before, and how each SW component has been mapped onto an ECU.

Once everything has been modelled, it's possible to run the *signal router*, that automatically creates a signal for each data element, and calculates the path every signal has to traverse to reach its final destination.

At this point, it's possible to group signals into PDUs and assign a PDU to a frame. In fig. 2.9 on page 17, the exterior lights signals (7 bits in total) have been mapped to a 3-bytes PDU and inserted into a CAN frame. In this step it's also possible to define E2E-protected PDUs and assign them to a frame in place of the normal PDU.

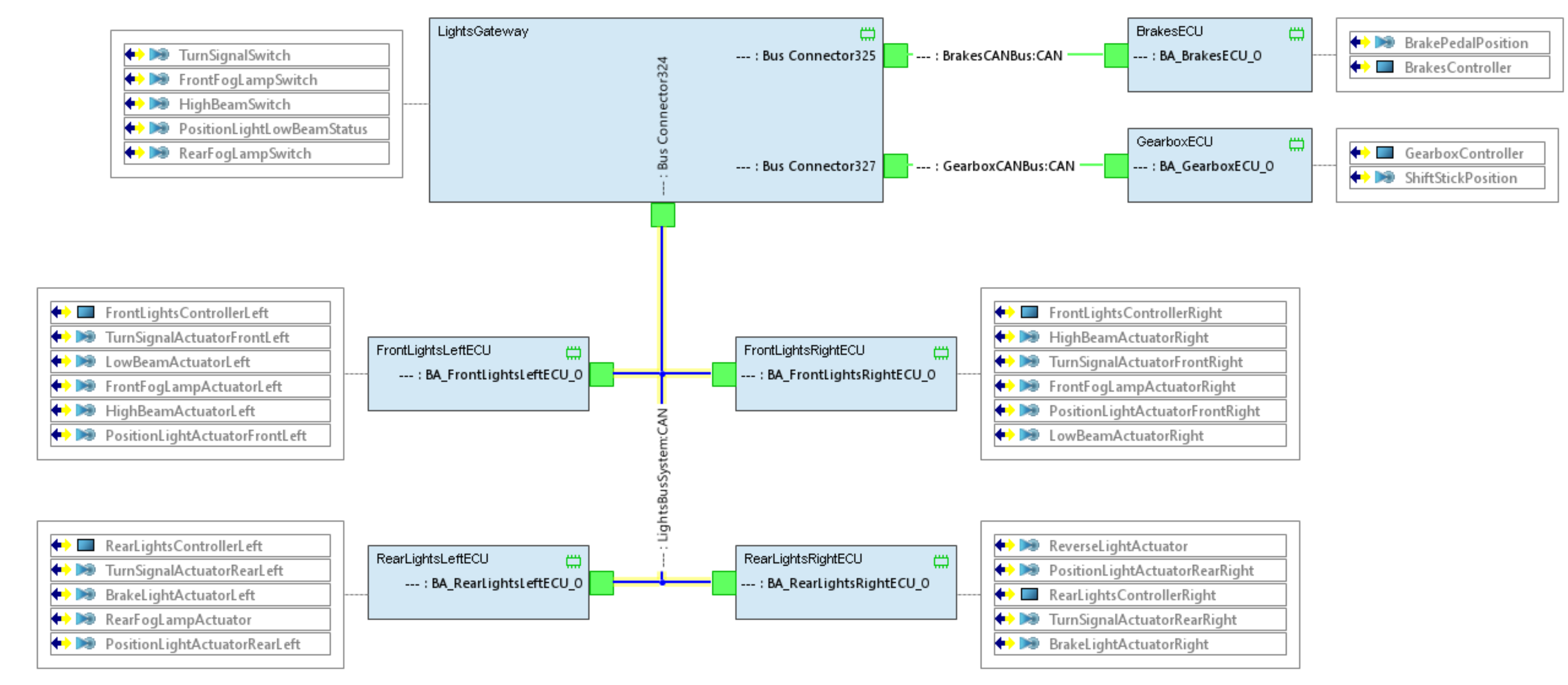


Figure 2.8: Hardware model of the exterior lights domain with SW components mapping visible.

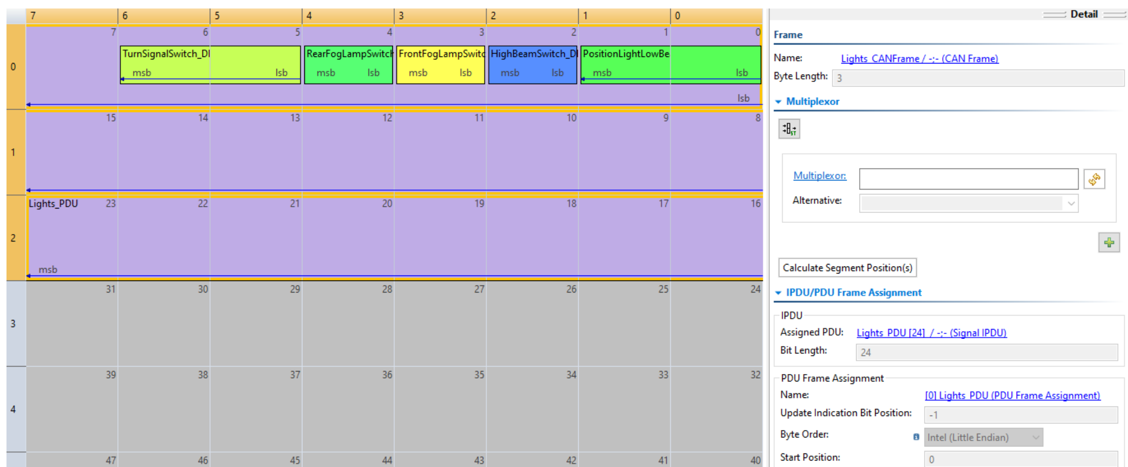


Figure 2.9: Lights signals in their CAN frame.

### 2.2.3 ARXML export

Once everything has been modelled, it's possible to export the result to one or more ARXML files, so that the software development phase of the V-model can be carried out.

In PREEvision this process requires only a few clicks, but it's important to note that very few checks are carried out on the output; in many cases, in PREEvision it's possible to insert contradictory information, omit needed parameters, provide incoherent or out-of-standard values, and so on, without the software complaining, so it's up to the user to verify the formal correctness of the model.

## 2.3 Software development and integration

These two phases of the V-model are often outsourced to third parties; the former consists in developing the software of all the ECUs modelled in the previous phase, while the latter includes the configuration of the physical hardware and deployment of the developed software.

### 2.3.1 Problems of the traditional approach

While this process is well established for series production, it can be tricky to apply to prototypes and low-series productions for a number of reasons:

- Is very expensive
- Takes time to get the final result

- Is not very flexible when a small change is needed

The conventional development phase is very expensive for various reasons: first of all, the traditional approach requires very expensive software, and people trained to use it, while following strict coding practices. This is fine for series production, since the cost is spread out over the millions of units that are going to be produced, but it can be uneconomical when developing for prototyping.

Then there's the problem of timing: offloading the work to contractors and subcontractors means that more time is required to complete the development process. Again, this is fine in production, where development cycles are longer, but not ideal for prototypes, where times are often tight.

Finally, one big problem is in the nature of prototyping itself: rarely the first prototype is the final design, since designers often experiment with various concepts and ideas before converging on the final product. This is in contrast with the traditional development approach, since a small change would mean a new development phase, with all the costs and timing problems discussed before.

The integration phase, on the other hand, is strictly tied to the development one, and usually a third party provides the physical hardware with the software already deployed and ready to go. Again, this means that there's no way to make some changes without starting the development phase all over again.

All things considered, it's clear that the car prototyping industry would benefit a lot from a new approach to the software development and integration phases, as I'll discuss in chapter 3 on the following page.

# Chapter 3

# V-model alternative

Given the aforementioned problems of the traditional V-model approach when applied to prototyping, the need to find an alternative arises. Moving the development in-house could solve the timing and flexibility problems, but can result in higher costs, given the need to acquire the needed software and human resources.

The solution seems to be a new workflow, specifically tailored for prototyping, but still compatible with the V-model system design phase. The first phase of the V-model is left in place, so that one can reuse the model in case the prototype will evolve in a series production vehicle.

In order to speed up the development process, and make it easier too, some processes can be automated, in particular the generation of the application interfaces. The details will be explained in the rest of this chapter, but the process is more or less composed of the following steps:

- Export of the ARXML files after the design phase
- Processing of the ARXMLs and generation of a description of networks, ECUs, messages, signals, and their interactions
- Development of the ECU software in C++
- Deployment of the software on a development board

In order to enable this process, two main components are needed:

- A way to process the ARXML files, extract all the needed info, and put them in a form that's easy to integrate with C++ code
- A C++ library to provide interfaces to both the communication and the application layers

In the following sections, both components will be explained in detail, and some snippets of code will be presented.

## 3.1 Python ARXML parser

In order to process the ARXML file, I decided to go the fast route and develop a Python script. Other approaches have been considered, but Python has been chosen as a trade-off between speed and ease of development. The script however, has been kept very simple, with only one external dependency: *lxml* [19]. Other Python modules, specifically made to process ARXML, already exist, but have been found to be incomplete and/or not well documented. Writing my own parser, moreover, allowed me to better understand the ARXML format and its peculiarities.

Listing 3.1 shows the first few lines of the Python parser. The *traverse_arxml* function is the core of the ARXML traversal. As stated before, an ARXML file is not traversed like a normal XML, so this function handles it. In order to improve the speed of the parsing, it also employs a caching mechanism, so that previously visited parts of the tree can be easily retrieved. The *open_arxml* function, instead, opens an ARXML file and sets all the needed variables. The rest of the code deals with traversal of the tree and extraction of all values and will not be listed here.

```
1  #!/usr/bin/env python3
2  import sys
3  import math
4  import json
5  from argparse import ArgumentParser, FileType
6  from lxml import etree as ET
7
8  root = None
9  ns = None
10 cached_root = None
11
12 def open_armxl(file):
13     global root
14     global ns
15     global cached_root
16     root = ET.parse(file).getroot()
17     ns = root.nsmap  # The default namespace, used by etree
18     cached_root = {
19         'ref': root,
20         'children': {}
21     }
22
23 def traverse_arxml(root, path_str):  # This function finds a node given the reference string
24     path = path_str.split('/')
25     if path[0] == '':
26         del path[0]  # Remove leading '/'
27     cached_node = cached_root
28     for e in path:
29         next_cached = cached_node['children'].get(e)
30         if next_cached is not None:
31             cached_node = next_cached
32         else:
33             cached_node['children'][e] = {
34                 'ref': cached_node['ref'].find(".//*/SHORT-NAME[.='" + e + "']/..", ns),
35                 'children': {}
36             }
37             cached_node = cached_node['children'][e]
38         if cached_node['ref'] is None:
39             return None
40     return cached_node['ref']
```

Listing 3.1: The first lines of the Python ARXML parser.

Of course not every aspect of the ARXML export is covered by the script, with only the necessary information being parsed and extracted. In particular, the following features are extracted:

- Topology of the network, i.e. ECUs and their connections
- Frames and mapping of their senders and receivers
- Signals and their internal to physical conversion methods
- End-to-end protection configurations and protected frames

First of all, a topology of the networks is created, by mapping all the ECUs and their connections, finding out which frames are sent on each port and what's their direction (outbound or inbound). Since the C++ library, as of today, can only handle CAN communications, only the CAN networks are extracted.

For each frame, the parser extracts the ID and length, knows if the extended addressing is to be used, if it's a CAN-FD frame, and what E2E protection (if any) it needs. If the frame represents a periodic message, also the timing and tolerance are extracted.

Then, the signals in each frame are processed; the parser extracts the position in the message, the length, endianness and initial value, and checks if the value is signed or unsigned.

The textual and numerical internal to physical conversion methods for the signal are then extracted, alongside the label and/or coefficients for each of the ranges.

Finally, the end to end protection configurations are extracted and associated with the corresponding frames.

The parser takes as input one or more ARXML files, and by default processes everything, but it can be configured to process only some ECUs, or exclude some specific ECUs. The way it works is that if some information is only relevant to an ECU that hasn't been selected, it is omitted from the output; this allows, for example, to have an output that's not generic for an entire network, but tailored to the single ECU.

The end result consists of two files, whose names are specified on the command line; one contains the map all all ECUs, while the other all the messages, signals and associated info. The format of these files is a C string containing the JSON dump of all the extracted info, and they can be directly linked with the C++ library.

JSON has been chosen because it is the de facto standard for data exchange between applications, guaranteeing portability and making it easier to eventually switch to a different programming language in the future.

The parser has been tested with the previously modelled components, but also with some series production ARXML files, to make sure that it would be suitable for real-world usage.

## 3.2 C++ library

In order to make the software development easier, a C++ library has been developed, aiming to abstract the CAN communication layer, and provide ease of

access to the messages and signals defined in the design phase. It has been designed to have as few dependencies as possible (in fact even the boost libraries have been avoided), and it makes use of Linux's SocketCAN to handle the communication and the *nlohmann/json* library [20] to translate the JSON exported from the previous step to C++ objects. In fact, the only hard requirement for the library is a Linux environment with SocketCAN and a working C++17 compiler.

The main features of the library are:

- Simplify the sending and receiving of messages by abstracting the details of the CAN layer
- Automatically send periodic messages, getting the correct timing directly from the ARXML
- Be notified when the value of a signal changes via a callback function
- E2E protect sent messages and check received ones for errors

The library is composed of two main classes: *CanLib*, that handles the communication side, and *CanMap*, that handles the messages side.

### 3.2.1 CanMap

The CanMap class has the duty to handle the maps of the messages and the ECUs. It is not directly used by the application, but provides its functionalities to *CanLib* (see section 3.2.2 on page 25).

First of all, CanMap defines the methods (not shown here) needed by the nlohmann/json library to convert the JSON representation of the data to C++ internal structures.

As for the data it holds, CanMap keeps three maps:

- The ECU map, a static map with the list of inbound and outbound messages for each ECU
- The signal map, a static map with all the info for each signal, such as its length and position in a message
- The values map, a dynamic map for all the non-static values, such as the current value of the signals, the E2E protection counters, and so on

The static data can be accessed at any time, but the dynamic map is protected from concurrent access.

The *getSignalCallback()*, *setSignalCallback()* and *removeSignalCallback()* functions allow the user to be notified via a callback function when the value of a signal changes.

Listing 3.2 on the next page shows the header file for the CanMap class.

```
/*******************************************************************//**
 * \file    canmap.hpp
 * \brief   Helper class to import message map from JSON
 *
 * \author Giovanni Musto
 * \date    2023/02/21
 * \copyright ItalDesign
 ********************************************************************/

#ifndef SOCKETCAN_CANMAP_HPP
#define SOCKETCAN_CANMAP_HPP

#include <cstdint>
#include <string>
#include <shared_mutex>
#include <unordered_map>
#include <sys/time.h>       // timeval
#include "canstruct.hpp"

/** \brief Obtain default tolerance (5%) from timing. */
#define GET_DEFAULT_TOLERANCE(timing)   (5 * (timing) / 100)

/** \copybrief canmap.hpp */
class CanMap
{
public:
    CanMap();
    CanMap(const CanMap&) = delete;
    CanMap& operator=(const CanMap) = delete;

    bool isMsgActive(const std::string &msg) const;
    bool getPeriodicReceive(const std::string &msg) const;
    bool setPeriodicReceive(const std::string &msg, bool periodicReceive);
    struct timeval getTimestamp(const std::string &msg) const;
    bool setTimestamp(const std::string &msg, struct timeval timestamp);
    void set_int_signal_value(const std::string &msg, const std::string &signal, uint64_t value);
    uint64_t get_int_signal_value(const std::string &msg, const std::string &signal) const;
    void set_signal_value(const std::string &msg, const std::string &signal, float value);
    float get_signal_value(const std::string &msg, const std::string &signal) const;
    e2e_err_t updateCounter(const std::string &msg, unsigned int &counter);
    e2e_err_t resetCounter(const std::string &msg);
    can_callback_t getSignalCallback(const std::string &msg, const std::string &signal) const;
    can_callback_t setSignalCallback(const std::string &msg, const std::string &signal, can_callback_t
         ↪ callback);
    can_callback_t removeSignalCallback(const std::string &msg, const std::string &signal);

    static const std::unordered_map<std::string, can_mex_t> &getMessage_map();
    static const std::unordered_map<std::string, ecu_map_t> &getEcu_map();

private:
    class MessageMap {
    public:
        std::unordered_map<std::string, can_mex_t> message_map;
        MessageMap();
        MessageMap(const MessageMap&) = delete;
        MessageMap& operator=(const MessageMap) = delete;
    };

    class EcuMap {
    public:
        std::unordered_map<std::string, ecu_map_t> ecu_map;
        EcuMap();
        EcuMap(const EcuMap&) = delete;
        EcuMap& operator=(const EcuMap) = delete;
    };

    std::unordered_map<std::string, sig_vals_t> values_map;
    mutable std::shared_mutex valuesMapMutex;

};

#endif //SOCKETCAN_CANMAP_HPP
```

Listing 3.2: Header file for the CanMap C++ class.

### 3.2.2 CanLib

The CanLib class is the core of the library, and handles almost all the remaining functions. Listing 3.3 on the following page shows the header file for the CanLib class.

First of all, it abstracts the communication layer, by providing the *initSocket()* and *closeSocket()* functions to handle the opening and closing of the socket, and the *sendMsg()* and *getMsg()* functions to simplify the sending and receiving of CAN messages.

The *handleCanMessage()* function is called after receiving a message, and checks if it has been received within the tolerance range (in case the message is periodic) and, for each signal contained therein, updates the last stored value, calculates the physical value via the conversion method, and calls any callback function previously registered with *setSignalCallback()*.

While *sendMsg()* allows the user to send an arbitrarily constructed message, *sendSingleShotMsg()* sends a known message given its name. The library keeps track of the 'current value' of each signal, allowing the user to change it via the *setStoredInternalValue()* function. When a known message is sent, it is automatically constructed from the stored values of its signals. This allows the user to instruct the library to send periodic messages, via the *startPeriodicSend()* function, and asynchronously update the values that need to be sent.

The *protectMessage()* function adds the counter and the calculated CRC to the message, when called with *test=false*, or checks if they match when called with *test=true*. At the moment, only the specific E2E profile used by Audi for their modern platforms has been implemented.

Then there are some *static* functions to get info about the messages, like *isMessageExtended()*, *isMessageCANFD()* and *isMessagePeriodic()*. *getMessageId()* and *getMessageName()* respectively return the CAN ID from the message name and vice versa, while *getSignalNames()* returns the list of signal names in a message.

The ECU map is accessed via the *getOutboundMessages()* and *getInboundMessages()* functions, that respectively return the messages an ECU sends and receives.

Other functions, such as *toPhysicalValue()* and *getValueLabel()*, handle the conversion methods.

The periodic send of messages is implemented in the *ThreadHandler* class. When *startPeriodicSend()* is called on a message, it checks if there's already a thread sending messages with the same periodicity, and if so, adds the message to the list; otherwise, it creates a new thread. The threads iterate over the list of messages to be sent, sending one after the other, constructing them from the stored signal values. The tolerances on the periodicities are kept very tight by using C++'s timing mechanisms.

```
1  /******************************************************************//**
2   * \file    canlib.hpp
3   * \brief   Utility library to handle CAN communication
4   *
5   * \author Giovanni Musto
6   * \date    2023/02/21
7   * \copyright ItalDesign
8   *******************************************************************/
9
10 #ifndef SOCKETCAN_CANLIB_HPP
11 #define SOCKETCAN_CANLIB_HPP
12
13 #include <cstdint>
14 #include <string>
15 #include <vector>
16 #include <atomic>
17 #include <thread>
18 #include <unordered_map>
19 #include <unordered_set>
20 #include <mutex>
21 #include <sys/time.h>       // timeval
22 #include <linux/can.h>      // canid_t
23 #include "canstruct.hpp"
24 #include "canmap.hpp"
25
26 #ifndef CANFD_FDF
27 #define CANFD_FDF   0x04
28 #endif
29
30 typedef struct canfd_frame canFrame;
31
32 /** \copybrief canlib.hpp */
33 class CanLib
34 {
35 public:
36     CanLib();
37     CanLib(const CanLib&) = delete;
38     CanLib& operator=(const CanLib&) = delete;
39     ~CanLib();
40
41     bool initSocket(const char *interface, bool canfd, int &errCode);
42     void closeSocket();
43     bool sendMsg(bool extended, canFrame &msg, bool rtr, int &errCode);
44     bool getMsg(bool &extended, canFrame &msg, bool &rtr, bool &error, int &errorCode, struct timeval*
        ↪ timestamp, struct timeval *timeout);
45     void enableErrMessages();
46     void handleCanMessage(bool extended, const canFrame &msg, bool rtr, bool error, int errorCode, struct
        ↪ timeval* timestamp);
47     can_callback_t setSignalCallback(const std::string &msg, const std::string &signal, can_callback_t
        ↪ callback);
48     can_callback_t removeSignalCallback(const std::string &msg, const std::string &signal);
49     float getStoredPhysicalValue(const std::string &messageName, const std::string &signalName) const;
50     e2e_err_t protectMessage(const std::string &msg, canFrame &frame, bool test);
51     e2e_err_t resetCounter(const std::string &msg);
52     bool startPeriodicSend(const std::string &msg);
53     bool startPeriodicSend(const std::string &msg, int timing);
54     bool stopPeriodicSend(const std::string &msg);
55     bool getPeriodicReceive(const std::string &msg) const;
56     void setStoredInternalValue(const std::string &messageName, const std::string &signalName, uint64_t value)
        ↪ ;
57     void sendSingleShotMsg(const std::string& msg);
58
59     static canid_t getMessageId(const std::string &messageName);
60     static std::string getMessageName(canid_t canId);
61     static const std::vector<std::string> getSignalNames(const std::string &messageName);
62     static bool isMessageExtended(const std::string &messageName);
63     static bool isMessageCANFD(const std::string &messageName);
64     static bool isMessagePeriodic(const std::string &messageName);
65     static const std::unordered_set<std::string> getInboundMessages(const std::string &ecuName, const std::
        ↪ string &busName);
66     static const std::unordered_set<std::string> getOutboundMessages(const std::string &ecuName, const std::
        ↪ string &busName);
67     static uint64_t getSignal(const uint8_t* frame, uint8_t startbit, uint8_t length, bool is_big_endian, bool
        ↪  is_signed);
68     static void setSignal(uint8_t* frame, uint64_t value, uint8_t startbit, uint8_t length, bool is_big_endian
        ↪ , bool is_signed);
69     static float toPhysicalValue(uint64_t target, const std::vector<compu_scale_t> &scales, bool is_signed);
70     static uint64_t fromPhysicalValue(float physical_value, const std::vector<compu_scale_t> &scales);
71     static const std::string& getValueLabel(uint64_t target, const std::vector<label_t>& labels, bool
        ↪ is_signed);
72     static int getBitMask(int start_bit);
73     static const char* get_E2E_err_msg(e2e_err_t err);
74
```

```
75  private:
76      class ThreadHandler {
77      public:
78          ThreadHandler() = delete;
79          ThreadHandler(const ThreadHandler&) = delete;
80          ThreadHandler& operator=(const ThreadHandler&) = delete;
81          ThreadHandler(CanLib* canlib, int timing);
82          ~ThreadHandler();
83          std::unordered_set<std::string>::size_type addMessage(const std::string &msg);
84          std::unordered_set<std::string>::size_type removeMessage(const std::string &msg);
85
86      private:
87          /** \brief Reference to CanLib instance. */
88          CanLib* canlib;
89          /** \brief Periodicity the thread uses to send the messages. */
90          int timing;
91          /** \brief Boolean to stop the thread. */
92          std::atomic<bool> cont;
93          /** \brief Thread handler. */
94          std::thread handler;
95          /** \brief Set of messages the thread has to send. */
96          std::unordered_set<std::string> messages;
97          /** \brief Mutex protecting messages. */
98          std::mutex messagesMutex;
99
100         void thread_handler();
101     };
102
103     /** \brief Flag telling if the socket has been initialized. */
104     bool  m_init;
105     /** \brief Socket number. */
106     int   m_socket;
107     /** \brief CanMap object. */
108     CanMap  canmap;
109     /** \brief Flag telling if we're using CAN-FD */
110     bool    m_canfd;
111     /** \brief Map storing one ThreadHandler for every periodicity we're sending messages with. */
112     std::unordered_map<int, ThreadHandler> thread_map;
113     /** \brief Mutex protecting thread_map. */
114     std::mutex threadsMutex;
115
116     static bool isValueInRange(double value, const range_t &range);
117     static double computePhysicalValue(double value, const std::vector<float> &num_coeffs, const std::vector<
            ↪ float> &den_coeffs);
118     static const std::unordered_set<std::string> getMessagesByDirection(const std::string &ecuName, const std
            ↪ ::string &busName, frame_direction_t dir);
119
120 };
121
122 #endif //SOCKETCAN_CANLIB_HPP
```

Listing 3.3: Header file for the CanLib C++ class.

## 3.3 Other components

Alongside the Python script and the C++ library, for this work I also developed two other components: a series of demo applications, in order to fully test the library, and some CMake files, to automate and speed-up the JSON generation and compiling.

### 3.3.1 Demo applications

Various test applications have been written during the development of the library, evolving with it in order to test more and more functionalities.

The first two applications were just meant to test the sending and receiving functionalities. Then, a synchronization test has been written, in order to make

sure that the data on both ends was coherent. This program was then expanded to support CAN FD and the E2E protection. When the library was expanded to support periodic messages, a periodic sender test application was written. Finally, a real world demo application was implemented, which will be discussed in chapter 4 on page 30.

### 3.3.2 CMake files

In order to orchestrate all the parts of the project, the code has been accompanied by a series of CMake files. CMake makes building the project for different platforms much easier, as I'll discuss in section 3.4.

Each component (the Python parser, the C++ library and the demo applications) has its own CMake file. The configuration of the Python script CMake (and thus of the entire project) requires a list of input ARXML files, and eventually a list of ECUs to include or exclude.

The library's CMake, on the other end, accepts a configuration flag to build a static or dynamic library, and also has a target to build the documentation with Doxygen, as the code has been annotated accordingly.

## 3.4 Deployment

Having developed a replacement for the software development phase of the V-model, the same need arises for the integration phase. Fortunately, there are some development boards available to fulfil this need. The one used in this work is the S32G by NXP, shown in fig. 3.1 on the next page. It features 18 CAN, 5 LIN, 1 FlexRay and multiple Ethernet ports, runs Linux and provides a complete development toolchain.

In order to deploy the software to the board, the first step is to compile the library and the application for the specific target. Fortunately, the manufacturer provides a toolchain that fully supports CMake, so the only step needed before compiling consists in running the environment configuration script provided by the manufacturer. In order to also run the Python parser in the same environment, its dependencies need to be installed the first time the toolchain is used[1]. At this point, configuring and running CMake will automatically build the library and applications for the development board, according to the given configuration.

To upload the software to the board, it's possible to use one of its Ethernet ports and connect to it via *SSH* while the system is running, or copy everything to its

[1]The Python parser's CMake doesn't need to be run in the cross-compilation environment, but doing so results in fewer steps. Therefore, it is recommanded to install *lxml*, the only dependency of the Python parser, in this environment.

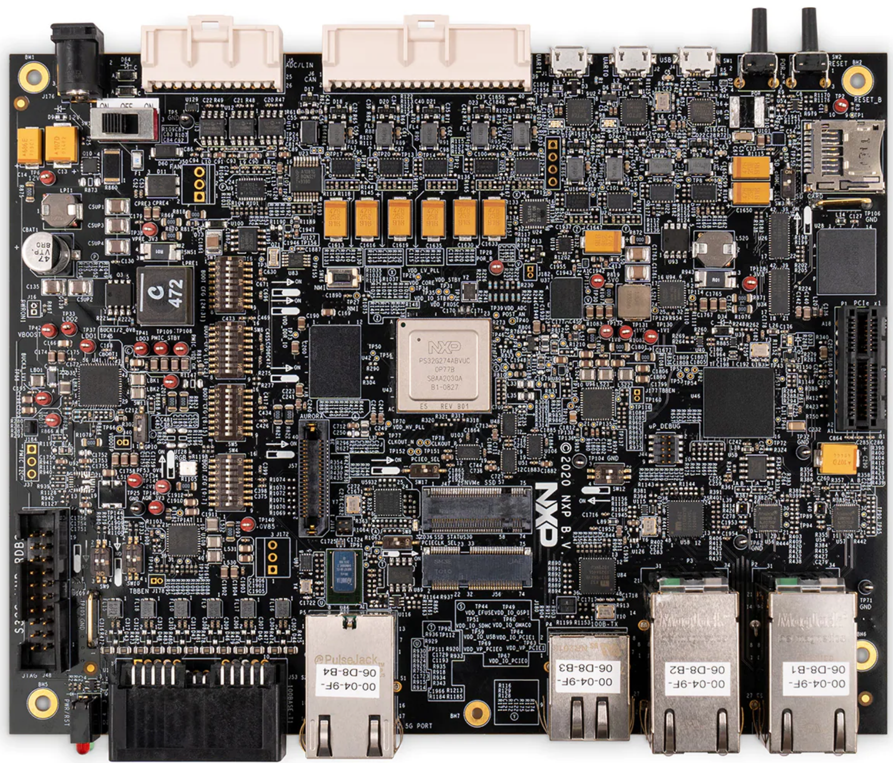

Figure 3.1: Picture of an S32G development board.

microSD card. The shared library can also be added as a Yocto recipe and built alongside the operating system.

Before running the software, the CAN ports need to be configured. The library avoids interfering with the CAN configuration, instead using the ports transparently, leaving the configuration to the user. The *ip* command line tool can be used to configure the ports. For example, running **`ip link set llcecan0 type can bitrate 500000 dbitrate 2000000 fd on`** will configure the CAN port called *llcecan0* as FD-capable, using use a bitrate of 500 kbps in CAN mode, and a bitrate of 2 Mbps during the bitrate-switching phase of CAN FD. Running **`ip link set up llcecan0`** will then bring up the interface with the given configuration. At this point, any software can use the *llcecan0* interface.

# Chapter 4

# Real world demo

In order to prove the validity of the V-model alternative, a real world application has been developed. The front lights ECU of a production car has been simulated and used to drive the headlamps from a 2020 Audi A3. This particular headlamp features matrix LEDs and requires specific signals to even turn on.

Unfortunately I can't show it here, but I had access to the CAN database for the front lights network. It wasn't in the ARXML format, so it needed to be converted, but apart from that, there was no difference between it and the test ARXMLs I used before.

Figure 4.1: Front view of a 2020 Audi A3.

Many CAN messages on this network use the E2E protection, and the headlamp will shut itself off if a specific periodic message is not received. All these aspects make the front lights ECU the perfect candidate to thoroughly test all the main aspects of the library.

The front lights demo application features a menu from which it's possible to select the desired option. The first thing to do is to enable the periodic send of the CAN messages, so that the headlamps can exit their error state (due to the lack of a specific message) and turn on. Then, the key status needs to be asserted, otherwise the lights will always stay off[1]. At this point, it's possible to drive the individual components of the headlamp: daytime running light, position light, low beam, matrix LED high beam and turn signal.

The demo works by sending only the messages needed to fully control the headlamps, and not all the messages that would be normally found on the network in a real vehicle. The largest part of the signals are sent with their default values, while some of them require to be set to specific values in order to turn the headlamps on. The menu options change the stored values of specific signals using the *setStoredInternalValue()* library function, while the threads sending the periodic messages handle the rest.

This real world implementation also provided some useful insights on the capabilities of the development board. Even though the scope of this work was just demonstrating that it is feasible to drive a series production component this way, without focusing too much on performance, the CPU load of the board and the deviation from the nominal timing of periodic messages were monitored. Those metrics turned out to be very useful, because they allowed me to optimize and improve the library (the last iteration uses almost one one-hundredth of the CPU power of the first one). It's important to note that those two paraments haven't been rigorously measured, since it wasn't the primary scope of this work, but I can report that the CPU load of the board stayed between 2% and 3%, with the timing of the periodic signals never deviating more than 2 ms from the nominal value when running the front lights demo. With higher loads, however, spikes in the timings that would need further investigation started to emerge.

[1] There are various ways the headlamps can be enabled, but turning the ignition key is the most obvious one.

# Chapter 5

# Conclusions and future developments

In the previous chapters, I defined a new approach to software development and integration, and established a development framework that proved to be applicable to real world scenarios. The S32G development board also proved to be more than capable of supporting one such application. With some refinements, this framework can be used in future projects of the company, such as new show cars or prototypes. As of today, the library is stable enough to to be used for a demo, but would need to be thoroughly validated in order to be used in a low series production vehicle. Given the ability to control off-the-shelf components, a vehicle could be fitted with components from different platforms, and have all talk together thanks to a central gateway implemented on a development board.

As for future developments, the main improvements that the C++ library would benefit from are: the support for other communication networks, such as LIN and Ethernet; the expansion of the E2E support to more profiles; a more automated approach to messages reception, perhaps paired with SocketCAN's ability to filter messages in kernel space. An expansion toward a service-oriented architecture would also be very interesting, perhaps using SOME/IP [21]. The Python script, instead, needs to be verified against some more real world designs and ARXML files generated by different tools.

# Bibliography


[1] Semiconductor History Museum of Japan. URL: https://www.shmj.or.jp/english/pdf/ic/exhibi739E.pdf.

[2] The MISRA Consortium Limited. *MISRA C:2023 Guidelines for the use of the C language in critical systems*. URL: https://www.misra.org.uk/.

[3] The MISRA Consortium Limited. *MISRA C++:2008 Guidelines for the use of the C++ language in critical systems*. URL: https://www.misra.org.uk/.

[4] AUTOSAR. URL: https://www.autosar.org/.

[5] Guillaume Sottas. *CanTp*. URL: https://github.com/Sauci/CanTp.

[6] Guillaume Sottas. *Xcp*. URL: https://github.com/Sauci/Xcp.

[7] Richard Haar. *AUTOSAR-CRC*. URL: https://github.com/richhaar/autosar-crc.

[8] kal102. *CanNm*. URL: https://github.com/kal102/CanNm.

[9] AUTOSAR. *ARXML Serialization Rules*. R22-11. URL: https://www.autosar.org/fileadmin/standards/R22-11/FO/AUTOSAR_TPS_ARXMLSerializationRules.pdf.

[10] AUTOSAR. *E2E Protocol Specification*. R22-11. URL: https://www.autosar.org/fileadmin/standards/R22-11/FO/AUTOSAR_PRS_E2EProtocol.pdf.

[11] *ISO 11898-4:2004: Road vehicles – Controller area network (CAN) – Part 4: Time-triggered communication*. International Organization for Standardization. 2004. URL: https://www.iso.org/standard/36306.html.

[12] *ISO 11898-3:2006: Road vehicles – Controller area network (CAN) – Part 3: Low-speed, fault-tolerant, medium-dependent interface*. International Organization for Standardization. 2006. URL: https://www.iso.org/standard/36055.html.

[13] *ISO 11898-1:2015: Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling*. International Organization for Standardization. 2015. URL: https://www.iso.org/standard/63648.html.

[14] *ISO 11898-2:2016: Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit*. International Organization for Standardization. 2016. URL: https://www.iso.org/standard/67244.html.

[15] Karen Parnell. "Put the Right Bus in Your Car". In: *Xcell Journal* Winter 2004 (). URL: https://www.rpi.edu/dept/ecse/mps/xc_autobus48(CAN).pdf.

[16] The kernel development community. *SocketCAN – Controller Area Network*. URL: https://www.kernel.org/doc/html/latest/networking/can.html.

[17] Luca Valentini. “A methodology for the design of anautomotive network architecture”. MA thesis. Politecnico di Torino, Apr. 2022. eprint: http://webthesis.biblio.polito.it/id/eprint/22797.

[18] Vector Informatik GmbH. *PREEvision – E/E Engineering Environment*. URL: https://www.vector.com/int/en/products/products-a-z/software/preevision/.

[19] Stefan Behnel. *lxml – XML and HTML with Python*. URL: https://lxml.de/.

[20] Niels Lohmann. *json*. URL: https://github.com/nlohmann/json.

[21] Dr. Lars Völker. *Scalable service-Oriented MiddlewarE over IP (SOME/IP)*. URL: https://some-ip.com/.