POLITECNICO DI TORINO

Master's Degree in MECHATRONIC ENGINEERING

Master's Degree Thesis

Serial Data Parsing for High-Fidelity GNSS Integration in ETSI C-ITS Vehicular Networks

Supervisors

Prof. Claudio Ettore CASETTI
Dott. Marco RAPELLI
Dott. Francesco RAVIGLIONE

Candidate

Mauro VITTORIO

APRIL 2025

Summary

This thesis presents the design, implementation, and validation of a robust, open-source approach to Cooperative Intelligent Transport Systems (C-ITS), with special emphasis on integrating Global Navigation Satellite System (GNSS) data from Real Time Kinematics (RTK) receivers. By leveraging hardware such as the ArduSimple RTK2B board in conjunction with the u-blox ZED-F9R module, the system captures high-fidelity positioning information, indispensable for safety-critical and efficiency-driven vehicular applications.

The framework centers on the OScar (Open Stack for Car) project, an ongoing C++ implementation of the ETSI C-ITS standards, designed to expedite adoption and reduce costs through open-source distribution. A key focus is the development of a serial data parser for UBX and NMEA protocols, which processes the incoming GNSS messages and encapsulates them into the V2X C-ITS Services such as the Cooperative Awareness Basic Service (CABS) and the Vulnerable Road Users Awareness Basic Service (VBS). By strictly adhering to the layered structure of C-ITS comprising the Application, Facility, Network/Transport, Access, and supplementary Management and Security layers, OScar ensures modular, interoperable message handling. Field experiments and simulations verify the parser's performance as well as the real-time feasibility of populating CAM structures with accurate position, speed, and heading data.

In turn, these messages enable low-latency hazard detection, traffic flow optimization, and advanced driver-assistance features. Through comparative performance testing against existing tools like GPSD, the thesis also underscores the advantages of high-frequency updates in safety scenarios.

By merging open-source flexibility, established communication standards, and centimeter-level positioning precision, this work demonstrates how the OScar framework can serve as a cost-effective, future-ready solution for C-ITS deployments. The resulting software stack sets a significant milestone toward widespread vehicular connectivity, paving the way for enhanced road safety and coordinated transport ecosystems.

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Prof. Claudio Casetti, Dr. Marco Rapelli, and Dr. Francesco Raviglione, for their invaluable guidance and support throughout this thesis. I am deeply thankful for the precious knowledge and insights I have gained from each of them.

An infinite thank you to my beloved family. To my father, Angelo, for always putting our family first and facing each day with unmatched grit. To my mother, Giuseppina, for gifting me the power of empathy and showing me how it can change the lives of those around me for the better. To my brother, Alberto, for being a constant presence in my life, someone I can always learn from and grow with, together shaping each other's paths.

To Najla, a precious gift life has given me. Thank you for always loving me even when it wasn't easy, for being a safe harbor through every storm, and for standing by me unconditionally. I will never be able to express my gratitude enough to you.

To Vanessa, for teaching me, day by day, how to fight through life by valuing the little, yet powerful things. For teaching me how to love and for surprising me with your strength, proving that even the hardest challenges can be faced with grace when the right person is by your side.

To all my beautiful friends, Daniele, Simone, and Vincenzo, thank you for sharing this journey with me. You helped me feel understood and supported, making the effort of studying lighter. Thank you for choosing to spend time with me, for studying together, and for helping me build an effective routine to reach our goals. To Giuseppe, for teaching me to never give up, to always find a way to break through the walls that block the path to the top, with relentless determination. To Marco, for teaching me patience and to look beyond the surface when trying to understand others.

To all my friends from the canteen, thank you for being a bright light when everything felt dark and heavy. You showed me that even a friendship just beginning can be deep and immensely valuable.

Finally, to my Instagram community, thank you for giving me the opportunity to live my dream: sharing my life in the service of others, helping them and helping myself to always keep pushing forward.

Thank you from the bottom of my heart.

Table of Contents

List of Figures

Acronyms

GNSS Global Navigation Satellite System

GPS Global Positioning System

GLONASS GLObal NAvigation Satellite System

QZSS Quasi-Zenith Satellite System

LBS Location-Based-Services

GIS Geographic Information System

RTK Real Time Kinematics

V2X Vehicle to Everything

ITS Intelligent Transport Systems

C-ITS Intelligent Transport Systems

VANET Vehicular Ad-hoc NETwork

MANET Mobile Ad-hoc NETwork

OEM Original Equipment Manufacturer

OBU On Board Unit

RSU Roadside Unit

CABS Cooperative Awereness Basic Service

VRU Vulnerable Road Users

VBS Vulnerable Road Users Awareness Basic Service

CPS Collective Perception Service

CAM Cooperative Awareness Message

DENM Decentralized Environmental Notification Message

IVIM Infrastructure to Vehicle Information Message

EV Electric Vehicle

AT Authorization Ticket

BTP Basic Transport Protocol

GN GeoNetworking

MAC Medium Access Control

OFDM Orthogonal Frequency-Division Multiplexing

C-V2X Cellular Vehicle-to-Everything

VAM Vulnerable road users Awareness Message

UART Universal Asynchronous Receiver-Transmitter

USB Universal Serial Bus

NMEA National Marine Electronics Association

(RTCM) Radio Technical Commission for Maritime Services

(NTRIP Networked Transport of RTCM via Internet Protocol

4G NTRIP 4G Networked Transport of RTCM via Internet Protocol

VDP Vehicle Data Provider

ASN.1 Abstract Syntax Notation One

TRACEN-X Telemetry Replay and Analysis of CAN bus and External Navigation data

Chapter 1

Introduction

1.1 Global Navigation Satellite System

The desire for a global positioning technology that could assist in navigation tasks has always been present in the human innovation process. From the use of the sun and the stars to the sophisticated algorithms that find the best route on a graph, navigators have always sought tools and strategies to improve and optimize travel routes.

The first prompt for a modern technology that would reach such objective was presented in 1957 with the launch of the *Sputnik* satellite by the Soviet Union and the analysis of its signal that showed how the Doppler effect could be leveraged to predict the satellite's orbit and the position of the ground stations. From this point on, the evolution of technology was nourished by the most influential countries in the world.

Nowadays the Global Navigation Satellite System (GNSS) defines a constellation of 24 satellites orbiting the Earth, providing precise positioning with complete global coverage. GNSS is made up of satellites from several countries that have developed their own positioning solutions: the United States (GPS), Russia (GLONASS), Europe (Galileo), China (BeiDou) and Japan (QZSS). One crucial GNSS feature to be underlined is that these satellites travel in a semi-synchronous orbit lasting 12 hours and are positioned in a way such that six of them are always visible all the time, from any point on the planet.

GNSS plays a crucial role in the engineering and realization of Location-Based Services (LBS), delivering precise and low-latency positioning in order to meet the requirements of applications in both the public and private sectors. [28] In particular for the automotive industry, GNSS takes advantage of its vast interoperability and integrates with Internet networks and complementary systems to further enhance its capabilities, A fundamental example is the Geographic Information System (GIS), which enables efficient positioning and map representation in an optimal way, making navigation possible.

The integration of complementary technologies has moved GNSS beyond traditional orientation and navigation tasks, bringing high-precision positioning solutions into diverse

fields. One of the crucial technological milestones that stands out is in this matter is Real Time Kinematics (RTK), that provides centimeter-level accuracy, largely improving LBS performances if a broad range of sectors, for example in agriculture, making automated seed planting, crop management, and other advanced applications both feasible and efficient.

Real Time Kinematics

Real-Time Kinematics (RTK) provides accurate corrections using carrier-phase measurements of satellite signals, reaching the higher accuracy needed in the automotive field and similar. RTK leverages the use of a fixed base station that generates correction data calculated using the signals obtained from the satellites and sends them to the GNSS receiver, providing precision up to the order of centimeters.

The project unfolded in this thesis makes extensive use of this technology in order to target and solve the problem of vehicular precise positioning and network communication, supporting the practical use of a complete and open implementation of the ETSI C-ITS standards, that constitutes the core of the OScar project.

The hardware used for the scope of this thesis is based on the ArduSimple SimpleRTK2B GNSS receiver in conjunction with the u-blox ZED-F9R module (both will be discussed in detail in chapter 2: section 2.1), with the goal of reaching the optimal trade-off between performance, precision, costs, and usability, to exploit all the services provided by the OScar project, thus trying to equip any car with the capacity of being able to interact with other cars and road users via V2X (Vehicle to Everything) communication.

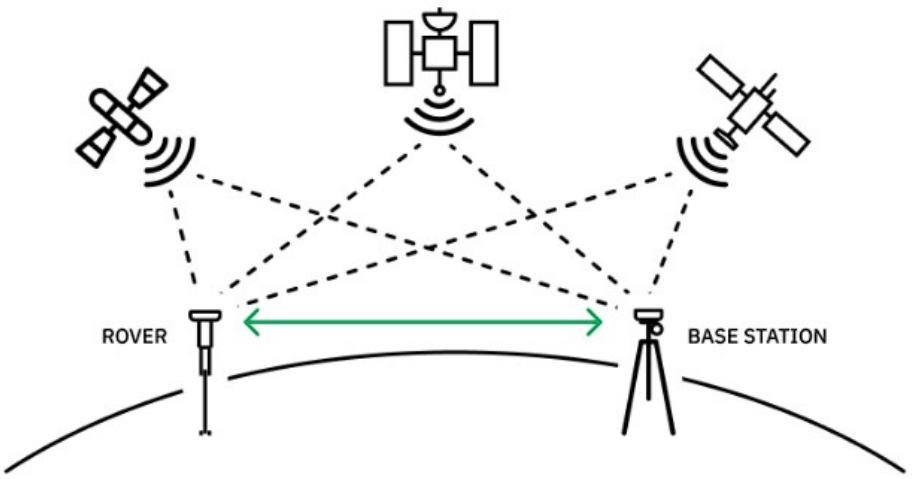

Figure 1.1: Real Time Kinematics [26]

1.2 Cooperative Intelligent Transport Systems

Intelligent Transport Systems (ITS) can be defined as a broad spectrum of standardized applications and technologies that provide diverse enhancements in the transportation networks area, greatly improving safety, sustainability and quality of vehicle trips.

ITS places its anchors in Vehicular Ad-hoc Networks (VANETs), a specific type of Mobile Ad-hoc Network (MANET) that permits vehicle interfacing and communication, favoring characteristics like dynamism, high mobility and real-time latencies. The key factor in VANETs is their decentralization, which enables the network to be flexible and adaptive to rapid changes in topology, configurations, and resources involved. In this scope, the composition of a vehicular network can involve various actors:

- **Vehicles** equipped with OEM (Original Equipment Manufacturer) built-in technology for vehicular communication or with custom OBU (On-Board Unit) with specific software (e.g. OScar).
- **Roadside infrastructure Units (RSUs)** electronically equipped in order to be part of the vehicular network (e.g. intelligent traffic lights).
- **Pedestrians and cyclists** that use specific software and/or wearable devices on smartphones and can be included in the network and more.

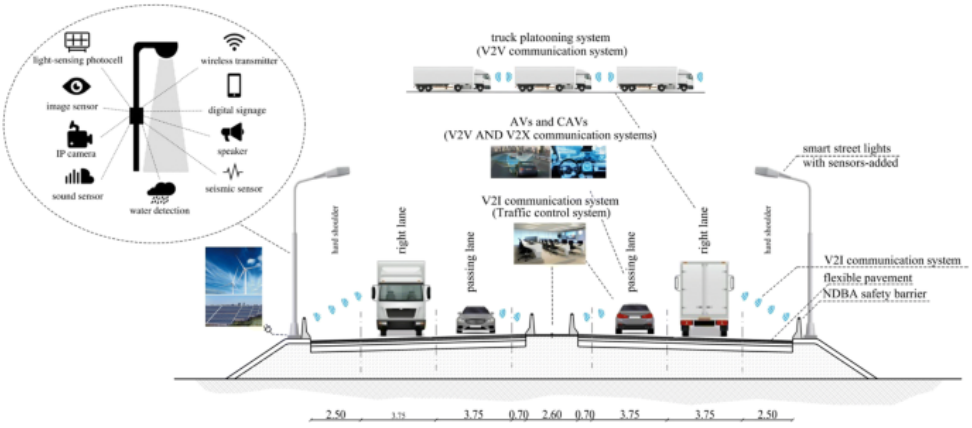

Figure 1.2: Intelligent Transport Systems complete overview [22]

The adoption of ITS technologies holds many positive outcomes: from improved safety and transport efficiency to a very low environmental negative impact, enabling intercommunication between vehicle and realizing the so called Cooperative Intelligent Transport Systems (C-ITS).

In the road safety realm, communication between vehicles is crucial to reach a new level of safety and crash prevention, for example, warning the driver about hazards or any important event happening on the upcoming road section that is worth signaling. Furthermore, a connected car can inform the driver in time about an infraction that occurred at the next intersection, by exchanging specific safety messages with the road

infrastructure and last but not least, the driver can always be up to date in matter of road works, deviations or any major event that can significantly influence the trip.

Intelligent Transport Systems also play a large role in the road efficiency domain, vastly improving both the car and the infrastructure end results. Some key examples include:

- **Improved route planning and reduced traffic congestion**: as an outcome of the continuous driver information process carried out by C-ITS technologies.
- **Efficient parking management**: combining gnss data and infrastructure data it is possible to find in advance the free parking spot, saving precious time to the driver
- **Fuel consumption optimization**: Leveraging both the capabilities of the driver assistance systems and the C-ITS it is possible to realize an efficient driving path, suggesting an optimal velocity to the cruise control, in order to minimize wait and idle time ad red lights.

This last topic reflects also the benefits that C-ITS offers to the environment: a greatly improved vehicle efficiency and thus a significative emissions down cut, moving always forward to a more optimized driving experience.

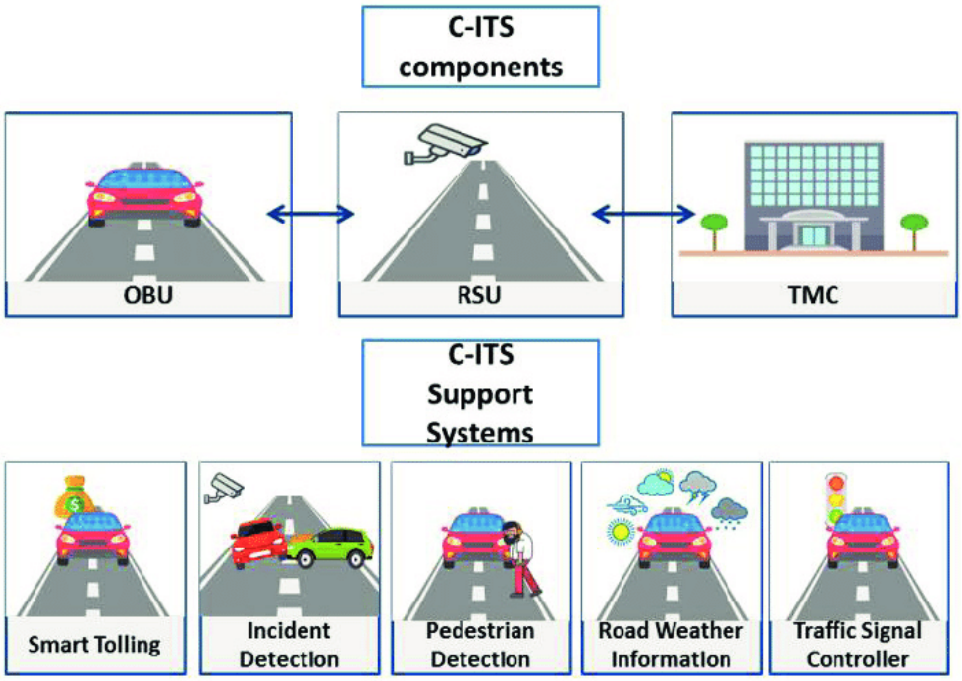

Figure 1.3: C-ITS scenarios [24]

Cooperative Intelligent Transport Systems have been engineered with a foundational layered structure, recalling the ISO/OSI model used for networking, that provides simple and efficient data handling, interoperability and modularity, with the intent of building a stack that can both have the efficiency needed by vehicular applications and the scalability

for future improvements and additional rules.

The layered structure of C-ITS is composed of four key layers: Application, Facility, Network and Transport and Access. Moreover, two additional layers that augment and complete core composition are present: Management and Security.

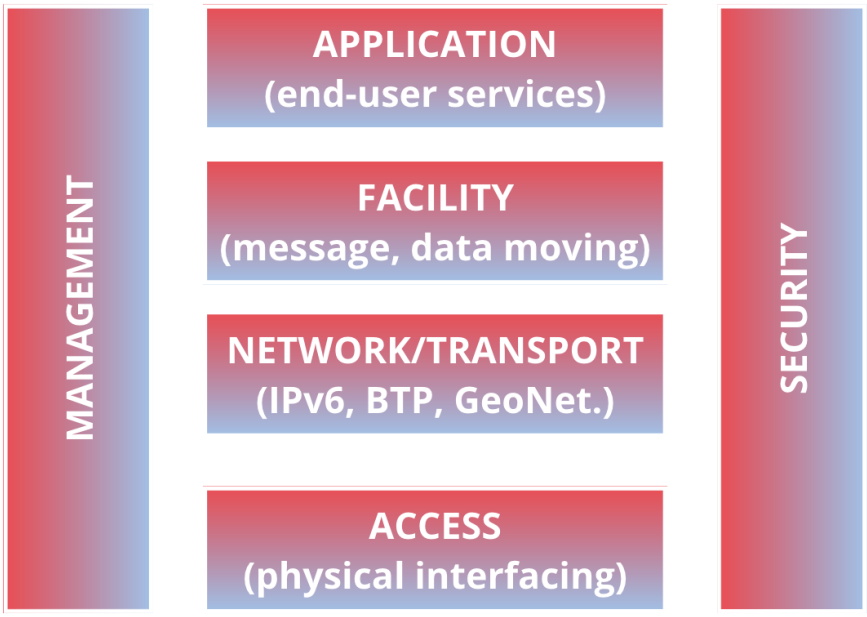

Figure 1.4: C-ITS Layered Structure [24]

Application Layer

The Application Layer in Cooperative Intelligent Transport Systems (C-ITS) represents the upper layer that directly communicates with the user and is responsible of implementing the final applications that will make use of the lower layers as communication interface. Its main role is to connect the end user with the communication stack that allows the car to be part of the network. As vehicles become increasingly connected and autonomous, the role of the Application Layer becomes fundamental in order to realize a fast and efficient communications between the car and the surrounding network peers.

Unlike traditional networking architectures, where the application layer primarily serves data processing purposes, the Application Layer in C-ITS is closely embedded within the operational mechanics of intelligent transport systems. It not only acts as data transmitter but encapsulates also critical features like message integrity, and communication actors verification, delivering sensitive services for the final user, for example: cooperative driving and hazard detection.

In addition, the application layer has been standardized with the intent of ensuring

maximum interoperability between different regions and countries rules, different manufacturers and vehicle typologies. All the messages handled by the application layer have a precise structure that guarantees consistency, security and process efficiency across all the possible implementations of the standardized rules.

Facility Layer

The Facility Layer sits between the Application and the Network and Transport layers providing essential data handling, message processing, and service management functions for vehicular communications.

The main responsibility of this layer is data processing, aggregation, and storage. Without the facility layer, the messages exchanged between vehicle and infrastructure cannot be correctly processed and would remain raw data that travel the network without the possibility to be useful to the application layer, in order to implement the V2X services.

As defined in [5], the facility layer provides multiple useful services for a large variety of applications and contains relevant sub services, including:

- **The Cooperative Awareness (CA) Basic Service** that generates and handles Cooperative Awareness Messages (CAMs).
- **The Decentralized Environmental Notification (DEN) Basic Service** which manages Decentralized Environmental Notification Messages (DENMs).
- **The Local Dynamic Map (LDM)** which stores and manages real-time situational awareness data keeping track of the sorroundings, leveraging precise and low-latency external sensor data.

Furthermore, the facility layer provides specialized services like Collective Perception, Vulnerable Road Users Awareness, and Infrastructure-to-Vehicle Information (IVI) Services. These services are defined in such a way that data can be logically organized and ready to be used, with the intent of providing the application layer a meaningful information representation.

Network and Transport Layer

In C-ITS, the Network and Transport layer acts as a connective pivot that allows for all the main services that compose the architecture, for example: Cooperative Awareness, Vulnerable Road Users Awareness and Collective Perception to realize data sending and receiving across the network. In particular, this layer serves as a bridge between the upper Applications and Facility layers, which will interpret and effectively use the data, and the lower Access layer that will be responsible for handling the physical and data-link part of the architecture.

Understanding how the Network and Transport layer performs its operation reveals all the advantages that characterize the C-ITS architecture, which contribute to the goal

of obtaining safety and traffic efficiency services among all the entities in the road environment.

The main feature of the Network and Transport layer is the combined use of GeoNetworking and the Basic Transport Protocol (BTP), in order to provide optimal networking solutions based on the specific needs of the road environment. GeoNetworking is defined and standardized by the European Telecommunications Standard Institute in the ETSI EN 302 636 family of standards [9] and its main duty is to specifically address vehicular networking requirements such as high mobility and position-based information exchange.

In order to accomplish this goal, GeoNetworking relies on geographic position data delivery instead of a classical address-based IP networking approach, letting senders specify a communication area of relevance so that the network layer routes or broadcasts information within that area of interest, communicating with the road agents included in that area. This is very useful when realizing awareness services, in particular safety-critical scenarios such as collision avoidance and hazards warnings, where the only interested recipients are the road agents that fall in the immediate vicinity of the vehicle.

By leveraging this location-based addressing approach, GeoNetwork is able to avoid using all the data necessary for addressing and re-addressing messages recipients, performing an optimal message dissemination. It can operate in single-hop mode, sending a message only to neighboring vehicles within range or in multi-cast mode, where a message can be sent over multiple vehicles or road side units until they reach the intended destination.

The main operating tool used by GeoNetworking is the Basic Transport Protocol. Introduced and standardized in [14] with the intent of realizing a lightweight transport procedure specifically adapted to the vehicular networks and to be used for broadcast or geocast scenarios. BTP widely simplifies and lightens the data transport process, removing many unnecessary overheads that would load up and slow down the process if used in a TCP/IP manner.

Access Layer

The Access layer represents the foundation of the entire vehicular network communication process, providing both the physical and the medium access tools needed to support the upper parts of the model and to correctly interface to the lower hardware.

In Europe, the vast majority of this layer is defined and standardized by ETSI, in particular by the ETSI EN 302 663 standard [12] which denotes the physical and MAC (Medium Access Control) characteristic of the layer, operating in the 5.9 GHz band.

This technique known as ITS-G5 is, in fact, the most adopted in Europe and sets its foundations on the IEEE 802.11p standards deriving from the 802.11b/g/n standards used for classical Wi-Fi networks.

In ITS-G5, message exchange typically occurs by distinguish between critical message and less time-sensitive data, making use of specific channels, respectively the Control Channel (CCH) and the Service Channels (SCH). Each channel has a 10 MHz bandwidth around the 5.9 GHz frequency and leverages a specialized version of the Orthogonal Frequency-Division Multiplexing (OFDM), specifically engineered in order to perform at its best in networks with very rapidly changing characteristics, such as in the automotive environment. [7]

Another important interface is Cellular V2X (C-V2X) over the PC5 sidelink. [10] Introduced and stardardized by 3GPP (Third Generation Partnership Project), it has been adapted to be used in direct short-range communication by adapting some part of LTE (Long Term Evolution) and, in more modern versions, of 5G NR (New Radio) radio protocols.

Even if this approach has many differences with ITS-G5, such as a more structured scheduling process in order to better handle the case when multiple vehicle try to access a channel, it works to deliver the same goals: fast broadcast messages dissemination under real-time and highly variable circumstances, priming efficiency and preserving the transmission quality.

In summary, the main feature of the Access layer is the capability to handle raw and real-time transmission and reception demands, providing a solid foundation to all the important and safety-critical services that sit on top of it. By balancing channel usage, coordination, and access control, it ensures that all data coming from the upper layers can reach their intended destination within a significantly high rate of success, remaining a key point of all the standardized technology as the vehicular networks domain continues to grow.

Management and Security Layers

In the ETSI C-ITS architecture, the Management and Security layers play a crucial role, maintaining and supporting the quality of communication, ensuring that every part of the process complies with the defined policies and regulations, guaranteeing its trustworthiness.

The Management and Security layers typically operate in a supportive way, instead of being responsible for a concrete section of the C-ITS stack, but their influence spans throughout all of it, determining and influencing how any kind of device present in the network, whether vehicles, roadside units, pedestrians and more, interact and communicate among the network.

The Management layer actualizes the task of being the principal operational overseer of the network entity, as well as being responsible for the correct configuration of it, concerning the network's characteristics. At a high level, it corresponds directly with the

Network and Transport, Facility and Access layers, as well as with the Security layer, to dictate policy adherence and maintain a consistent runtime environment.

These tasks are accomplished through formal interfaces, described in detail in the ETSI TS 102 723 specification document, that is divided in several part, starting from the layered architecture structure [8] of the C-ITS stack.

The above specifications are applied as instructed by the ETSI EN 302 665 standard, which dictates how the Management layer collets information, performing information and network status logging and communication flow adjustments by monitoring and modifying all the parameters involved in the process.

The Security layer, as briefly described earlier, provides the cryptographic basis to ensure message integrity and, when necessary, confidentiality. The layer's operational flow is heavily founded on a public key encryption approach and on the use of security certificates. This operational method, described in detail in [17] and [20], imposes a custom security level, based on messages types.

Simple environment-awareness messages, such as Cooperative Awareness Messages, use general security authorizations that are easier to handle. In contrast, more safety-critical messages, for example the Vulnerable Road User Awareness Message require special permissions, encoded in Service Specific Permissions (SSP), so that the security layer can attach signatures. These signatures are then used by the receiver to verify the integrity of both the message and its sender. To ensure privacy, the security layer manages also the so-called pseudonym certificate which rotate from message to message, keeping safe the identity of the sender and making sure that a vehicle path cannot be tracked by an external observer. In case of sensible data content, along with certificates, encryption comes into play as defined in detail in [19] and uses public keys in order to exclude any possible data content breaching.

In conclusion, the Management and Security layers work together in the full C-ITS stack ensuring full direction and control of the overall communication process by managing timing, environmental conditions and special messages needs, applying encryption and privacy protection measures when needed, making the system run smoothly and securely.

1.3 The OScar Project for ITS Applications

The main antagonist to the rise of interconnected vehicular technologies has always been the slow increase of the rate of the actualization of the technology itself, in terms of end user adoption. This phenomenon, also defined as "penetration rate", indicates all the obstacles that modern vehicular networks technologies face in terms of end adoption. Some examples can be given by the restrictive costs to afford a modern vehicle that equips vehicular communication hardware and software as an OEM feature or the business choices

that manufacturers take against vehicular technologies, justified by the risks in terms of security or other engineering or market factors, and lastly the most important: the wide presence of vehicles that do not equip the necessary assets in order to benefit from the vehicular network's applications.

The Open Stack for car (OScar) project is an open source and lightweight solution to the penetration rate problem in vehicular networks [2] and at the time of this writing it provides a complete implementation of the most important vehicular networks services standardized by ETSI including:

- **Cooperative Awareness Basic Service (CABS)**: defined in [11], provides useful information about vehicles dynamics and road presence.
- **Vulnerable Road Users Awareness Basic Service (VBS)**: defined in [15], accounts for specific rule to follow in order to handle vulnerable road user's protection, interaction and communication tasks.
- **Collective Perception Service (CPS)**: defined in [21], allows vehicle to exchange information about environment perception using specific ambient sensor to scan for obstacles or hazard and diffuse the advice to other vehicles.

The project's mission is to expand in the upcoming future, introducing network and security improvements and including support to other types of vehicular networks such as:

- **Decentralized Environmental Notification Messages (DENMs)**: defined in [13], are fundamental for safety enhancement and widely used in V2V (Vehicle-to-Vehicle) and V2I (Vehicle-to-Infrastructure) communication, providing immediate notifications about road accidents and adverse weather conditions, promptly informing all the vehicles nearby.
- **Infrastructure to Vehicle Information Messages (IVIMs)**: defined in [16], regulate the communication rule for the infrastructure to communicate with road agents, resulting essential for specific scenarios where detailed data about the road profile are crucial, such as autonomous driving.
- **Electric Vehicle Charging Spot Notifications (EVCSNs)**: defined in [6], provide real-time and up-to-date data about the presence and the current state of charging spots for electric vehicles, strongly supporting the unfolding electric vehicles diffusion.

The support for security headers and certificates is being developed according to [18] in order to provide resistant and resilient vehicular messages exchange, with a strict approach that spans across the main security requiring principles:

- **Integrity**: ensure that the message is not modified in any way.
- **Privacy**: protect both communication's part identities while maintaining traceability and accountability of the message.
- **Non-repudiation**: impede the receiver to deny the message.

- **Authentication**: Guarantee the legitimacy of both communication parts.

This approach will be realized introducing security headers that will include a digital signature with a certificate of authenticity and a timestamp string, to ensure the individuality and the trustworthiness of the message. Along with ETSI-compliant certificates, vehicle security will be handled using Authorization Tickets (AT) and specific certificate validation procedures, offering OScar a robust and reliable stack, to tackle one of the main reason why OEM tend to be skeptical about implementing V2X technologies.

The project is intended to be manageable, modular and immediately ready to land on real hardware thanks to its open-source nature and this thesis will focus on its Cooperative Awareness Basic Service implementation, in particular, on the integration into OScar of the software that provides accurate serial data parsing, in order to process data coming from the GNSS receiver and to encapsulate them correctly in the Cooperative Awareness Messages (CAMs).

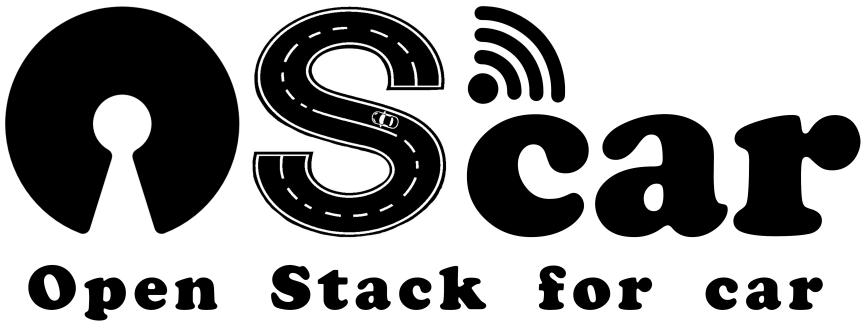

Figure 1.5: OScar project logo [2]

Project Background and Motivations

Since the first appearance of modern vehicular interconnection technologies, the need for accelerating the standardization and adoption process has been very strong and many solutions have been proposed.

In 2016 the European Commission published a strategy documents with the goal of accelerating the C-ITS diffusion, choosing to focus on the communications by both vehicles and infrastructure, which were considered the topics of major priority so that to boost the diffusion of C-ITS. Another attempt was made in 2019, with the proposal of a juridical framework that aimed at the coordinate adoption of C-ITS, which was dismissed shortly after though due to a missed agreement on the way to realize the vehicular connection, in particular between the use of Wi-Fi networks or cellular network (C-V2X).

A more successful example can be found in C-ROADS: An European project that has the scope of nourishing the C-ITS and V2X development in various countries of Europe, including Italy. It also concentrates on integrating the new instrumentation with the existing infrastructure, trying to find the correct innovation trade off between renewing

and adapting. This last paradigm is what leads the OScar project as well, giving it the opportunity to bring a very cost-reduced, efficient and most importantly practical solution to equip as many mobile road agents as possible with a device that provides the out-of-the-box foundational vehicular hardware and software, tearing down the development time of V2X end-applications.

OScar role in C-ITS Layered structure

It is crucial to say that Cooperative Intelligent Transport Systems come with many difficult challenges to face, concerning sensitive topic like:

- **Efficient standardization**: the standardization path is usually very complex and long, creating a significative temporal gap between the posing of the rules that define a new technology and the actual implementation and utilization of it, undermining its intrinsic potential and performance.
- **Security**: the biggest downside of a connected vehicle is the very high risk in terms of cybersecurity that can quickly escalate and easily become a tangible threat to the owner and passengers of the car and to all the other road users and infrastructures as well.
- **Privacy**: any sensitive data about positioning and all other vehicle characteristics can be sensible to malicious attacks, so it is crucial to prevent these hazards, implementing the right solutions.
- **Overall costs**: can be an important obstacle to the development of the C-ITS infrastructure since it is required to install many hardware and software resources in large urban areas. This issue can also propagate into other technological slowdowns due to the fact that the installation contract have to be handled by the government.
- Public response: it can be difficult to transmit the knowledge required to fully take advantage of the technology by the public, particularly in a context that faces many different end users and a very low rate of technological renewal and innovation.

The project presented in this thesis has the goal of easing and speeding up the process that seeks to bring an efficient solution to these problems.

By providing a full open source implementation of the European Telecommunications Standard Institute (ETSI) C-ITS stack, the OScar project effectively contributes in closing the gap created by the issues mentioned above, bringing a powerful, stable and secure way to make any car actively part of a vehicular network without any specific knowledge by the end user and keeping the costs as low as possible by leveraging the OpenWrt free Linux distribution for embedded systems.

1.4 Serial Data Parser

One of the main challenges for the OScar project has always been retrieving GNSS data from a reliable external source, in order to process and encapsulate them into the C-ITS stack. In the earlier development stages, the only external data source has been *gpsd*, a software daemon that collects and handles GPS data in Unix-like systems. This design choice was very effective in terms of speeding up the testing and refining phases, vastly improving the development of the standard's implementation basics, but it was also limiting for the complete goal of the project, which aims to process GNSS data in a more complete, precise and efficient way.

To address these limitations and to fully meet the implementation requirements, a dedicated data gathering and parsing software module has been introduced, augmenting the capabilities of OScar to directly communicate with dedicated GNSS hardware like the ArduSimple SimpleRTK2B, having access to more precise and more reliable data, along with a larger configurations and operations options.

The following chapter and all this thesis project will present in technical detail the data handling process, highlighting the advantages of the serial data parser contributions to a more robust and standard-compliant vehicular network ecosystem.

Chapter 2

Serial Communication and GNSS Data Parsing

2.1 Serial Interfacing with SimpleRTK2B Board

The SimpleRTK2B board is a very precise, lightweight and cost effective serial device that hosting the ublox ZED-F9R GNSS sensor fusion module, it is capable of providing centimeter-level positioning data thanks to the Real Time Kinematics technology. Moreover, thanks to its sensor fusion capabilities, the module provides dynamics and gyroscopic measurements such as lateral and vertical accelerations and acceleration rates, useful to analyze in depth the behavior of the vehicle.

The key technology that supports this process is represented in a high level realm by the specific message protocols that expose the structure of data and all its related information. In a lower hardware level, the communication is executed by data transmission interfaces, mostly in a serial manner, such as the Universal Asynchronous Receiver-Transmitter (UART) and the Universal Serial Bus (USB).

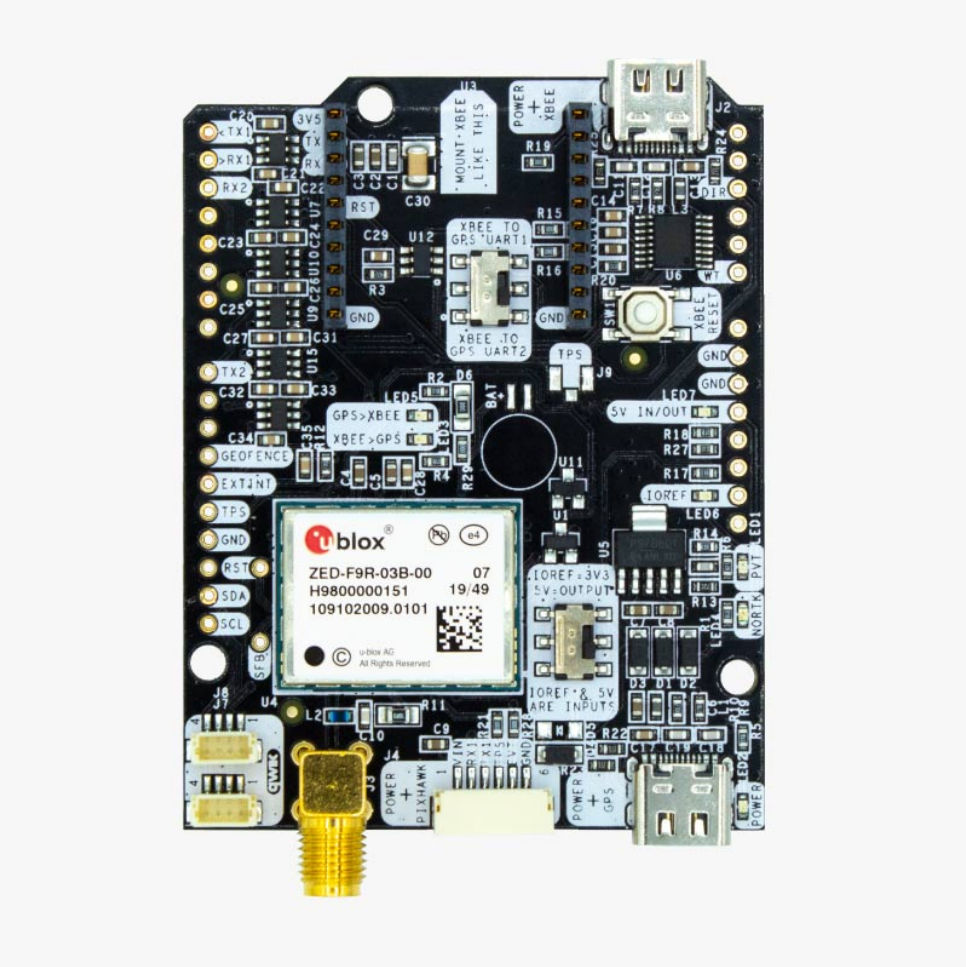

Figure 2.1: ArduSimple SimpleRTK2B with the ublox ZED-F9R module [24]

2.1.1 Relevant Board Communication Protocols

In default working settings, the device operates generating National Marine Electronics Association (NMEA) sentences to communicate positioning, status and time data in an organized ASCII-based format, as well as using proprietary UBX (ublox) messages that have a more sophisticated form but offer far more data ranges and customization possibilities.

When used with Real Time Kinematics settings, additional protocols such as Radio Technical Commission for Maritime Services (RTCM) can be exploited along with other communications mediums such as cellular data networks, obtaining high precision performance thanks to the fast Differential GNSS (DGNSS) data correction applied via LTE, also known as 4G Networked Transport of RTCM via Internet Protocol (NTRIP) that will be introduced and described in this section.

The use of these protocols is crucial to ensure both data integrity and data standardization, providing a reliable structure upon which an effective and secure data exchange

process can be built.

UBX

The UBX protocol is a proprietary protocol developed by u-blox in order to operate on its GNSS receivers. In contrast to the human-readable message content of the NMEA protocol, the information contained in the UBX protocol's message uses a binary representation format, with the advantage of obtaining much more efficiency, reliability (thanks to the binary checksum system that follow a particular algorithm) and optimized resources utilization.

UBX offers a more vast and detailed board control capacity, giving the user more enlarged and customizable configuration patterns that exploit the power of RTK applications and advanced position use cases, for example: it is possible to update the board's configuration parameters while operating by sending a special UBX message related to the configuration settings (UBX-CFG).

The UBX protocol will be discussed again during the work description and discussion and can be consulted in detail in the dedicated appendix A.

NMEA

The National Marine Electronics Association protocol was originally designed and engineered only for marine applications, but with time it became more and more diffused in every kind of positional application thanks to its lightweight, simple and effective human-readable ASCII format.

Although its capabilities are limited, the NMEA protocol provides relevant and important information to the user in a complete and accessible way. Every NMEA sentence starts with the "$" character and ends with an asterisks followed by two checksum characters that ensure the integrity of the message. The fields contained in the sentence are separated by commas, in order to be easily accessed, readable and processed.

RTCM

The Radio Technical Commission for Maritime is a data protocol specifically designed to enhance the capabilities of GNSS receivers, providing them with accurate data correction by exploiting the use of fixed base stations. In practice, raw GNSS data can be polluted by various measurement errors, for example:

- **Atmospheric delays**: caused by variations in the Earth's troposphere and weather conditions, leading to signal speed inconsistencies.
- **Satellite clock inaccuracies**: slight deviations in satellite onboard clocks that result in timing errors.

- **Ionospheric disturbances**: variations in the ionosphere's electron content affecting signal propagation speed.
- **Orbit errors**: discrepancies between the actual satellite position and the predicted orbit used in calculations.

To compensate for these errors, RTCM employs specific algorithms that process the very low position uncertainties of some fixed base stations used as references. These correction data are encoded in RTCM format, which isn't human readable but easily parsable and sent to the GNSS receiver to obtain a centimeter-level accuracy on the final data, implementing the Real Time Kinematics technology.

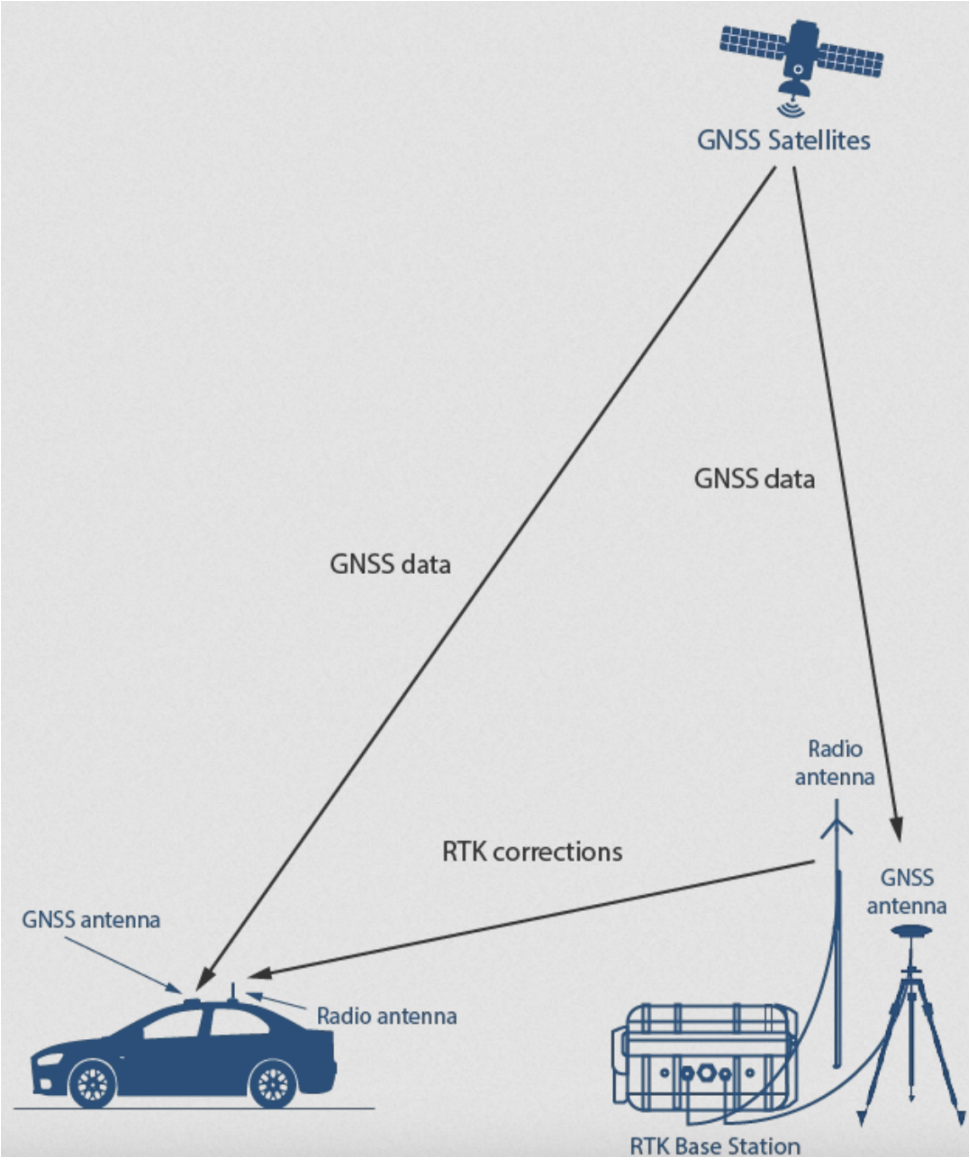

Figure 2.2: RTCM visual scheme [29]

At the time of writing, two different versions of RTCM do exist:

- **RTCM 2.X**: most common and diffused
- **RTCM 3.X**: offers advanced features like a wider satellites constellations support and multi-frequency data correction.

Since most modern GNSS applications, across diverse domains such as the automotive and agricultural sectors, can greatly benefit from highly accurate positioning data, RTCM plays a key role in enabling and advancing precise positioning solutions. It continuously contributes to improving final data accuracy and enhancing the effectiveness of RTK-based projects.

4G NTRIP

4G NTRIP (Networked Transport of RTCM via Internet Protocol) is a key technology in positioning applications, especially the ones requiring the precision offered by RTK technologies along with flexibility and efficiency needs. The power of the NTRIP protocol lies in the fact that it permits receivers to access high precision GNSS data correction using the internet, acting in fact as a network protocol and solving the problem of having a near base station that provides correction data. This capability is essential for intelligent transport systems (ITS), automated driving, and cooperative vehicular applications that are based on dynamic environments.

The 4G lettering indicates the technology used by the protocol in order to transmit data. In this case, RTK correction data are sent from NTRIP to the GNSS receiver using LTE cellular connection, which provides a fast and reliable transmission. A typical NTRIP system consists of three primary components:

- **NTRIP Server**: located together with the base station, it interacts with it and collects raw GNSS data directly from the satellite, processing and then forwarding them to the NTRIP Caster.
- **NTRIP Caster**: acting as a middle agent, it interfaces directly with the final GNSS receiver via LTE connection, effectively providing the correction data.
- **NTRIP Model**: installed on the receiver, it equips the necessary hardware to employ the LTE technology and gather the processed GNSS data.

The use of LTE represents a stable and reliable solution even in challenging conditions and scenarios like the automotive one, where the GNSS receiver is constantly moving and, most of the times, in an environment which does not favor an optimal signal propagation, like urban areas where multiple signals overlap with each other and the presence of potential signal obstruction is prevalent. This reliability is crucial for the effectiveness of all the C-ITS stack process execution. Given the high data throughput and low latency of 4G networks, NTRIP clients can maintain real-time corrections with minimal signal noise, ensuring that vehicles can make precise navigational adjustments in every condition.

With the advance of 4G technology, 4G NTRIP is expected to be replaced with 5G, offering a substantial improvement in both speed and latency performances, raising the effectiveness of the whole C-ITS stack even further.

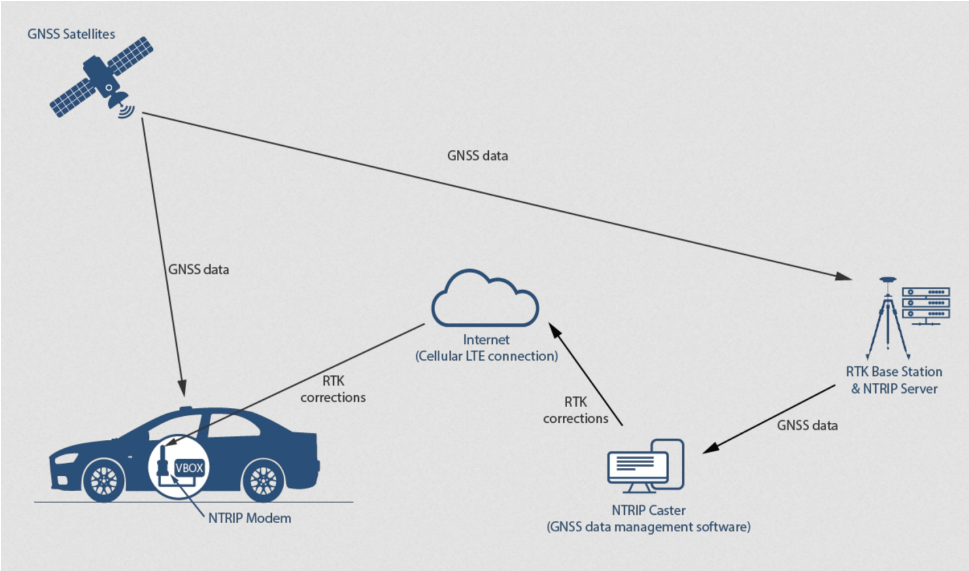

Figure 2.3: 4G NTRIP visual scheme [29]

2.1.2 Basic Interfacing

The first step of the working flow with GNSS receivers is a correct interface configuration with a personal computer in order to have simple and direct access to both positioning and configuration data. The board used in this thesis project includes a 4G NTRIP modem provided by Telit that enables the RTK data correction, using LTE cellular networks and fulfilling to this purpose.

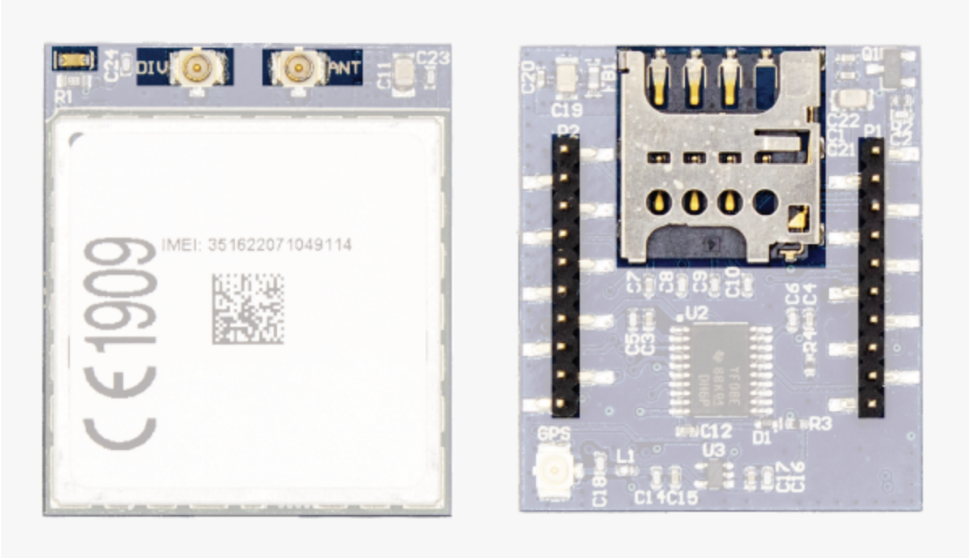

Figure 2.4: 4G NTRIP modem [1]

The board is interfaced via classical direct plug-and-play procedure provided by Windows, that will automatically attempt to install the board's driver software. In some cases, though, it may be necessary to manually install the drivers to ensure a correct interface recognition process. This software can be found in the u-blox or ArduSimple support pages. Once the initial software configuration is completed, the user can customize and interact with the board using the u-blox specific board monitoring software: u-center.

U-center Monitoring Software

The most efficient way to communicate with a u-blox GNSS module to keep track of its operational state along with all the communication and configuration data flow is the u-center monitoring software, freely provided by u-blox. This software offers a very detailed and visually viable interface useful for visualizing satellite information, setting and analyzing device performance and importing/exporting data in various format.

For what concern this thesis project, using u-center has been particularly crucial in order to set up the device and assuring correct data parsing via the binary console provided by the software, that shows the serial stream byte by byte. Moreover, having the possibility to quickly enable or disable single UBX/NMEA messages has been very convenient in order to execute the primary parsing tests.

Even though the differential GNSS RTK setup of the board has been executed externally of u-center, it is worth to mention that u-center offers a simple and direct way to manage RTCM data correction streams, allowing the user to adjust the NTRIP client settings directly via the software. It is possible to provide the NTRIP caster credentials (IP, port, username, and password) in order to accelerate the configuration project and, in addition, the software offers post-processing and GNSS RTK data logging with tools

like RTKLIB.

Last but not least, u-center provides the necessary utility tools to ensure constant and rapid firmware updates so that the user is able to manually handle the firmware update, which is critical in order to comply with the latest GNSS compatibility requirements. In some cases, though, being able to revert the device to a different firmware version than the latest one can be useful for specific testing purposes.

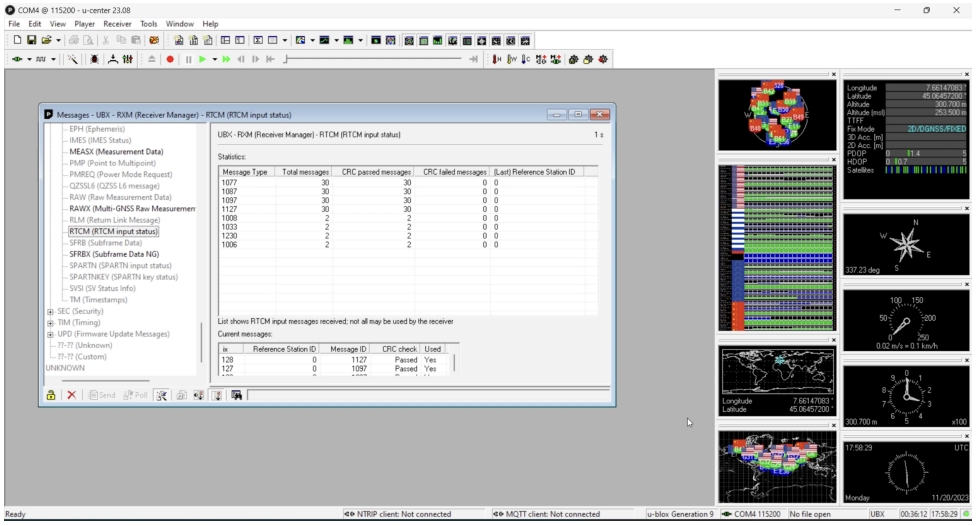

Figure 2.5: u-center main screen

2.2 Parsing Logic for UBX and NMEA Messages

The implementation of the messages parsing process finds its foundation in a serial communication interface between the software and the receiver device. This foundational binding is realized leveraging an external C++ library that provides simple APIs to manage serial connection, along with reading and writing serial data with ease.

In this section, the serial reading algorithm is presented and explained both for the UBX and NMEA messages, along with the message validation check that is different for the two protocols.

2.2.1 UBX Messages

UBX messages provide high-precision navigation data with a structured binary format using particular protocol rules specifically formulated for u-blox devices. The serial parser continuously monitors incoming data trying to detect an UBX message by looking for its specific header composed by the bytes "B5" and "62" in hexadecimal format. When a valid header is found, it then seeks the message field that gives the length of the message and continues to acquire one byte after the other until a number of scanned bytes equals

the previously read length value, ensuring a full message extraction without premature termination or data corruption. It then applies the validation algorithm in order to verify the correctness of the message and in the case this check is passed, it identifies the message type by analyzing the two bytes that follow the message's header.

The reading algorithm also accounts for the presence of overlapping UBX messages by checking if a new "B562" byte sequence is found while reading the current message. In such case, the parser starts reading the two potential message in parallel and when the reading is concluded, it verifies which one of the message is valid, parsing it accordingly.

Message Validation and Parsing

UBX message validation involves two key processes: length verification and checksum validation. The expected length of an UBX message is derived from its payload length field, ensuring the correct number of bytes are received before parsing and checksum validation is performed using the 8-bit cyclic redundancy check (CRC) Fletcher algorithm applied to the message payload. This step ensures data integrity and prevents the use of corrupted messages in downstream processing. If a checksum mismatch occurs, the message is discarded, and parsing resumes from the next valid header.

If validation criteria are met, the current message is classified using its second and third byte that indicate, respectively, the message class and the message ID, for example a message with "10" as message class and "15" as message ID indicates as "UBX-ESF-INS" message that provides data about the device dynamics. Unhandled UBX messages are ignored, ensuring efficient processing without unnecessary overhead.

2.2.2 NMEA Sentences

Since the NMEA protocol follows a simple, human-readable and ASCII based format, the reading and parsing approach is simpler and shorter than the one used for UBX messages. The algorithm starts by looking for the specific dollar character "$" that marks the start of a NMEA sentence and when it is found, it starts storing characters until a new line character "\n" is found, since the end of a NMEA sentence is marked by this latter.

Sentence Validation and Parsing

NMEA sentence validation is performed with two important supervisions: firstly on the sentence format and then in checksum matching. In order to do so, the sentence is scanned to see if the first character is a dollar sign, and if an asterisk sign, that represents the start of the checksum field, is found. If these conditions are not met, the function immediately returns `false`, indicating that the sentence is invalid.

The second format check takes place if the first one has a positive result and verifies

that the sentence's checksum field is present by finding the position of the asterisk characters and checking if there are at least two characters after it, since that the remaining part of a correct sentence, after this point, should be composed only by the two checksum values, the carriage return `\r` and the new line `\n` special characters.

When a correct sentence format is assured, the actual validation process can be performed. The operation consist in calculating the XOR value of all the characters of the sentence from the dollar sign (excluded) up to the asterisk sign (excluded) and in the final check with the checksum value obtained in the format check. Once validated, the parser extracts relevant data to be processed and used within the OScar framework. NMEA sentences that are recognized but not explicitly handled are ignored.

The UBX and NMEA message parsing mechanisms ensure reliable acquisition of GNSS data by employing structured validation and extraction techniques. By implementing robust error detection and handling, the parser effectively processes navigation data while maintaining resilience against malformed of polluted data, that can be very frequent in a serial transmission. Moreover, it is worth mentioning that the parser provides OScar the possibility to externally specify a wrong data threshold via CLI (Command Line Interface) that suspends the reading process if this limit value of wrong bytes read is exceeded.

2.3 Integration of GNSS Data Parsing in OScar

The GNSS serial parser of the OScar project consists in a specific C++ module composed by its own header and source code files, which define a Serial Parser Class that performs all the basic serial input and output operations, along with specific functions that handle the data of interest and provide them to the rest of the program easily, with a scalable and modular approach.

The integration of GNSS data parsing within the OScar framework represents a critical aspect of vehicular communication systems, ensuring precise localization and kinematic state estimation. This section elaborates on the implementation details of GNSS data parsing in OScar, particularly focusing on the Serial Parser and the Vehicle Data Provider classes. These components facilitate the acquisition, processing, and provision of essential vehicular data for Cooperative Intelligent Transport Systems (C-ITS).

2.3.1 Serial Parser Class

The Serial Parser class is responsible for interfacing directly with GNSS receivers, parsing the data received over serial communication and maintaining an atomic buffer structure to ensure efficient and real-time data access, accounting also for thread safety. It contains the main useful data and functions definition: for example the unavailable data markup values for the information parsed and the definition of an important macro that allows

the calculation of the age of the current information parsed.

```
[...]
#define Latitude_unavailable_serial_parser 900000001
#define Longitude_unavailable_serial_parser 1800000001
#define AltitudeValue_unavailable_serial_parser 800001
#define HeadingValue_unavailable_serial_parser 3601
#define SpeedValue_unavailable_serial_parser 16383

#define COMPUTE_EPOCH_TS_VALIDITY(value) (value==0?0:(((epoch_ts-value)
   >=LONG_MAX?0:(epoch_ts-value))))
[...]
```

Figure 2.6: Unavailable data constants definition and age of information calculation macro

Atomic Buffer Data Structure

Since the OScar project involves a multi-threading environment to perform its operations, it is a necessity that critical processes such as data gathering and handling respect a restrictive thread safety requirement. The solution algorithm proposes an atomic data buffer that groups all the data read from the GNSS receiver device with the scope of being accessed in a secure and thread-safe way.

Introduced in C++ 11, the `std::atomic` template provides lock-free atomic operations between threads, allowing multiple instances to safely read and modify data in a well-defined way, without the need to operate with mutual exclusion techniques such as `std::mutex`, and keeping low synchronization overheads.

Atomic operations are very important in real-time systems like OScar, where data coherence is a key factor that must be accounted for and that can drastically influence the software behavior, so it is truly worth paying the cost of implementing a computational challenging solution, gaining though, the certainty of a thread safe environment using a tool that additionally optimized this process.

In the serial parser module of the project, the `std::atomic` template is used only to define the buffer data structure privately in the context of the class, storing it in the `m_outBuffer` variable, which is accessible through the appropriate class methods for gathering data.

The atomic data buffer provides critical data for the correct execution of OScar such as:

- **Fix mode information from both UBX messages and NMEA sentences**: provide the current GNSS level of quality and accuracy of the data, indicating, for example, if the receiver is operating in Real Time Kinematics mode or, in the case of temporary satellite signal absence, in "Dead Reckoning" mode.

- **Positioning data** such as latitude, longitude and altitude, obtained from both the UBX and NMEA protocols.
- **Velocity and heading information**: also referred respectively as "speed over ground" and "course over ground", they are widely used in all the services implemented by OScar, for example they are mandatory in order to determine when to send a CAM message in the Cooperative Awareness Basic Service.
- **Attitude data**: yaw, pitch, roll. Obtained from the ZED-F9R module, they are very important for data coherence assurance and, in the case of the yaw value, for having an alternative measure that additionally verifies the correct behavior of the system.
- **Accelerations and angular rates**: contribute to the scope of having a complete understanding of the vehicle functioning, in order to provide a more efficient and unified testing process, paving also the way for future services implementation that exploit this data.

Moreover, the output buffer makes use of atomic variables to ensure data consistency even in high-frequency update scenarios, defining the last update value timestamped in microsecond thanks to the `std::chrono` template for every information, with the scope of computing the "age" of the information itself, continuously verifying if such data value respects the low-latency requirements dictated by the standard, assuring the quality of the services implementation.

Parsing Algorithms Overview

The serial GNSS data parsing module has been developed in order to have specific compatibility with the ublox ZED-F9R receiver but at the same time to be also modular and easily adaptable to other devices. With this goal in mind, the software handles every message, both from UBX and NMEA with a specific parsing function that works only on a particular data value.

Since the UBX protocol is more specific and is used for a wide variety of data it requires a slightly longer parsing procedure that involves vector manipulation along with format and endianness conversions.

With exception of the function `parseNavStatus()` which provides the fix value and validity of the receiver, every UBX parsing function involves the use of a support vector that will contain the relevant part of the message being parsed. Following the u-blox ZED-F9R interface description [27], the bytes of interest have been sought manually and with the help of the UBX payload offset, all the functions have been written following the same flow of the reference document, with the scope of producing clear code that results to be as self-explanatory as possible.

For what concerns the NMEA protocol, the fact that it is designed to be simple and

comprehensive makes the parsing process simpler, so the code procedures are shorter than the ones adopted for the UBX protocol. In summary: the algorithm scans the sentence dividing it by fields with the use of comma characters that delimit every sentence field and then distributes every information of interest in specific variables, in order to be processed.

Below follows a brief overview of the key parsing functions, clarifying that a complete algorithm description along with code snippets are present in the thesis appendices:

- `parseNmeaRmc()`: This sentences is one of the most valuables in the NMEA protocol, RMC means in fact that the sentence contains the recommended minimum GNSS data, that include time, date, position, heading and speed data. The parser extracts data including: speed-over-ground, course-over-ground (heading), and fix status. Speed values originally provided in knots are converted to meters per second, and course-over-ground values are appropriately parsed, including considerations for potential negative values. The function also rigorously checks and assigns the fix validity status, which is essential for downstream processing tasks.
- `parseNmeaGns()`: this function addresses the "GXGNS" NMEA sentences, commonly used to retrieve position-related information. It first divides the incoming NMEA string based on comma delimiters, extracting the fields for latitude, longitude, and altitude. Latitude and longitude values initially provided in the traditional NMEA "degrees-minutes" format are converted to decimal degrees thanks to the function `decimal_deg()` utility method. Additionally, this function integrates logic to interpret the global fix mode string, handling multiple GNSS constellations (GPS, Galileo, GLONASS, BeiDou) simultaneously, ensuring the most reliable positioning fix mode is determined and logged.
- `parseNmeaGga()`: this procedure manages the parsing of "GXGGA" sentences, known for providing essential positional data including latitude, longitude, altitude, and GNSS fix information. The parsing strategy mirrors the one of parseNmeaGns(), focusing on extracting and converting geographic coordinates and determining the fix status based on standard indicators within the NMEA sentence fields. Parsed data undergoes validation steps and is then timestamped, ensuring synchronization of data availability information.
- `parseNavStatus()`: this function specifically handles the receiver navigational status, gathering information about the fix mode and the validity of this latter. These two values are obtained selecting the correct bytes of the UBX message based on the payload offset described in the interface reference document and than, after a fix mode validation check, a series of checks are done in order to determine the current fix mode. Finally, according to the found fix mode, the internal flags that indicate the current fix inside the class are set.

- `parseNavPvt()`: this function deals specifically with the UBX-NAV-PVT message type. According to the ublox ZED-F9R interface documentation, the NAV-PVT message encapsulates comprehensive positioning, velocity, and timing data. The function extracts speed-over-ground (SOG), course-over-ground (COG), latitude, longitude, altitude, and fix quality. Each of these parameters is parsed through selective byte extraction based on predefined offsets within the message payload. Since UBX messages employ a big-endian byte order, the function first reverses the byte order to match the little-endian architecture and then the numerical values are reconstructed using the specialized utility function `hexToSigned()`, before being scaled appropriately according to their documented units.
- `parseNavAtt()`: this function handles the UBX-NAV-ATT message. This UBX message provides detailed attitude information, specifically the vehicle's roll, pitch, and heading. As defined in the ublox protocol, the payload contains these values in fixed-size byte sequences, with specific offsets and data lengths. The parser extracts these sequences, again converting byte-order from network (big-endian) to little-endian and interprets the resultant integer values. The resulting attitude angles are scaled by predetermined factors (e.g., the heading value is scaled by 10^{-5} degrees). After parsing, these values are again stored into the atomic buffer to facilitate synchronized use by other modules or functions within the OScar framework.
- `parseEsfRaw()`: processes the UBX-ESF-RAW message, which provides raw sensor data from the ZED-F9R sensor fusion module. The payload of the UBX-ESF-RAW message consists of multiple repeated groups of bytes, each encoding raw sensor data about the acceleration values along the x, y, and z axes. This function parses each group selectively, identifying the axis type from a specific data-type byte and subsequently reading and converting the corresponding 3-byte data sequences. After parsing, the raw acceleration data is converted from its binary representation into floating-point numbers using the previously mentioned utility function `hexToSigned()`, then appropriately scaled dividing its value by 1024 as stated in the documentation. Finally, these values are timestamped and updated in the atomic buffer for further processing.
- `parseEsfIns()`: formulated for the UBX-ESF-INS message, this function carries the extraction of compensated accelerations and angular rates without gravity components. The function isolates specific sections of the payload, separately selecting angular velocities (yaw rates) and compensated accelerations. Each byte group corresponding to an axis-specific measurement is reversed for byte-order adjustment, reconstructed into signed integers, and scaled according to protocol specifications (accelerations in m/s^2 and angular rates in deg/s). The parsed data and associated timestamp are again atomically stored, allowing precise synchronization and

retrieval in real-time.

These parsing functions collectively enable the robust derivation of GNSS data from both UBX and NMEA formats, converting raw byte streams into structured, usable information in order to be used in the Cooperative Awareness Messages (CAMs) generation within the broader scope of the OScar project. The careful design considerations, such as atomic thread-safe data storage and precise handling of message validity, align with the project's goal of providing reliable and timely positioning information for cooperative vehicular communication systems.

2.3.2 Vehicle Data Provider

The Vehicle Data Provider module (VDP) is a fundamental code section whose main task is binding the C-ITS services implementations with the data gathering procedures implemented in the serial parser or in the interfacing with the `gpsd` daemon code.

In order to fulfill its duties, the VDP object hosts an instance of the serial parser object and works with it using the specific interaction functions that gather data from the atomic buffer in order to create vehicular messages and to verify the conditions that confirm their validity. A key example of this workflow is the encapsulation of the Cooperative Awareness Messages encapsulation and verification, which will be extensively discussed in the next chapter.

The Vehicle Data Provider module's design ensures that information from the GNSS receiver (through gpsd or other integrated services such as the serial parser module) is accurately captured, processed, and then packaged into relevant vehicular messages including, for the sake of this thesis project, Cooperative Awareness Messages (CAMs) and Vulnerable Road Users Awareness Messages. In doing so, the VDP not only collects the necessary parameters such as position, velocity, and time but also verifies their correctness before passing them to the C-ITS implementation layers.

This strategy guarantees that any changes in the real world vehicle conditions are reflected promptly, enhancing the adaptability and reliability of the overall communication system. By systematically verifying the validity of the data against the service's requirements, the VDP stands at the core of a robust C-ITS environment, helping maintaining high quality and standards-compliant messages transmissions that support safer and more efficient vehicular interactions.

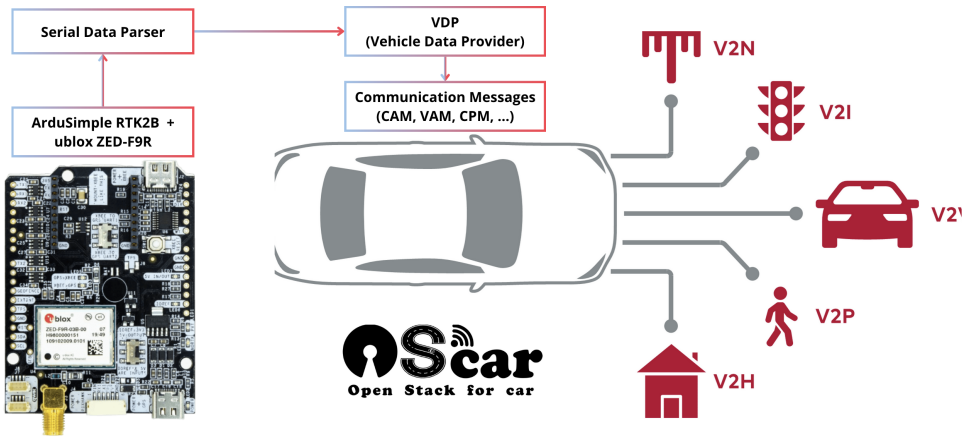

Figure 2.7: OScar project information flow scheme

2.3.3 Utilities and Debugging Options for OScar

Over the course of this thesis project, the serial parsing module was introduced alongside three Command Line Interface (CLI) options designed to enhance system security and provide a more immediate and transparent debugging process during testing. The first option, specified by the command `-set-wrong-input-threshold`, enables the user to set a byte-reading threshold beyond which the program will terminate if exceeded. The default threshold is 1000 bytes. The second option, invoked with the `-y` parameter, allows the user to define the maximum number of seconds after which a piece of information is marked invalid. Lastly, the `-show-live-data` option makes it possible to display, at a given refresh rate, all the main data read by the serial parser alongside each piece of data corresponding âage of informationâ. This metric quantifies the time elapsed between the moment the data is initially read and the moment it is requested by the VDP or any other part of the software.

Employing these options proves to be critical especially during local and field testing. In particular, the debug option `-show-live-data` offers a comprehensive real-time overview of the vehicle's behavior, enabling immediate detection of inconsistencies or unexpected events. This approach significantly reduces the time and complexity associated with post-test log analysis by allowing potential issues to be identified and addressed as they arise.

Chapter 3

Cooperative Awareness Basic Service and Data Encapsulation

3.1 Cooperative Awareness Messages Overview

Cooperative Awareness (CA) represents one of the crucial milestone of contemporary vehicular communication systems such as C-ITS, ensuring that vehicles, road infrastructure, and other road users can exchange important status and position information in real-time.

At the heart of the Cooperative Awareness paradigm lies the concept of Cooperative Awareness Messages (CAM): a standardized message format defined by the European Telecommunications Standards Institute (ETSI) in its EN 302 637-2 specification [11]. CAMs enable vehicles to broadcast their position, motion parameters, and basic status information in a very fast periodic way, thereby improving situational awareness and supporting a range of Intelligent Transport Systems (ITS) safety and efficiency applications.

Even with the remarkable standards and technologies improvement, CAMs have remained a fundamental block of the vehicular network communication stack, still continuing nowadays to guarantee reliability and information integrity, representing a concept of paramount importance in the vehicular communications environment.

3.1.1 Introduction to Cooperative Awareness Messages

A Cooperative Awareness Message is a periodic broadcast message that contains essential information about a vehicle's position, speed, heading, and other status data. Being this content standardized, any receiving entity: be it another vehicle, a roadside unit, or a traffic control center can interpret the transmitted information unambiguously and with very low latency. The importance of CAMs derives from their role as fundamental data carriers in cooperative vehicular communication.

By disseminating real-time status updates, CAMs enable the creation of a coherent

picture of the current traffic situation, with useful information about any kind of user desire. Some examples are represented by: traffic congestion level, roadwork and deviations warnings, hazard and road accidents signaling, collision avoidance, lane-change assistance, adaptive cruise control, and vulnerable road user (VRU) protection [25]. Moreover, CAMs serve as an essential input to higher-level services like Collective Perception Service (CPS) [21] and Vulnerable road users Basic Service (VBS) [15], which build upon the same ETSI communication frameworks.

3.1.2 CAM Structure

By establishing a common format and set of rules for how messages should be generated, packaged, and interpreted, ETSI provides a framework that helps ensuring that safety-critical information are reliably transmitted and understood across diverse networks and different devices. For this important purpose, the definition of a clear and coherent message structure format for all the communication messages in the C-ITS realm comes into play.

Although updates to ETSI specifications can introduce refinements or additional parameters, the foundational structure of a CAM remains stable and follows a container-based approach. Each container is designed to encapsulate data relatively to a specific set of functionalities or information types, which collectively describe an ITS station's status and environment. A CAM must include at least two primary containers: the Basic Container and the High-Frequency Container. The Basic Container stores fundamental information such as the ITS station's type (e.g., vehicle, roadside unit), its unique station identifier, and other core attributes needed to ensure that the message is valid and, most importantly, traceable. On the other hand, the High-Frequency Container captures dynamic information that changes regularly, such as the vehicle's current speed, heading, position, and acceleration.

Another very important aspect of this container-based design is the possibility of incorporating different information from various types of ITS stations defining alternative containers as needed without compromising the interoperability guaranteed by the CAM base structure. For example, some ITS stations might include positioning quality indicators or environmental data that benefit traffic management or safety applications, others could integrate data for cooperative driving maneuvers support or enhanced situational awareness, enabling more sophisticated forms of vehicles coordination.

In all cases, these optional extensions must adhere to the well-defined container architecture, ensuring that a station receiving the message can interpret the extra data properly if it supports the relevant standards, or safely ignore the additional information if it does not. This flexible yet uniform design underpins the message's role as one of the main message types in cooperative ITS, making it possible for emerging automotive

technologies, infrastructure deployments, and even non-vehicle transport systems (such as cyclists equipped with ITS devices) to exchange critical data in real time without region-specific or manufacturer-specific barriers.

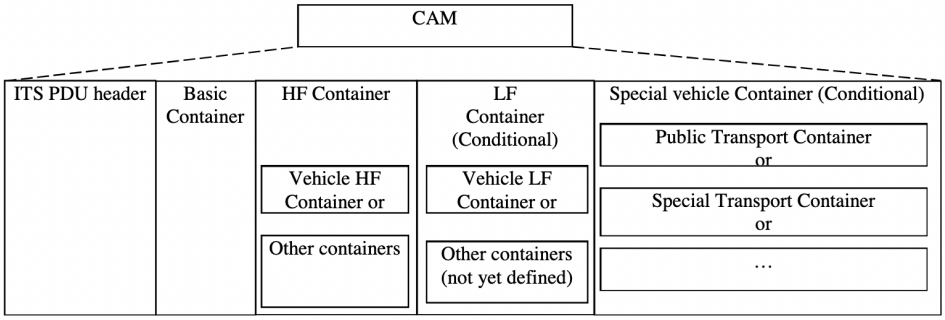

Figure 3.1: Cooperative Awareness Message structure, following the official ETSI standard [4]

Key Message Components

While the Cooperative Awareness Message (CAM) is crucial in its entirety for vehicular communication, certain components within its structure are particularly stand out:

- **Positioning Fields**: These message components captures the real-time coordinates of the transmitting station. Highly accurate positioning is critical for safety-critical maneuvers and cooperative features; thus, modern CAM implementations often rely on GNSS systems capable of multi-constellation and multi-frequency operation using, when available, Real Time Kinematics solutions to ehnance the precision and the final quality of the results even more.
- **Motion Parameters**: speed, acceleration, and heading collectively describe the kinematic state of the vehicle or ITS station, forming the bases for predictive algorithms that estimate future positions and trajectories.
- **Station Type and Attributes**: This metadata clarifies the nature of the transmitting entity, distinguishing between passenger cars, trucks, motorcycles, bicycles, pedestrians or infrastructure components like Road Side Units (RSUs). In certain safety applications, these distinctions enable tailored response protocols (e.g., giving higher priority to emergency vehicles), making use of other Cooperative Basic Services such as the Vulnerable Road Users Basic Service (VBS).
- **Security Header** (when implemented): Though not always mandatory, security mechanisms defined in ETSI TS 103 097 [18] can be integrated into CAMs, providing integrity and authenticity checks through digital signatures. The presence of a security header is crucial in large-scale deployments to prevent malicious impersonation or message tampering. As seen previously, the OScar project gives particular value to the security aspect, providing to all the vehicular messages of the C-ITS

stack the appropriate security methodologies.

Together, these key components enable applications to interpret each CAM in a meaningful context, supporting functionalities that augment the quality offered by the final vehicular network applications.

3.1.3 CAM Adaptive Generation Frequency

A key aspect that sets Cooperative Awareness Messages apart is their high transmission frequency, which generally ranges between 1 Hz and 10 Hz under normal driving conditions. This value determines how often a vehicle's status is shared and, consequently, how consistently surrounding participants can track the vehicle's dynamic state.

The choice of frequency is not arbitrary: on one hand, a higher generation rate increases the fidelity of information exchanged and on the other hand, generating and transmitting these messages too frequently can overload the communication channel, risking congestion and collisions of broadcast messages. This constitute the main reason why the CAM generation frequency varies in time adaptively.

This message congestion phenomenon is especially pronounced in traffic-dense areas or large-scale event scenarios, where many vehicles might simultaneously transmit status updates. In order to deal with these problems vehicles are instructed to increase their CAM generation frequency when executing demanding maneuvers like sharp turns or hard braking and to reduce it when traveling at a steady velocity, with minimal changes. By adjusting messaging rates based on a vehicle's real-time behavior, the system can optimize network utilization while still delivering timely, high-quality awareness information for safe and efficient driving.

The dynamic frequency adaptation mechanism ensures that when vehicle motions are smooth and stable, the transmission rate can be scaled down to reduce unnecessary network overhead, and under more complex or safety-critical conditions, vehicles can ramp up their transmission frequency, allowing other road users to react more promptly and accurately. Thus, the key challenge lies in managing the trade off between minimizing communication overhead and ensuring timely updates for precise situational awareness. [4, 30].

Impact on Real-Time Communication

Real-time communication in vehicular networks relies on short latency, high reliability and consistent throughput. The frequency of CAM transmission directly influences all three of these factors:

- **Latency**: With frequent broadcasts, the time between the moment a vehicle's state changes and when neighbors learn of that change becomes shorter. This is particularly important for collision warning systems and other safety-critical alerts.

- **Reliability**: A higher generation rate can improve robustness, as multiple CAMs that carry similar information can increase the probability of successful reception under varying radio conditions.
- **Scalability and Network Load**: in high-density traffic scenarios, the simultaneous transmission of messages from a large number of vehicles can create significant scalability challenges. Communication systems, regardless of the underlying technology, need to implement effective strategies to manage network congestion and ensure a reliable and real-time data exchange. Adaptive mechanisms are typically used to regulate message generation and transmission in order to maintain stable channel conditions.

By addressing these aspects of real-time communication, CAMs fulfill their goal of improving traffic safety and efficiency, setting the stage for advanced use cases such as cooperative maneuvering, platooning, and fully autonomous driving.

3.2 High-Frequency Container Population Using GNSS Data

One critical component of the CAM, particularly relevant for real-time vehicular scenarios, is the High-Frequency (HF) Container. As illustrated above, this container specifically addresses highly dynamic parameters that change rapidly over time, such as the vehicle's instantaneous position, speed, heading, and acceleration. The precision and reliability of the data populated within this container are fundamental for the effective operation of V2X safety applications.

Using Real Time Kinematics devices it is possible to improve the quality of the service even more, filling the CAM message with positioning data with centimeter-level precision, along with a broad selection of additional optional data provided by serial boards such as the ArduSimple Simple RTK2B paired with the u-blox ZED-F9R module.

3.2.1 Populating CAM Containers

The process of populating Cooperative Awareness Message (CAM) containers requires careful integration of various data inputs from onboard sensors, with GNSS devices being primary data sources. The serial parser software module aims at the optimization of the efficiency of this process by providing data to the Vehicle Data Provider module employed in the Cooperative Awareness Basic Service implementation process, which will ultimately perform the final check operations before filling the CAM message with the data received.

This message population phase is supported by the Abstract Syntax Notation One

(ASN.1) standard, which provides a precise and unambiguous way to define data structures for encoding and decoding messages. ASN.1 ensures that each element of a CAM message, from its core fields to optional extensions, is rigorously specified in a manner that is both human-readable and machine-interpretable. This level of detail helps developers implement the CAM population phase across diverse systems, enhancing compatibility and reducing the likelihood of errors during data exchange. By exploiting ASN.1 it is possible to describe the full life cycle of CAM messages, including how they are populated, transmitted, and validated, because of the standard's encoding rules (such as BER, DER, PER, or OER) that govern every aspect of message formatting. Consequently, the ASN.1 supported definition of the CAM population phase determines a reliable framework for intelligent transportation and connected vehicle applications, ensuring robust communication protocols and seamless system integration.

In conclusion, after being correctly parsed and validated, all the necessary information for an optimal CAM filling undergo a mapping phase that uses the ASN.1 C++ library to perform the operations illustrated earlier. The code example below shows ASN.1 data mapping for the altitude value and altitude confidence in the CAM basic container filling.

```
/* Fill the basicContainer */
asn1cpp::setField(msgstruct->cam.camParameters.basicContainer.
    referencePosition.altitude.altitudeValue,cam_mandatory_data.altitude.
    getValue());

asn1cpp::setField(msgstruct->cam.camParameters.basicContainer.
    referencePosition.altitude.altitudeConfidence, cam_mandatory_data.
    altitude.getConfidence());
```

Figure 3.2: CAM Basic Container filling using ASN.1

3.3 CA Basic Service Implementation

3.3.1 GNSS Serial Data Parsing and Vehicle Data Provider Integration

At the core of CAM generation within the OScar framework lie the Vehicle Data Provider (VDP) module, specifically implemented by the `VDPGPSClient` object and defined in the `gpsc.cpp` file and the serial parser module, represented by the `UBXNMEAParserSingleThread` object.

The GNSS data utilized in this context originates primarily from serial interfaces handling NMEA and UBX protocols. The parser object is capable of handling real-time GNSS data from devices such as the u-blox ZED-F9R, providing robust position, velocity, heading, and acceleration values. This parsing stage ensures timely and accurate retrieval of kinematic parameters crucial for cooperative vehicular awareness.

The parser functions, such as: `getPositionUbx()`, `getSpeedUbx()`,`getYawRate()` and many more, systematically extract and validate GNSS data, applying rigorous checks on the validity and integrity of the received signals and ensuring that only precise and up-to-date GNSS data are gathered and sent to the subsequent CAM message population phase.

```
std::pair<double,double>
UBXNMEAParserSingleThread::getPositionUbx(long *age_us, bool
    print_timestamp_and_age) {
    if(m_parser_started==false) {
        std::cerr << "Error: The parser has not been started. Call
    startUBXNMEAParser() first." << std::endl;
        return std::pair<double,double>(0,0);
    }
    out_t tmp = m_outBuffer.load();
    auto now = time_point_cast<microseconds>(system_clock::now());
    auto epoch_ts = now.time_since_epoch().count();
    long local_age_us = epoch_ts - tmp.lu_pos_ubx;
    long validity_thr = getValidityThreshold();
    if (m_debug_age_info_rate) m_debug_age_info.age_pos_ubx =
    local_age_us;
    if (age_us != nullptr) *age_us = local_age_us;
    if (local_age_us >= validity_thr) {
        m_pos_valid_ubx.store(false);
    }
    else m_pos_valid_ubx.store(true);
    return std::pair<double,double>(tmp.lat_ubx,tmp.lon_ubx);
}
```

Figure 3.3: `getPositionUbx()` method

3.3.2 CAM Generation and Sending Condition Checking

A key passage in the CAM message generation logic within the OScar project is the function `checkCamConditions()`. This method periodically checks the conditions outlined by the ETSI EN 302 637-2 [11] standard to determine if CAM generation and dissemination should be performed. These critical checks involve verifying if significant changes have occurred in the vehicle's heading, position, or speed. Specifically, according to the standard, CAM messages are generated if:

- **Heading value condition**: The absolute heading deviation exceeds 4 degrees since the last CAM sent.
- **Position value condition**: The positional displacement is greater than 4 meters.
- **Speed Value Condition**: The vehicleâs speed changes by more than 0.5 m/s.
- **Time value condition**: The time elapsed since the last CAM sending is equal to or greater than `T_GenCam`

The variable `T_GenCam` represents the CAM generation interval and is dynamically calculated within the range specified by the ETSI CAM standard and lies between 100 milliseconds and 1000 milliseconds. Its value depends primarily on the vehicle's dynamics, for example under conditions of rapid change in speed, heading, or position, a shorter time interval (approximately 100 ms) is used. Instead, when the vehicle's state is relatively stable, the value can be safely extended towards the maximum permitted limit of 1000 milliseconds.

Another crucial variable defined by the ETSI standard is `N_GenCam`, which counts the number of consecutive CAM generations triggered by the dynamic conditions explained before, up to reaching a predefined threshold (`N_GenCamMax`). According to the standard, if no significant vehicle dynamics changes occur, CAM messages must still be generated periodically when the elapsed time reaches `T_GenCam`. Conversely, `N_GenCamMax` ensures that a certain number of dynamically triggered CAMs can be generated rapidly when needed, but it also prevents indefinite high-frequency message generation by enforcing an upper limit.

These conditions are meticulously assessed by leveraging the data acquired from the Vehicle Data Provider object that ensures consistent and accurate monitoring of the vehicle's dynamic status. When a condition is triggered, the workflow switches to the `generateAndEncodeCam()` method that, relying on the Basic Transport Protocol (BTP) and GeoNetworking (GN) services, prepares the lower stack layers for the sending operation and, after setting up the mandatory ASN.1 format variables, calls the `fillInCam()` procedure that will gather data, mapping them accordingly to all the CAM required message fields.

Chapter 4

Application Testing and Results

4.1 Field Testing

Field testing constitutes an essential component in the validation of GNSS-based cooperative awareness applications due to its complete reflection of real-world scenarios. Unlike simulations, real-world testing involves unpredictable and dynamic conditions that substantially influence the accuracy, availability, and reliability of GNSS data. To achieve accurate, reliable, and replicable outcomes, selecting an appropriate environment for testing is crucial. Field tests conducted in carefully selected environments ensure that the system is assessed under various conditions closely, resembling actual operational scenarios. A controlled yet realistic testing environment provides insights into how vehicular communication systems perform under practical conditions, which laboratory or simulated environments might not fully replicate.

4.1.1 Environmental Considerations

A significant aspect of environmental considerations in GNSS-based testing is the influence of weather conditions on signal reliability and accuracy. Weather phenomena such as heavy rain, snow, dense fog, and atmospheric moisture can significantly affect GNSS signals, causing signal attenuation, multipath effects, and increased signal delays due to atmospheric disturbances. An great example can be heavy rainfall and dense cloud cover that can attenuate GNSS signals, leading to reduced accuracy or even temporary signal loss. Similarly, ionospheric disturbances, common during solar storms or heavy cloud cover, can cause errors in GNSS signal propagation, impacting positional accuracy.

Additionally, urban and rural environments present distinctive challenges for GNSS signal quality. Urban canyons-like areas characterized by tall buildings, bridges, or tunnels pose an relevant interference, obstructing direct GNSS signals and causing reflected signals to reach receivers, thereby resulting in degraded positional accuracy.

4.1.2 Hardware Components Overview

The accuracy and reliability of Cooperative Awareness Message (CAM) dissemination depend considerably on the robustness of the hardware components employed during field testing. Within the OScar project, specific hardware selections have been made to ensure high-quality GNSS data acquisition, reliable data processing, and effective message transmission. As described before, the Ardusimple Simple RTK2B board, integrated with the u-blox ZED-F9R module, is a crucial hardware component within the OScar testbed and, in addition to all the features described earlier, it has a compact, light and thin shape, so that it can occupy as low space as possible both in the testbed and in the final product design.

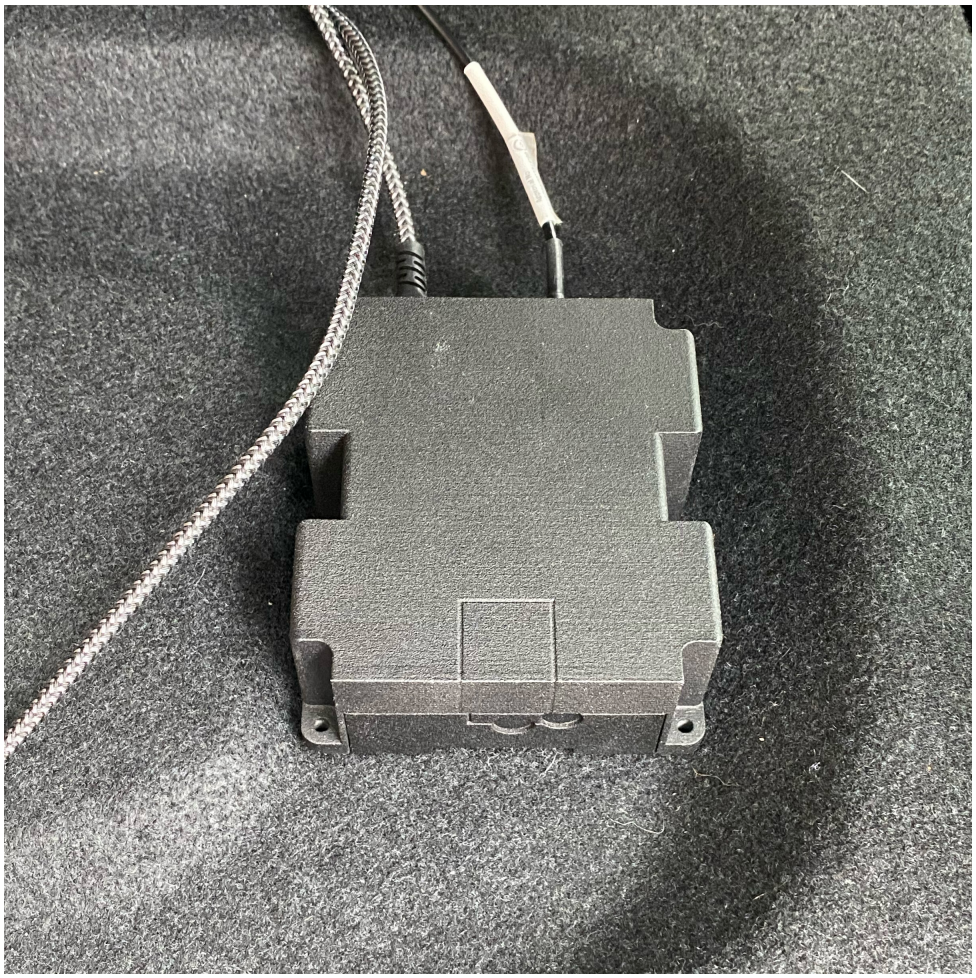

Figure 4.1: ArduSimple Simple RTK2B Board inside its protective field test case

Complementing the GNSS receiver, the GNSS antenna is strategically placed on top of the test vehicle to optimize sky visibility and minimize signal obstruction. The antenna's optimal placement significantly mitigates potential multipath errors reflections from surfaces near the receiver that could introduce inaccuracies into the GNSS measurements and

an unobstructed line of sight between the GNSS antenna and satellites ensures the highest achievable data quality, which is crucial for accurate vehicle positioning and subsequent CAM generation.

Figure 4.2: GNSS High Fidelity Antenna

Finally, an Advanced Processing Unit (APU), running the OpenWrt Linux distribution, hosts the OScar framework. OpenWrt is chosen due to its lightweight, flexible, and efficient operating specifically tailored for embedded and networking applications. The APU processes GNSS data acquired from the RTK GNSS receiver, applies data validation logic, populates CAM containers, and manages message encapsulation and transmission processes. This integration allows real-time processing and reliable message dissemination over vehicular communication networks, fulfilling the stringent timing and reliability requirements of ITS applications.

Figure 4.3: Advanced Processing Unit running OpenWrt and the OScar executable

By combining RTK positioning, inertial-based dead reckoning, and efficient data processing, these components ensure that position information remain precise even in challenging signal environments. This synergy confirms the reliability of the overall testing environment and the reduction of inaccuracies and latency in the flow of cooperative awareness information.

4.1.3 Real-Time Data Collection

Real-time data monitoring emerges as a crucial practice in vehicular communication system field testing, significantly impacting the efficiency and reliability of test sessions. Real-time monitoring provides immediate feedback on system behavior, allowing developers to quickly identify and address unexpected anomalies or deviations from the expected outcomes. With particular focus to the OScar project, this approach contrasts sharply with offline analysis methods, which inherently delays the identification of issues until test completion, potentially compromising subsequent data validity and requiring costly re-testing in terms of both time and resources.

The OScar framework explicitly acknowledges the necessity of real-time monitoring through the introduction of the `-show-live-data` feature. This option, developed as part of the OScar project's serial parser module, provides users with live access to fundamental vehicle status information such as geographical coordinates, speed, heading, acceleration, yaw rate, and altitude. Through the possibility of specifying the data refresh rate directly within the command line interface it is possible to adapt the output to the specific needs for any type of testing, for example, by issuing the command option `-show-live-data 900` the data will have a refresh rate of 900 milliseconds.

This live data visualization capability significantly enhances field tests value by facilitating immediate comprehension of the vehicle's status and operational context, as illustrated by the output shown in the figure below.

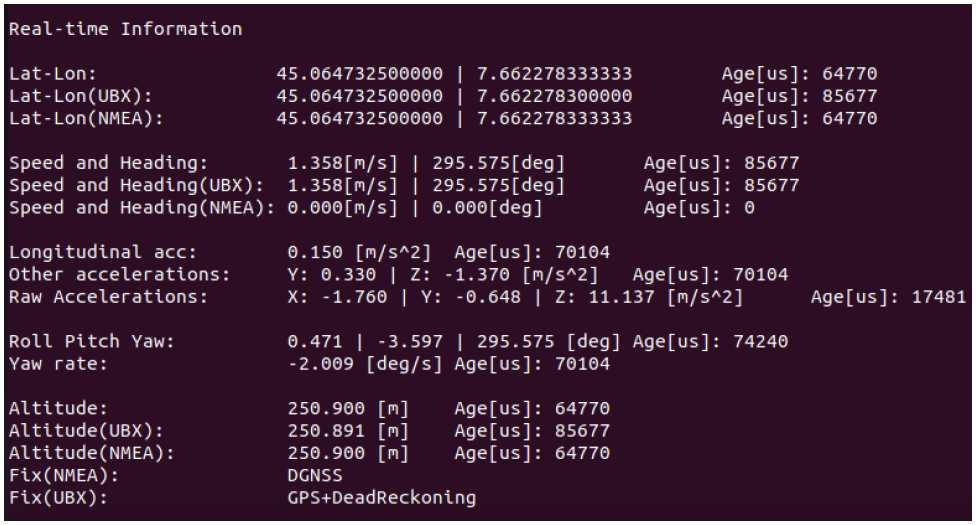

Figure 4.4: Output of the OScar executable with the `-show-live-data` option enabled

Continuous visualization of positional and dynamic parameters, derived from GNSS data and inertial sensors, empowers users to proactively monitor system health and validate data correctness in real-world scenarios. For example, discrepancies between different data sources (UBX versus NMEA positioning data) can immediately highlight problems related to data acquisition or parsing mechanisms. Similarly, sudden anomalies in acceleration or yaw rate values might indicate issues unrelated to GNSS hardware itself, such as incorrect installation of inertial measurement units (IMUs) or their uncompleted calibration process, electrical interference, or mechanical vibrations from the test vehicle.

By leveraging the `-show-live-data` feature, OScar allows testers to ensure that GNSS and inertial data are being continuously and correctly processed and transmitted to subsequent cooperative awareness layers, and ultimately, this real-time approach helps ensure that data quality is maintained as high as possible throughout the test campaign, pre-optimizing in a robust and reliable way the subsequent output log data analysis phase.

4.2 Local Simulation Testing

Local simulation testing represents a critical complementary approach to field experimentation, allowing extensive verification and validation of software and hardware behavior under controlled conditions. While field testing is indispensable for capturing the real-world operational peculiarities and variations, local simulation testing completes these efforts by providing a rapid and replicable environment for verifying software functionalities and performances without the logistical and the time complexities of physical tests deployment.

Local simulation environments allow developers to iterate swiftly, ensuring that new functionalities and bug fixes are reliably validated before integrating them into the main codebase and subsequent real-world experiments, optimizing thus the outcomes of the confirmation field test. Moreover, simulation testing significantly reduces development time and associated costs by identifying issues early and effectively narrowing the scope of necessary field verifications. To fulfill this main task, in parallel with the serial parser module, the OScar development team has been working on a lightweight, simple to use and efficient software utility called TRACEN-X which stands for Telemetry Replay and Analysis of CAN bus and External Navigation data [3].

TRACEN-X

The TRACEN-X software environment significantly streamlines development cycles by providing immediate feedback on code modifications through rapid and controlled testing of GNSS data parsing, CAM generation, and communication logic in a simulated but highly realistic environment. The core advantage offered by TRACEN-X is its functionality to replicate GNSS hardware behavior virtually. TRACEN-X achieves this by simulating GNSS receivers and serial devices, accurately replaying previously recorded real-world GNSS data traces.

The workflow associated with TRACEN-X starts with the recording of GNSS data during actual field tests, where the software captures all critical GNSS output via a serial interface and encodes this information into structured JSON trace files. Subsequently, these recorded JSON trace files can be loaded into the TRACEN-X environment, reproducing the original GNSS data stream under controlled local conditions, effectively simulating a serial GNSS receiver.

```
python3 record/record.py --enable_serial --device=/dev/ttyACM0 --
    serial_filename=./data/outlog.json --baudrate=115200 --end_time=50 --
    enable_CAN --CAN_device=vcan0 --CAN_filename=./data/CANlog.json --
    CAN_db=./data/motohawk.dbc
```

Figure 4.5: Example command for recording a field test trace

```
replay/replay.py --serial-filename ./data/passeggino_short.json --enable
    -serial
```

Figure 4.6: Example command for replaying a field test trace

In addition to the main usage methods, when using TRACEN-X it is possible to take advantage of the `-enable-test-rate` option that provides a structured overview of the JSON data trace, converting it into a CSV (Comma Separated Value) file for efficient analysis. This feature enables testers to quickly assess key GNSS parameters such as speed, heading, and altitude variations while ensuring data integrity by identifying gaps or inconsistencies. The structured format allows for performance metric computation, facilitating comparisons across different test scenarios and software versions.

The simulated environment offered by TRACEN-X gives unparalleled advantages during the debugging and validation phases. Firstly, it significantly reduces the testing overhead, eliminating the need for frequent deployments into physical testbeds. Secondly, it allows precise scenario reproduction, enabling efficient identification of software bugs, logical issues, or anomalies in data handling. Lastly, it provides a powerful way of testing system robustness against edge conditions or abnormal behaviors that may rarely occur in actual field testing, such as sudden GNSS signal loss, corrupted or partial GNSS data packets, or unpredictable vehicle movements. Moreover, the ability to simulate a broad range of GNSS conditions, from ideal open-sky scenarios to obstructed urban environments, allows a complete exploration and verification of the CAM generation logic, significantly speeding up development cycles. By bridging field testing complexities with rapid and controlled local debugging capabilities, TRACEN-X notably enhances the robustness and resilience of cooperative awareness software developed within the OScar ecosystem.

4.2.1 Peri-Urban Test Scenario in Mixed Driving Conditions

The test scenario that illustrates the results described in this thesis was carefully chosen so that to include a broad set of driving situations to be analyzed, including sections without direct satellite signal, in order to also evaluate the Dead Reckoning performances.

Recorded in Biella, Piedmont, it introduces a peri-urban environment, bridging the gap between urban density and rural expanses. This scenario features a mixed road environment incorporating moderate-density residential areas, open roads, a tunnel and a roundabout, offering valuable insights into GNSS performance and CAM messages behavior in transitional driving conditions.

The test route begins in a suburban residential area, where lower speed limits and frequent intersections require precise vehicle positioning and accurate heading estimation. As the vehicle progresses, it transitions into more open roads characterized by higher average speeds and reduced infrastructure interference, presenting an opportunity to assess GNSS stability in less obstructed conditions. A critical feature of this scenario is the roundabout, which poses a unique challenge for CAM-based cooperative awareness. The rapid lateral and angular movements within the roundabout emphasize the importance of yaw rate accuracy, heading changes, and acceleration consistency. Accurate GNSS updates in this phase ensure correct vehicle trajectory estimation, a fundamental requirement for cooperative maneuvering assistance and collision avoidance systems.

Furthermore, peri-urban settings often introduce partial GNSS obstructions due to vegetation, low-rise buildings, and road-side infrastructure. These elements, while less pronounced than urban canyons, still contribute to multipath effects and occasional signal degradation, particularly when transitioning between open and built-up areas. The recorded GNSS trace from this scenario, when replayed through TRACEN-X, provides a crucial validation tool to assess how well the OScar application maintains consistent CAM generation rates, accurate speed estimation, and reliable positioning data under such conditions.

4.3 Performance Assessment and Analysis

4.3.1 Overall CAM Generation and Distribution

The figure below illustrates the complete test route on a satellite map with color-coded markers indicating every CAM transmitted during the drive. Each color corresponds to the primary trigger condition for the message, namely: distance, velocity, heading, or timeout. One can observe how message triggers vary with the driving environment: for instance, where steering angles are more pronounced (e.g., at roundabouts), a greater number of heading-triggered CAMs are present.

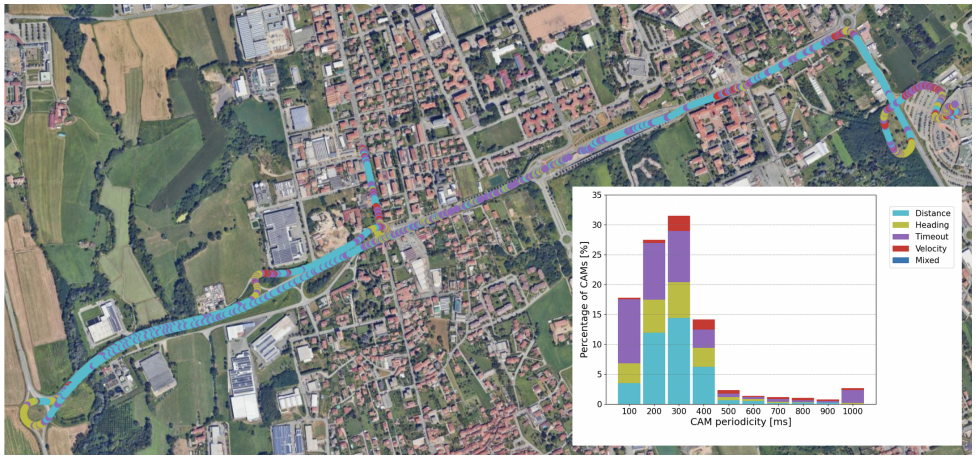

Figure 4.7: Satellite map of the test route with CAM messages sent color coded by sending reason (distance, heading, velocity, timeout)

A more quantitative view of CAM triggers is shown in the pie chart of the following figure, where the overall percentages of each trigger type are displayed. Distance-based triggers account for the largest share of messages (approximately 38%), followed closely by the timeout trigger (36%), while heading triggers and velocity triggers comprise roughly 19% and 7% respectively. This distribution underscores that although periodic messages (due to timeout) are common, significant numbers of CAMs are still generated by more dynamic triggers related to changes in position, heading, and speed.

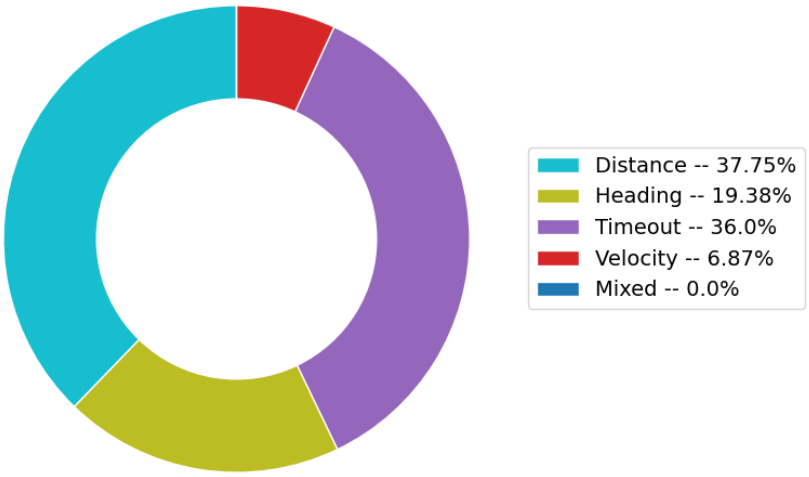

Figure 4.8: Percentage distribution of CAMs sent during the peri-urban test route, illustrated by trigger category

4.3.2 GNSS Fix Quality and Dead Reckoning

A notable feature of the test route is the short tunnel segment, chosen specifically to evaluate Dead Reckoning performance when direct satellite signals are unavailable. The fix data evaluation illustrated below, shows the vehicle's GNSS fix status throughout the route, distinguishing between RTK Fixed, RTK Float, Differential GNSS, Dead Reckoning, and 3D fixes. During the tunnel passage, the system must rely primarily on the onboard inertial sensors and wheel odometry to maintain an accurate position. In the figure, one sees the transition from an RTK Float or Differential GNSS fix to a Dead Reckoning solution upon entering the tunnel, followed by a smooth return to more accurate RTK states after re-emerging.

Figure 4.9: GNSS fix state evolution along the route

This behavior demonstrates that the OScar platform can maintain sufficient positional awareness in short-duration GNSS outages. Though accuracy slightly degrades in Dead Reckoning mode, the continuity of position estimates remains acceptable for cooperative awareness needs. Once satellite visibility is restored, the system re-establishes high-precision RTK fixing.

4.3.3 Yaw Rate Analysis in Roundabout

With the intent of evaluating the u-blox ZED-F9R sensor fusion capabilities, an additional graphic is included, measuring yaw rate data in order to have a correct and coherent vehicle behavior overview. As visible below near the roundabout to the very left, one observes a spike in messages triggered by heading changes and velocity variations. These triggers help ensure timely and accurate cooperative awareness during lane changes or merges, where small positioning or heading errors could adversely affect collision avoidance systems.

Figure 4.10: Sensor fusion yaw rate value along the test road.

4.3.4 Summary of Observations

Overall, the results show that:

- A significant proportion of CAMs are generated by distance and timeout triggers, though heading changes become predominant in roundabouts and sharper turns.
- The Dead Reckoning fallback strategy operates effectively in short GNSS absence situations, maintaining acceptable positioning until RTK or differential fixes are reacquired.
- Rapid heading fluctuations in roundabouts underscore the importance of accurate inertial data for cooperative awareness.

This mixed peri-urban scenario validates the OScar application's adaptability to different driving situations and highlights the importance of high-rate, low-latency data processing in ensuring that cooperative awareness messages remain both timely and contextually relevant, especially in complex road geometries where even slight deviations in position or orientation could compromise safety and interoperability in V2X scenarios.

Chapter 5

Conclusions

This thesis has explored the development of a dedicated serial data parsing solution for high-precision GNSS integration in Cooperative Intelligent Transport Systems. By leveraging the ArduSimple SimpleRTK2B receiver and the u-blox ZED-F9R module, the work demonstrates how raw GNSS, inertial, and RTK-corrected positioning data can be efficiently acquired, parsed, and injected into the OScar framework to provide real-time, centimeter-level vehicle state information. This integrated approach bridges fundamental gaps in vehicular communication by enabling accurate generation of ETSI-compliant awareness messages (e.g., CAMs), ensuring that even demanding peri-urban and mixed-driving scenarios are robustly handled.

Beyond simply interfacing with GNSS hardware, the project focuses on delivering consistent, thread-safe data delivery mechanisms that preserve the integrity of high-frequency updates. Parsing algorithms for both UBX and NMEA protocols have been thoroughly dissected, highlighting the value of strong validation steps, checksum checks, and buffer management. When coupled with the layered C-ITS architecture, this parsing solution allows the OScar application to broadcast real-time awareness data for enhanced safety, traffic optimization, and cooperative driving assistance.

Furthermore, controlled simulated local tests and real-world field tests confirm that both accuracy and reliability remain stable despite challenging environments such as tunnels, roundabouts, and partial satellite obstructions. The results also show how a flexible, open-source approach can accelerate the adoption of C-ITS, offering a lower barrier to entry for researchers, industry practitioners, and municipalities seeking to advance to next-generation transportation solutions. By unifying robust GNSS support with standardized V2X protocols, the OScar framework positions itself as a strong candidate for large-scale deployment, proving that true cooperative intelligence is both attainable and cost-effective.

5.1 Future Work

Although the current parsing module provides a solid foundation for high-precision GNSS data integration, its scalability can be pushed further. Future efforts will focus on extending the parser's compatibility with additional sensor types and message configurations, potentially incorporating supplementary GNSS constellations, advanced sensor fusion input (e.g., LiDAR or radar metadata), and customized proprietary UBX messages. The intent is to make the system more robust, supporting higher data rates and more dynamic error-handling capabilities, without compromising on real-time performance.

Another key direction lies in broadening the parser's adaptability across diverse hardware platforms. By making the serial parsing interface more configurable, the OScar project can accommodate an even wider range of GNSS and inertial sensor boards. This evolution will foster an ecosystem where the same open source software stack supports a broad variety of vehicles, sensor packages, and operational needs. Ultimately, the parser's continued refinement will further strengthen OScar's role as a cornerstone for safe, efficient, and future-ready C-ITS deployments.

Appendix A

UBX Protocol

A.1 History and Development of UBX

The UBX protocol was introduced by u-blox as a proprietary binary messaging format to overcome the limitations of standard protocols like NMEA in GNSS applications. Its origins trace back to u-blox's early focus on developing robust and high-performance navigation solutions. By designing a dedicated protocol, u-blox could manage advanced features such as real-time configuration, multi-constellation tracking, and detailed receiver diagnostics more efficiently than text-based formats. Over time, UBX has evolved to accommodate higher data rates, enhanced accuracy requirements, and new satellite constellations, maintaining its compact binary structure to ensure minimal overhead in data exchange. Today, the UBX protocol forms a foundational element of u-blox's receiver technology stack, facilitating precise, low-latency communication between GNSS modules and host systems across a broad range of applications, from automotive navigation to industrial asset tracking.

In its latest iteration, the UBX protocol has introduced several advanced features to enhance GNSS receiver performance. Notably, it now supports dual and multi-band signal processing, allowing receivers to utilize multiple frequency bands such as L1 and L5 for GPS, and E1 and E5 for Galileo. This capability significantly improves positioning accuracy and reduces time to first fix (TTFF) in challenging environments. Additionally, the protocol includes advanced interference and jamming detection mechanisms, providing detailed diagnostics and countermeasures to maintain reliable navigation data despite signal disruptions.

Enhanced security features have also been integrated to detect and counter spoofing attempts, ensuring data integrity for critical GNSS-dependent applications. Furthermore, the protocol now supports high-precision positioning techniques like Real Time Kinematics (RTK) and Precise Point Positioning (PPP), enabling centimeter-level accuracy by incorporating correction data from reference stations or global services. These

advancements collectively position the UBX protocol at the forefront of modern GNSS communication standards.

A.2 UBX Message Structure

According to the official u-blox documentation [27], each UBX message follows a well-defined binary structure designed to ensure reliable parsing and clear separation of message types. Two constant synchronization characters (0xB5 and 0x62) mark the beginning of every UBX message and help receivers detect message boundaries. Directly after these sync characters, each UBX message includes a one-byte "class" field and a one-byte "ID" field, which together uniquely identify the purpose and format of the data. The subsequent two-byte "length" field, stored in little-endian format, indicates the exact size of the payload section, allowing for precise extraction of message contents. The payload itself follows, containing information such as positioning data, configuration parameters, or status updates. At the end of the message, two cyclic redundancy check bytes (CK_A and CK_B) are appended. These checksum characters are calculated over both the class/ID fields and the payload, providing a straightforward but robust error detection mechanism that guards against corruption during transmission. Through this structure, the UBX protocol achieves compactness, modularity, and a clear definition of data boundaries, which simplifies the integration of high-precision GNSS functionality in embedded systems.

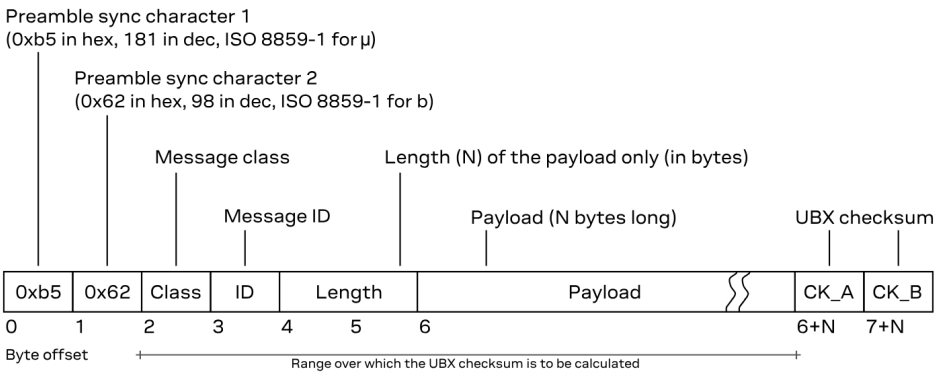

Figure A.1: UBX Message Structure

UBX Message Flow and Main Data Types

UBX messages flow back and forth between the GNSS receiver and the host system in a bidirectional manner, forming the basis of configuration, control, and data retrieval. Within the UBX payload, various data types are used to encapsulate device parameters or measurement results. Notably, the I4 data type represents a signed 32-bit integer (generally stored in little-endian format), convenient for values such as geographic coordinates

or velocity components that can be positive or negative. The U1 type is an 8-bit unsigned integer often used for small control parameters or counters. The U_8 bitfield is also an 8-bit entity but is usually interpreted as a collection of individual bits, each carrying a specific on/off or status-related meaning, making it especially handy for flags, booleans, or packed status indicators.

Name	Type	Size (Bytes)	Range	Resolution
U1	unsigned 8-bit integer	1	$0...2^8-1$	1
I1	signed 8-bit integer, two's complement	1	$-2^7...2^7-1$	1
X1	8-bit bitfield	1	n/a	n/a
U2	unsigned little-endian 16-bit integer	2	$0...2^{16}-1$	1
I2	signed little-endian 16-bit integer, two's complement	2	$-2^{15}...2^{15}-1$	1
X2	16-bit little-endian bitfield	2	n/a	n/a
U4	unsigned little-endian 32-bit integer	4	$0...2^{32}-1$	1
I4	signed little-endian 32-bit integer, two's complement	4	$-2^{31}...2^{31}-1$	1
X4	32-bit little-endian bitfield	4	n/a	n/a

Figure A.2: Relevant UBX data types

Checksum Calculation and Validation

As explained above, UBX messages conclude with a two-byte checksum that follows a straightforward implementation of the Fletcher algorithm. This particular implementation relies on two eight-bit accumulators, commonly referred to as CK_A and CK_B. Both accumulators begin at zero and are incrementally updated for every byte of the message payload that lies between the message class (the first byte in the body, right after the sync characters 0xB5 and 0x62) and the final payload byte. Formally, for each byte in this range, CK_A is updated by adding the current byte to its current value, and CK_B is then updated by adding the new CK_A value.

When the last byte of the payload has been processed, CK_A and CK_B each hold an eight-bit result. These two values are then appended to the message, with CK_A going first (thus forming the "lower" checksum byte) and CK_B following it ("upper" checksum byte). In effect, this yields a 16-bit checksum that can detect single-bit errors, swapped bytes, and other common forms of data corruption. Since every byte between the class field and the end of the payload is incorporated into the sums, the checksum is able to capture any alterations to the message contents. Furthermore, the Fletcher approach is computationally inexpensive, making it a popular choice for devices like GNSS receivers where real-time performance and reliability are both critical.

In the serial parser module, the task of verifying the authenticity of a UBX message is taken care of by the `validateUbxMessage()` function, which systematically verifies the integrity of the UBX message by computing and comparing its checksum values across the relevant portion of the data following the 8-bit Fletcher Algorithm explained earlier.

Once this process completes, the function checks whether the final two bytes of the message match the calculated values of CK_A and CK_B. If they coincide, the function marks the message as valid and returns true, thereby guaranteeing that subsequent processing receives a correctly formed UBX message. Otherwise, if the values do not match, the function determines that the message is invalid and returns `false`, thus preventing corrupted or incomplete data from propagating further in the system.

```
bool UBXNMEAParserSingleThread::validateUbxMessage(std::vector<uint8_t>
    msg) {
     uint8_t CK_A = 0x00;
     uint8_t CK_B = 0x00;

     for (long unsigned int i = 0; i < msg.size(); i++) {
         if (i >=2 && i <= msg.size() -3) { //checksum calculation range
             CK_A = CK_A + msg[i];
             CK_B = CK_B + CK_A;

             CK_A &= 0xFF;
             CK_B &= 0xFF;
         }
     }

     if (msg[msg.size() -2] == CK_A && msg[msg.size()-1] == CK_B) {
         return true;
     }
     else {
         //printf("\n\nINVALID MESSAGE: CK_A: %02X  CK_B: %02X\n\n", CK_A
    ,CK_B);
         //printUbxMessage(msg);
         return false;
     }
}
```

Figure A.3: `validateUbxMessage()` method

Parsing Algorithm

The logic for reading and parsing UBX messages makes use of an open source serial library [31], which was specifically chosen to assist the serial parser's development. After reading the latest two bytes from the serial buffer, the parser stores them in the variables `byte` and `byte_previous`. This approach allows the code to verify whether the pair of bytes corresponds to the sought UBX message header. In other words, whenever `byte_previous` equals 0xB5 and `byte` equals 0x62, the code recognizes the start of a potential UBX message and proceeds further.

In addition, two crucial boolean flags control the flow of the parsing process for UBX messages:

- `started_ubx`: Indicates whether the parsing of a UBX message has begun.

- `started_ubx_overlapped`: Indicates whether a newly detected header occurs within the body of another UBX message, thus signaling an overlapping message.

By using these flags, the parser can handle both standard UBX messages and situations where one UBX message may partially overlap with another. This is especially useful when high-throughput serial data streams arrive without strict separation between distinct messages. Once recognized, the bytes that compose any potential UBX message are temporarily stored in vectors of type `uint8_t`. Two such vectors: `ubx_message` and `ubx_message_overlapped`, preserve the incoming data until they have been either validated as correct UBX messages or identified as invalid. As soon as validation completes, any valid UBX data proceeds to the next processing stage (parsing specific payloads), and the corresponding buffers are cleared.

The parser also provides a wrong input recognition, which stores any byte that does not belong neither to a UBX message nor to a NMEA sentence in an appropriate vector called `wrong_input`. Once the size of this vector reaches a maximum threshold defined by the user, the parser will raise a flag, terminating the execution of OScar. In terms of C++ code, the UBX reading logic can be analyzed in different sections, starting from the initial reading phase that seeks a correct UBX message header:

```
[...]
// UBX message reading and parsing
if (started_ubx == false) {
    if (byte_previous == m_UBX_HEADER[0] && byte == m_UBX_HEADER[1]) {
        expectedLength = 0;
        ubx_message.push_back(byte_previous);
        ubx_message.push_back(byte);
        wrong_input.clear();
        started_ubx = true;

        byte_previous = byte;
        continue;
    }
}
[...]
```

Figure A.4: Parsing algorithm initial reading phase

The second reading phase is triggered when a correct message header is found and the `started_ubx` flag becomes true, at this moment, the parser starts storing bytes in the appropriate array, after performing the checks to determine if the parser is reading an overlapped message or not.

```
[...]
else { // started_ubx is true
    // Check if B5 62 is part of the current message or if it is a new
   message
    // We enter here if we enounter B5 62 inside a UBX message (it could
    be to store a normal value, or it
    // could be due to an overlapped message due to some issue with the
   serial device)
    if (byte_previous == m_UBX_HEADER[0] && byte == m_UBX_HEADER[1]) {
        if (ubx_message.size() <= 4) {
            // std::cerr << "length not reached!" << std::endl; //debug
            //printf("byte: %02X byte_previous: %02X\n", byte,
   byte_previous);
            //printUbxMessage(ubx_message);
            ubx_message.clear();
            ubx_message.push_back(byte_previous);
            ubx_message.push_back(byte);
            wrong_input.clear();

            byte_previous = byte;
            continue;
        }

        // If a B5 62 sequence is read in the middle of the message,
   start storing bytes in a separate vector
        // as this may be an overlapped message
        // We read the potentially overlapped message in parallel with
   the previous one, and then we check both
        // for validity
        expectedLengthOverlapped = 0;
        ubx_message_overlapped.clear();
        ubx_message_overlapped.push_back(byte_previous);
        ubx_message_overlapped.push_back(byte);
        ubx_message.push_back(byte);
        wrong_input.clear();
        started_ubx_overlapped = true;

        byte_previous = byte;
        continue;
    }
    // started_ubx is true: store the current byte and reset the
   wrong_input vector
    ubx_message.push_back(byte);
    wrong_input.clear();
[...]
```

Figure A.5: Parsing algorithm's second message recognition phase

The third reading phase takes care of checking that a correct message has been parsed, verifying that the length read from the serial buffer actually corresponds to the length of the populated message array. If this initial verification is passed, the message undergoes to the validation phase through the `validateUbxMessage()` method illustrated earlier, in order to be sent to the parsing phase, which will take care of identifying what kind of

message it is and forwarding it accordingly to the parsing functions.

```
[...]
if (ubx_message.size() == 6) {
    sprintf(expectedLengthStr.data(), "%02X%02X", ubx_message[5],
   ubx_message[4]);
    expectedLength = std::stol(expectedLengthStr, nullptr, 16) + 8;
    expectedLengthStr.clear();
}

// We enter here when we have finished reading
// Any message with expectedLength = 0 is considered invalid and ignored
if (expectedLength > 0 && ubx_message.size() == expectedLength) {
    if (started_ubx_overlapped == false) {
        // If we don't have any potentially overlapped message, validate
    the current message, checking the checksum, and parse it
        if (validateUbxMessage(ubx_message) == true) {
            // Configuration messages
            // If we receive a UBX-ACK-NAK message, we terminate the
   parser as there is a problem
            if (ubx_message[2] == 0x05 && ubx_message[3] == 0x00) {
                std::cerr << "Configuration Error: UBX-ACK-NAK received.
    Terminating." << std::endl;
                // *m_terminatorFlagPtr=true;
            }
[...]
```

Figure A.6: Length checking and Message validation phases

A.3 Relevant UBX Messages

Among the vast range of data provided by the UBX protocol, some messages were identified as particularly relevant to the scope of obtaining crucial data for the OScar project because of their content. These messages were thoroughly analyzed, and specific parsing methods were developed to extract meaningful information to deliver into the OScar framework and to enable advanced processing and validation functionalities. The selected messages can be categorized primarily into two main classes:

- **Navigation (NAV) messages:** NAV messages provide precise and comprehensive GNSS-based navigation data, including the vehicle's absolute geographical position (latitude, longitude, altitude), velocity, timing information, and orientation (roll, pitch, heading). They also convey the receiver's operational status, such as GNSS fix validity, differential corrections, and signal quality indicators. These messages are fundamental for an accurate description of a vehicle's real-time state within Intelligent Transport Systems.
- **External Sensor fusion (ESF) messages:** ESF messages integrate GNSS-derived

navigation data with external sensor measurements from inertial units (accelerometers, gyroscopes) and odometers. This sensor fusion enhances navigation accuracy and robustness, especially in challenging environments with limited or degraded GNSS signal reception. ESF messages supply both raw sensor data and processed inertial navigation solutions, enabling reliable position estimation even during GNSS interruptions, thus greatly improving vehicular state estimation for cooperative vehicular communication.

Below is provided a comprehensive and detailed overview of every message parsed, along with the functions that implement the parsing algorithm.

A.3.1 UBX-NAV

The UBX-NAV message class contains detailed navigation-related information, crucial for accurate vehicular positioning, velocity estimation, and orientation awareness. These messages provide insights into positional accuracy, temporal synchronization and navigation status, enabling real-time updates of the vehicle's state. The NAV messages analyzed in this thesis include UBX-NAV-PVT, UBX-NAV-ATT, and UBX-NAV-STATUS messages, each serving a distinct purpose within the navigation data acquisition pipeline.

UBX-NAV-PVT

The UBX-NAV-PVT (Position, Velocity, and Time) message is one of the most comprehensive navigation messages provided by the u-blox GNSS modules. It integrates position coordinates (latitude, longitude, and height), velocity components in three-dimensional space, precise timing information, as well as fix quality and fix validity flags. By parsing this message, the receiving system obtains crucial real-time insights into the vehicle's absolute position and velocity, alongside GNSS-based timing. This message plays a fundamental role in the OScar framework's operation, as accurate vehicular localization directly influences the precision of Cooperative Awareness Messages (CAMs). Consequently, a dedicated parsing function called `parseNavPvt()` was implemented to extract and use the position and velocity data, which are subsequently used for updating the vehicle's dynamic parameters within the vehicular communication protocol.

The `parseNavPvt()` function algorithm code flow, as well as for the other parsing functions, can be described by dividing it in two subsections: data gathering and timing updates, since the parser has to keep track of the age of information and of the parsing period of every data handled.

Every parsing method starts with the atomic buffer loading, performed making use of the `.load()` method from `std::atomic`, then the part of the UBX message the contains the sought information is found leveraging the use of the `m_UBX_PAYLOAD_OFFSET` constant that simplifies the process since it is aligned with the interface description document that

describes the content of the message in detail. In the case of the UBX-NAV-PVT message, for example, the speed value is found at offset position 60 and the heading value at offset position 64. After finding the right position in the read message, an endianness conversion from the big-endian format to little-endian is performed and, based on the documentation specifications, a final value conversion is calculated before proceeding with the second part of the algorithm.

```
void UBXNMEAParserSingleThread::parseNavPvt(std::vector<uint8_t>
    response) {
  // Load the atomic data buffer
  out_t out_pvt = m_outBuffer.load();

  // Extract the speed value
  std::vector<uint8_t> sog(response.begin() + m_UBX_PAYLOAD_OFFSET + 60,
     response.begin() + m_UBX_PAYLOAD_OFFSET + 64);
      // Convert to little endian
  std::reverse(sog.begin(),sog.end());
     // Cast and convert units
  out_pvt.sog = static_cast<double>(hexToSigned(sog)) * 0.001; //
    Converts from mm/s (UBX unit) to m/s
     out_pvt.sog_ubx = out_pvt.sog;

     // Extract the course over ground (byte 64) - 4 bytes
  std::vector<uint8_t> head_motion(response.begin() +
    m_UBX_PAYLOAD_OFFSET + 64, response.begin() + m_UBX_PAYLOAD_OFFSET +
    68);
     // Convert to little endian
  std::reverse(head_motion.begin(),head_motion.end());
     // Cast and convert units
  out_pvt.cog = static_cast<double>(hexToSigned(head_motion)) * 0.00001;
     // Converts from 0.0001 degrees (UBX unit) to degrees
     out_pvt.cog_ubx = out_pvt.cog;

     // Extract the latitude (byte 28) - 4 bytes
     std::vector<uint8_t> lat(response.begin() + m_UBX_PAYLOAD_OFFSET +
    28, response.begin() + m_UBX_PAYLOAD_OFFSET + 32);
     std::reverse(lat.begin(),lat.end());
     out_pvt.lat = static_cast<double>(hexToSigned(lat)) * 0.0000001; //
    Converts from 0.1 microdegrees (UBX unit) to degrees
     out_pvt.lat_ubx = out_pvt.lat;

     // Extract the longitude (byte 24) - 4 bytes
     std::vector<uint8_t> lon(response.begin() + m_UBX_PAYLOAD_OFFSET +
    24, response.begin() + m_UBX_PAYLOAD_OFFSET + 28);
     std::reverse(lon.begin(),lon.end());
     out_pvt.lon = static_cast<double>(hexToSigned(lon)) * 0.0000001; //
    Converts from 0.1 microdegrees (UBX unit) to degrees
     out_pvt.lon_ubx = out_pvt.lon;

     // Extract the altitude (byte 36) - 4 bytes
     std::vector<uint8_t> alt(response.begin() + m_UBX_PAYLOAD_OFFSET +
    36, response.begin() + m_UBX_PAYLOAD_OFFSET + 40);
     std::reverse(alt.begin(),alt.end());
     out_pvt.alt = static_cast<double>(hexToSigned(alt)) * 0.001; //
    Convert from mm (UBX unit) to m
     out_pvt.alt_ubx = out_pvt.alt;
```

Figure A.7: UBX-NAV-PVT data parsing code flow

The next part of the algorithm takes care of recording the timestamp of the data parsing, updating the "last update" values stored in the atomic buffer. It also updates the period of the message and acts on the validation flags of interest in order to confirm that the information in question is valid and usable. This last operation results to be of particular interest because every time an information is needed, for example when filling a CAM message, the parser taps from the atomic data buffer and performs a validity check before giving away the information.

```
    // Compute the period ("prd") occurred between this update and the
   last update
    if (m_debug_age_info_rate) {
        m_debug_age_info.prd_sog_cog = update.time_since_epoch().count()
    - out_pvt.lu_sog_cog;
        m_debug_age_info.prd_sog_cog_ubx = update.time_since_epoch().
   count() - out_pvt.lu_sog_cog_ubx;
        m_debug_age_info.prd_pos = update.time_since_epoch().count() -
   out_pvt.lu_pos;
        m_debug_age_info.prd_pos_ubx = update.time_since_epoch().count()
    - out_pvt.lu_pos_ubx;
        m_debug_age_info.prd_alt = update.time_since_epoch().count() -
   out_pvt.lu_alt;
        m_debug_age_info.prd_alt_ubx = update.time_since_epoch().count()
    - out_pvt.lu_alt_ubx;
    }

    // Set the last update (lu) time to now for all the information
   parsed from UBX-NAV-PVT
  out_pvt.lu_sog_cog = update.time_since_epoch().count();
  out_pvt.lu_sog_cog_ubx = out_pvt.lu_sog_cog;
  out_pvt.lu_pos = update.time_since_epoch().count();
  out_pvt.lu_pos_ubx = out_pvt.lu_pos;
    out_pvt.lu_alt = update.time_since_epoch().count();
    out_pvt.lu_alt_ubx = out_pvt.lu_alt;

    // Validates data
    // TODO: this can be potentially removed in the future
    m_sog_valid = true;
    m_sog_valid_ubx = true;
    m_cog_valid = true;
    m_cog_valid_ubx = true;
    m_pos_valid = true;
    m_pos_valid_ubx = true;
    m_alt_valid = true;
    m_alt_valid_ubx = true;

    // Updates the buffer
    m_outBuffer.store(out_pvt);
}
```

Figure A.8: UBX-NAV-PVT time updates and validation logic

UBX-NAV-ATT

The UBX-NAV-ATT (Attitude) message contains detailed information regarding the orientation of the vehicle in space. Specifically, it provides the roll, pitch, and heading angles, along with their respective accuracies. Such data is critical for applications where precise vehicle orientation is required, such as: advanced driver assistance systems, augmented navigation systems, and cooperative perception scenarios. The availability of reliable orientation data allows the OScar framework to effectively incorporate attitude information into CAMs, enhancing cooperative awareness among participating vehicles.

The `parseNavAtt()` method parses navigation attitude data (roll, pitch, and heading) from a vector of bytes. It uses a fixed payload offset and reads specific groups of bytes corresponding to each attitude parameter. For each parameter, it reverses the byte order (to convert from little-endian to big-endian), converts the byte group to a signed integer, scales the value, and stores it in the atomic data structure. The method also records the current date and time, calculates the time elapsed since the last update, validates the data, and updates the output buffer accordingly.

```
void UBXNMEAParserSingleThread::parseNavAtt(std::vector<uint8_t>
    response) {
     const int ATT_DATA_START_INDEX = 8;
     const int ATT_DATA_END_INDEX = 20 + m_UBX_PAYLOAD_OFFSET;
     int offset = m_UBX_PAYLOAD_OFFSET + ATT_DATA_START_INDEX;
     std::vector<uint8_t> att_data;

     out_t out_att = m_outBuffer.load();

     for(int i = offset; i <= ATT_DATA_END_INDEX; i++) {
         if (i == 18) {
             std::reverse(att_data.begin(), att_data.end());
             out_att.roll = static_cast<double>(hexToSigned(att_data)) *
    0.00001;
             att_data.clear();
         } else if (i == 22) {
        std::reverse(att_data.begin(), att_data.end());
        out_att.pitch = static_cast<double>(hexToSigned(att_data)) *
    0.00001;
        att_data.clear();
     } else if (i == 26) {
        std::reverse(att_data.begin(), att_data.end());
        out_att.heading = static_cast<double>(hexToSigned(att_data)) *
    0.00001;
        att_data.clear();
        break;
     }
     att_data.push_back(response[i]);
  }

  std::time_t now = std::chrono::system_clock::to_time_t(std::chrono::
    system_clock::now());
  strcpy(out_att.ts_att, std::ctime(&now));

  auto update = time_point_cast<microseconds>(system_clock::now());

     if (m_debug_age_info_rate) {
         m_debug_age_info.prd_att = update.time_since_epoch().count() -
    out_att.lu_att;
     }

  out_att.lu_att = update.time_since_epoch().count();
  m_att_valid = true;
  m_outBuffer.store(out_att);
}
```

Figure A.9: `parseNavAtt()` method

UBX-NAV-STATUS

The UBX-NAV-STATUS message provides essential information regarding the GNSS receiver's operational state. This includes the GNSS fix type (e.g., no fix, dead reckoning,

2D, 3D fix), differential correction status, and various flags indicating the availability and quality of navigation data. Accurate interpretation of this message allows the OScar framework to dynamically adapt its data handling and message generation logic based on the GNSS status, thus ensuring that Cooperative Awareness Messages (CAMs) are only transmitted when valid positional data is available. A specific parser function was implemented for the NAV-STATUS message to enable robust monitoring of GNSS data validity and reliability, improving the overall robustness of vehicular communication within the ITS framework.

The `parseNavStatus()` method extracts the fix mode and fix flags from a response byte vector to determine the navigation fix status. It then updates the atomic output buffer with a string representing the fix type (e.g., "NoFix", "2D-Fix", "3D-Fix", etc.) and sets flags indicating the validity of 2D and 3D fixes. If the fix flags indicate an invalid fix, it immediately marks the status as "Invalid" and exits.

```
    void UBXNMEAParserSingleThread::parseNavStatus(std::vector<uint8_t>
   response) {
    uint8_t fix_mode  = response[m_UBX_PAYLOAD_OFFSET + 4];
    uint8_t fix_flags = response[m_UBX_PAYLOAD_OFFSET + 4 + 1];

    out_t out_sts = m_outBuffer.load();

    if (fix_flags < 0xD0) {
        strcpy(out_sts.fix_ubx, "Invalid");
        m_2d_valid_fix = false;
        m_3d_valid_fix = false;
        m_outBuffer.store(out_sts);
        return;
    }

    if (fix_mode == 0x00) {
        strcpy(out_sts.fix_ubx, "NoFix");
        m_2d_valid_fix = false;
        m_3d_valid_fix = false;
    }
    else if (fix_mode == 0x01) {
        strcpy(out_sts.fix_ubx, "Estimated/DeadReckoning");
        m_2d_valid_fix = true;
        m_3d_valid_fix = true;
    }
    else if (fix_mode == 0x02) {
        strcpy(out_sts.fix_ubx, "2D-Fix");
        m_2d_valid_fix = true;
        m_3d_valid_fix = false;
    }
    else if (fix_mode == 0x03) {
        strcpy(out_sts.fix_ubx, "3D-Fix");
        m_2d_valid_fix = true;
        m_3d_valid_fix = true;
    }
    else if (fix_mode == 0x04) {
        strcpy(out_sts.fix_ubx, "GPS+DeadReckoning");
        m_2d_valid_fix = true;
        m_3d_valid_fix = true;
    }
    else if (fix_mode == 0x05) {
        strcpy(out_sts.fix_ubx, "TimeOnly");
        m_2d_valid_fix = false;
        m_3d_valid_fix = false;
    }
    else {
        strcpy(out_sts.fix_ubx, "Unknown/Invalid");
        m_2d_valid_fix = false;
        m_3d_valid_fix = false;
    }

    m_outBuffer.store(out_sts);
}
```

Figure A.10: Length checking and Message validation phases

A.3.2 UBX-ESF

The UBX-ESF (External Sensor Fusion) message class provides sensor fusion data, obtained by combining GNSS measurements with external sensor inputs such as inertial measurement units (IMU). The availability of fused sensor data greatly enhances navigation accuracy, particularly in challenging GNSS environments. The ESF messages considered relevant to the work presented in this thesis include ESF-INS (Inertial Navigation Solution) and ESF-RAW (Raw Sensor Data). Detailed parsers were created for these messages to facilitate extraction and analysis of inertial navigation and sensor measurement data.

UBX-ESF-INS

The UBX-ESF-INS message delivers a comprehensive inertial navigation overview, combining data from inertial sensors and GNSS measurements. It provides real-time vehicle dynamics parameters, such as sensor-based position increments, orientation rates, and sensor calibration states. Parsing and interpreting the ESF-INS message allows the system to reliably estimate the vehicle's state even during short periods of GNSS outages or degraded GNSS reception. This capability significantly improves the reliability of vehicular communication and cooperative awareness in urban environments characterized by GNSS signal interruptions.

The `parseEsfIns()` method extracts compensated angular rates and compensated accelerations for all the three axes from a response byte vector. It processes each measurement by extracting the corresponding bytes, reversing their order to account for endianness, converting them to signed integers, and scaling them to obtain values in physical units (deg/s for angular rates and m/s^2 for accelerations). The method then records the current timestamp, computes the elapsed time since the last update if debugging is enabled, marks the data as valid, and stores the updated output buffer.

```
    void UBXNMEAParserSingleThread::parseEsfIns(std::vector<uint8_t>
   response) {
    out_t out_ins = m_outBuffer.load();

    std::vector<uint8_t> comp_ang_rate_x(response.begin() +
   m_UBX_PAYLOAD_OFFSET + 12, response.begin() + m_UBX_PAYLOAD_OFFSET +
   16);
    std::reverse(comp_ang_rate_x.begin(), comp_ang_rate_x.end());
    out_ins.comp_ang_rate_x = static_cast<double>(hexToSigned(
   comp_ang_rate_x)) * 0.001;

    [...]

    std::vector<uint8_t> comp_acc_x(response.begin() +
   m_UBX_PAYLOAD_OFFSET + 24, response.begin() + m_UBX_PAYLOAD_OFFSET +
   28);
    std::reverse(comp_acc_x.begin(), comp_acc_x.end());
    out_ins.comp_acc_x = static_cast<double>(hexToSigned(comp_acc_x)) *
   0.01;

    [...]

    std::time_t now = std::chrono::system_clock::to_time_t(std::chrono::
   system_clock::now());
    strcpy(out_ins.ts_comp_acc, std::ctime(&now));
    strcpy(out_ins.ts_comp_ang_rate, std::ctime(&now));

    auto update = time_point_cast<microseconds>(system_clock::now());

    if (m_debug_age_info_rate) {
        m_debug_age_info.prd_comp_acc = update.time_since_epoch().count
   () - out_ins.lu_comp_acc;
        m_debug_age_info.prd_comp_ang_rate = update.time_since_epoch().
   count() - out_ins.lu_comp_ang_rate;
    }

    out_ins.lu_comp_acc = update.time_since_epoch().count();
    out_ins.lu_comp_ang_rate = update.time_since_epoch().count();

    m_comp_acc_valid = true;
    m_comp_ang_rate_valid = true;

    m_outBuffer.store(out_ins);
}
```

Figure A.11: `parseEsfIns()` method

UBX-ESF-RAW

The UBX-ESF-RAW message provides direct access to raw sensor measurements from external sensors integrated into the u-blox system, such as accelerometers, gyroscopes,

and odometers. These measurements include sensor values along each axis, as well as precise timestamps that allow synchronization of sensor data with GNSS measurements. The parsing of ESF-RAW messages facilitates advanced sensor fusion algorithms and enables developers to implement custom inertial navigation and dead reckoning techniques. Within the scope of the OScar project, an ESF-RAW message parser was constructed to systematically extract sensor data and enable deeper analysis of vehicle dynamics, thus significantly augmenting the capability of the vehicular communication framework in accurately estimating vehicle states under diverse operational conditions.

The `parseEsfRaw()` function processes raw ESF sensor data from a response byte vector. It skips the reserved header bytes and reads 24 bytes of sensor data in groups of 8. For each 8-byte group, it checks the data type identifier at the fourth byte and, based on the type (0x10, 0x11, or 0x12), extracts 3 bytes of acceleration data. The method reverses the byte order to correct for endianness, converts the data to a signed integer, and scales it by dividing by 1024 to obtain the corresponding acceleration value for the x, y, or z axis. Finally, it updates the timestamp, calculates the elapsed time since the last update if debugging is enabled, marks the acceleration data as valid, and stores the updated output buffer.

```
    void UBXNMEAParserSingleThread::parseEsfRaw(std::vector<uint8_t>
   response) {
    int offset = m_UBX_PAYLOAD_OFFSET + 4;
    int j = 0;
    std::vector<uint8_t> esf_data;
    out_t out_esf = m_outBuffer.load();

    for (int i = offset; i < offset + 24; i++) {
        esf_data.push_back(response[i]);
        j++;
        if (j == 8) {
            if (esf_data[3] == 0x10) {
                esf_data.erase(esf_data.begin() + 3, esf_data.end());
                std::reverse(esf_data.begin(), esf_data.end());
                out_esf.raw_acc_x = static_cast<double>(hexToSigned(
   esf_data)) / 1024;
                esf_data.clear();
                j = 0;
            } else if (esf_data[3] == 0x11) {
                esf_data.erase(esf_data.begin() + 3, esf_data.end());
                std::reverse(esf_data.begin(), esf_data.end());
                out_esf.raw_acc_y = static_cast<double>(hexToSigned(
   esf_data)) / 1024;
                esf_data.clear();
                j = 0;
            } else if (esf_data[3] == 0x12) {
                esf_data.erase(esf_data.begin() + 3, esf_data.end());
                std::reverse(esf_data.begin(), esf_data.end());
                out_esf.raw_acc_z = static_cast<double>(hexToSigned(
   esf_data)) / 1024;
                esf_data.clear();
                j = 0;
            }
        }
    }

    std::time_t now = std::chrono::system_clock::to_time_t(std::chrono::
   system_clock::now());
    strcpy(out_esf.ts_acc, std::ctime(&now));

    auto update = time_point_cast<microseconds>(system_clock::now());
    if (m_debug_age_info_rate) {
        m_debug_age_info.prd_acc = update.time_since_epoch().count() -
   out_esf.lu_acc;
    }
    out_esf.lu_acc = update.time_since_epoch().count();
    m_acc_valid = true;
    m_outBuffer.store(out_esf);
}
```

Figure A.12: `parseEsfRaw()` method

Appendix B

NMEA Protocol

B.1 NMEA Protocol Overview

The National Marine Electronics Association (NMEA) protocol is a widely adopted standard for communication between electronic marine instruments, GPS receivers, and navigation software. Created primarily to address interoperability among marine electronic equipment, the NMEA standard ensures that devices from diverse manufacturers can communicate clearly, reliably, and consistently. The protocol defines a structured ASCII-based sentence format through which navigation-related data is encoded and transmitted. Each NMEA sentence begins with a dollar sign ($), is followed by a two-letter talker identifier (e.g., GP for GPS devices), a three-letter sentence identifier (e.g., RMC for Recommended Minimum Specific GPS/Transit Data), and a series of comma-separated data fields. It concludes with an asterisk (*) followed by a two-character hexadecimal checksum, computed as an XOR of all characters between $ and * to ensure data integrity.

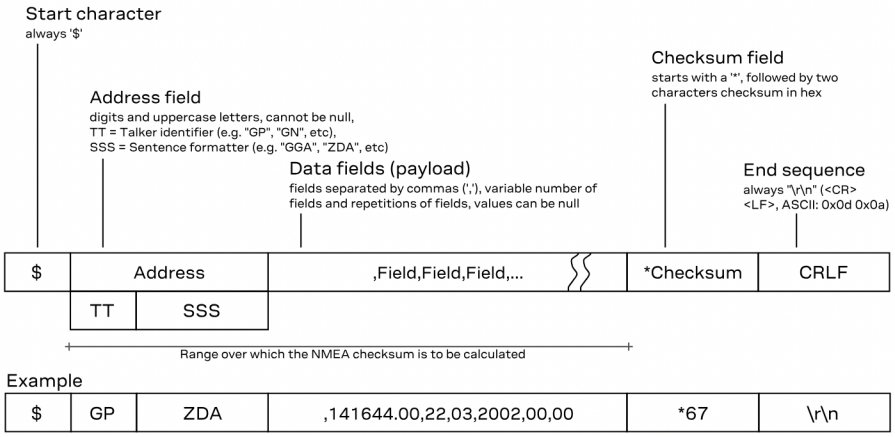

Figure B.1: NMEA sentence structure [27]

NMEA sentences are typically transmitted at regular intervals, commonly at a rate of one per second, although higher update rates may be supported by some devices and applications. The simplicity of the ASCII-based structure makes the NMEA protocol exceptionally versatile, allowing easy parsing, debugging, and integration into navigation and positioning systems.

Origin and Standardization

The origins of the NMEA protocol date back to 1957, with the establishment of the National Marine Electronics Association in the United States. The initial goal of the organization was to foster standardization and cooperation within the marine electronics industry, which was characterized at that time by proprietary and incompatible communication protocols. Throughout the 1980s and 1990s, as Global Positioning System (GPS) technology became increasingly accessible, the NMEA 0183 standard emerged as the state of the art communication standard for marine electronic systems. Officially introduced by NMEA in 1983, NMEA 0183 was a significant milestone, defining electrical signal characteristics and sentence structures, thereby promoting widespread adoption and interoperability.

Today, multiple versions of NMEA standards coexist, notably NMEA 0183 and its successor, NMEA 2000. While NMEA 0183 remains highly popular due to its simplicity and compatibility with legacy systems, NMEA 2000 is designed to offer higher data rates, multi-point connections, and greater robustness, reflecting evolving technology demands in modern navigation systems. The continuous development and updating of these standards are supervised and regulated by the NMEA organization, ensuring that the protocol remains relevant and responsive to technological advancements, regulatory changes, and evolving industry practices.

Figure B.2: NMEA protocols evolution [23]

Key Differences from UBX

NMEA and UBX protocols, despite serving similar purposes in communicating navigation data, differ significantly in structure, complexity, and intended usage scenarios:

- **Sentence Structure and Format**: NMEA sentences use a standardized, human-readable ASCII format, beginning with a dollar sign and terminating with a checksum. This facilitates ease of parsing and debugging. Conversely, the UBX protocol employs a proprietary binary format optimized for efficient, compact data representation and increased data throughput. UBX messages typically contain headers, message classes, IDs, length indicators, payload data, and checksums, necessitating specific parsers to interpret the binary data stream accurately.
- **Data Throughput and Efficiency**: Given their ASCII nature, NMEA sentences are larger and less efficient in terms of bandwidth usage compared to UBX messages. UBX's binary format significantly reduces the size and overhead of transmitted data, allowing for higher data rates and a more extensive array of detailed and specific navigation-related messages.
- **Customizability and Vendor-Specific Extensions**: NMEA sentences are well-defined and standardized, providing limited flexibility. Vendor-specific sentences can be defined but may vary significantly across different implementations, often complicating interoperability. UBX, however, provides an extensive and systematic protocol structure with clearly-defined message classes and subclasses, allowing for advanced configurations and customizations specifically tailored for high-precision positioning and timing applications provided by u-blox modules.
- **Typical Usage Scenarios**: NMEA is prevalent in general navigation scenarios, both marine and terrestrial, due to its broad acceptance, simplicity, and readability. UBX, on the other hand, is particularly suitable for specialized applications requiring high precision, rapid update rates, and detailed GNSS receiver status and configuration data, such as in automotive, robotics, and drone-based systems.

NMEA Protocol Configuration

U-blox receivers provide configurable parameters allowing customization of NMEA message output for specific applications. Parameters include protocol version selection (e.g., NMEA 4.11, 4.10, 4.00, 2.3, or 2.1), message filtering to handle invalid or unknown data, GNSS-specific message controls, and sentence-length limitations. Special modes, such as compatibility mode (ensuring legacy system support), high-precision mode (increasing decimal precision for latitude and longitude), and multi-GNSS output modes, are also configurable. Such flexibility supports various integration requirements, from legacy marine equipment to high-precision land navigation systems.

NMEA Key Data Fields

Data fields within NMEA sentences follow standardized formats and require precise interpretation rules. These fields include:

- **Latitude and Longitude**: expressed in degrees, minutes, and decimal fractions of minutes (DDMM.MMMM), require conversion for certain applications (e.g., degrees-minutes-seconds or decimal degrees). The parser provides a specific utility function that performs this operation:

```
double UBXNMEAParserSingleThread::decimal_deg(double value, char
    quadrant) {
     // Concatenate int part and dec part
     int degrees = floor(value/100);
     double minutes = value - (degrees*100);
     value = degrees + minutes/60;

     // Value is positive when the quadrant is 'N' or 'E'
     if(quadrant=='W' || quadrant=='S') {
         value = -value;
     }

     return value;
}
```

Figure B.3: Position data conversion function

- **Position Fix Indicators**: indicate the type and validity of GNSS fixes (e.g., autonomous fix, differential fix, RTK float or fixed, dead reckoning status).
- **Satellite and Signal Numbering**: satellite identifiers differ depending on the GNSS system and NMEA protocol version, with strict and extended numbering schemes defined to accommodate newer GNSS systems.
- **Extra Data Fields**: starting from NMEA 4.10, extra fields such as signal identifiers and navigation status are introduced to support enhanced multi-GNSS interoperability and high-precision positioning.
- **Invalid or Unknown Data**: standard practices for handling missing, invalid, or unknown data involve the output of empty fields or designated placeholders, depending on device configuration, ensuring consistent message parsing.

These detailed definitions ensure accurate interpretation and integration across diverse applications, from simple position logging to complex multi-GNSS navigation solutions.

B.2 Relevant NMEA Sentences

For the sake of the project, the NMEA protocol is used as a backup data gathering source, keeping UBX the first choice since it provides more complete and detailed data. For this reason, only the fundamental NMEA sentences have been covered, even though the software module has been developed with a scalable approach so that, if needed, support for other sentences can be easily added.

GNS - GNSS Fix Data

The GNS sentence provides essential positioning data from multiple Global Navigation Satellite Systems. It contains position fixes, time stamps, number of satellites used, HDOP (Horizontal Dilution of Precision), altitude, geoid separation, and positioning status indicators. It is especially useful for hybrid GNSS systems integrating GPS, GLONASS, Galileo, or other navigation constellations.

An example of a standard GNS sentence is shown below:

```
$GNGNS,123519,4807.038,N,01131.000,E,DA,08,0.9,545.4,46.9,,0000*6E
```

The fields represent:

- **123519**: Fix taken at 12:35:19 UTC
- **4807.038, N**: Latitude 48 deg 07.038' North
- **01131.000, E**: Longitude 11 deg 31.000' East
- **DA**: Mode indicators (e.g., Differential mode, Autonomous)
- **08**: Number of satellites used
- **0.9**: HDOP
- **545.4**: Altitude (meters) above mean sea level
- **46.9**: Geoid separation (meters)
- **0000**: DGPS Station ID (blank if unused)

The parsing function of the GNS sentence is called `parseNmeaGns()` and its algorithm extracts data to determine the vehicle position and GNSS fix status. It starts by atomically retrieving the current output buffer and splits the incoming NMEA sentence into individual fields and then latitude, longitude, fix mode, and altitude fields are extracted and processed into numeric values. The function identifies the GNSS fix mode and determines a single representative fix status (e.g., NoFix, AutonomousGNSSFix, RTK-Fixed). The latitude, longitude, and altitude are converted into numeric formats, handling unavailable values appropriately. Finally, the function generates a timestamp, updates internal validity flags, and stores the parsed data atomically back into the shared buffer.

```
out_t out_nmea = m_outBuffer.load();

std::vector<std::string> fields;
std::stringstream ss(nmea_response);
std::string field;

while (std::getline(ss, field, ',')) {
    fields.push_back(field);
}

std::string slat = fields[2];
std::string slon = fields[4];
std::string posMode = fields[6];
std::string salt = fields[9];

char cp_lat = fields[3].at(0);
char cp_lon = fields[5].at(0);
[...]
```

Figure B.4: Code snipped of the GNS sentence parsing method

RMC - Recommended Minimum Specific GNSS Data

The RMC sentence provides a minimum set of recommended navigation information from GNSS devices. It includes essential data such as time, position, ground speed, course, and date. It is one of the most commonly used NMEA sentences for basic navigation and positioning.

An example of a standard RMC sentence:

`$GPRMC,123519,A,4807.038,N,01131.000,E,022.4,084.4,230394,003.1,W*6A`

The fields represent:

- **123519**: Fix time at 12:35:19 UTC
- **A**: Status, 'A' = Active, 'V' = Void (invalid)
- **4807.038, N**: Latitude 48 deg 07.038' North
- **01131.000, E**: Longitude 11 deg 31.000' East
- **022.4**: Speed over ground in knots
- **084.4**: Course over ground (degrees True)
- **230394**: Date (23rd of March 1994)
- **003.1, W**: Magnetic variation (West)

The `parseNmeaRmc()` function interprets the Recommended Minimum Specific GNSS Data (RMC) sentence by extracting and processing key navigation information, such as speed over ground (SOG), course over ground (COG), and GNSS fix status. Following the methodology adopted in the parsing of GNS sentences, it begins by dividing the incoming NMEA sentence into individual comma-separated fields. Subsequently, the function retrieves the relevant fields representing the fix mode, fix validity, speed, and heading

values. The SOG and COG values are parsed and converted from their original string representation into numeric values, explicitly converting speed from knots to meters per second. Fix validity, indicated by a separate character, is combined with the fix mode to comprehensively assess the overall fix status (e.g., Autonomous GNSS Fix, RTK Fixed, or No Fix), including explicit handling for invalid data conditions.

Furthermore, the function stores the parsed fix status along with the precise timestamp of parsing, facilitating synchronization and validation with other GNSS sentence parsers, particularly for integration with the GNS parsing strategy.

Finally, a timestamp in a human-readable format is produced, along with precise timing data for diagnostic purposes, internal validity flags are updated accordingly, and the processed data is atomically stored into a shared output buffer to be utilized by other system components.

```
void UBXNMEAParserSingleThread::parseNmeaRmc(std::string nmea_response)
   {
    out_t out_nmea = m_outBuffer.load();

    std::vector<std::string> fields;
    std::stringstream ss(nmea_response);
    std::string field;

    while (std::getline(ss, field, ',')) {
        fields.push_back(field);
    }

    char fix = fields[12].at(0);
    char fix_validity = fields[2].at(0);
    std::string sog = fields[7];
    std::string cog = fields[8];

    if (!cog.empty()) {
        if (cog[0] == '-') {
            cog = cog.erase(0,1);
            out_nmea.cog = std::stod(cog) * -1;
            out_nmea.cog_nmea = out_nmea.cog;
        } else {
            out_nmea.cog = std::stod(cog);
            out_nmea.cog_nmea = out_nmea.cog;
        }
    } else {
        out_nmea.cog = HeadingValue_unavailable_serial_parser;
        out_nmea.cog_nmea = out_nmea.cog;
    }
    [...]

    if (fix == 'N') {
        strcpy(out_nmea.fix_nmea, fix_validity != 'A' ? "NoFix (V)" : "
   NoFix");
        m_2d_valid_fix = m_3d_valid_fix = false;
    } else if (fix == 'E') {
        strcpy(out_nmea.fix_nmea, fix_validity != 'A' ? "Estimated/
   DeadReckoning (V)" : "Estimated/DeadReckoning");
        m_2d_valid_fix = m_3d_valid_fix = true;
    }
    [...]
    else
        strcpy(out_nmea.fix_nmea, fix_validity != 'A' ? "Unknown/Invalid
    (V)" : "Unknown/Invalid");
        m_2d_valid_fix = m_3d_valid_fix = false;
}
```

Figure B.5: Key passages of the RMC sentence parsing method code

Bibliography

[1] ArduSimple. Simplertk2b hookup guide. https://www.ardusimple.com/4g-ntrip-client-hookup-guide/.

[2] DriveX-devs. OScar - Open Stack for Car Framework, 2025. Accessed: 2025-02-01.

[3] DriveX-devs. TRACEN-X, 2025.

[4] ETSI. Intelligent transport systems (its); vehicular communications; basic set of applications; part 2: Specification of cooperative awareness basic service. https://www.etsi.org/deliver/etsi_en/302600_302699/30263702/01.04.01_60/en_30263702v010401p.pdf.

[5] ETSI. Intelligent Transport Systems (ITS); Communications Architecture, 2010.

[6] ETSI. ETSI TS 101 556-1 V1.1.1 (2012-07) Intelligent Transport Systems (ITS); Infrastructure to Vehicle Communication; Electric Vehicle Charging Spot Notification Specification, 2012.

[7] ETSI. Intelligent Transport Systems (ITS); Harmonized Channel Specifications for Intelligent Transport Systems operating in the 5 GHz frequency band, 2012.

[8] ETSI. Intelligent Transport Systems (ITS); OSI cross-layer topics; Part 1: Architecture and addressing schemes, 2012.

[9] ETSI. ETSI EN 302 636-1 V1.2.1 (2014-04) Intelligent Transport Systems (ITS); Vehicular Communications; GeoNetworking; Part 1: Requirements, 2014.

[10] ETSI. Intelligent Transport Systems (ITS); Congestion Control Mechanisms for the C-V2X PC5 interface; Access layer part, 2018.

[11] ETSI. ETSI EN 302 637-2 V1.4.1 (2019-04) Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 2: Specification of Cooperative Awareness Basic Service, 2019.

[12] ETSI. Intelligent Transport Systems (ITS); ITS-G5 Access layer specification for Intelligent Transport Systems operating in the 5 GHz frequency band, 2019.

[13] ETSI. Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 3: Specifications of Decentralized Environmental Notification Basic Service, 2019.

[14] ETSI. Intelligent Transport Systems (ITS); Vehicular Communications; GeoNetworking; Part 5: Transport Protocols; Sub-part 1: Basic Transport Protocol, 2019.

[15] ETSI. ETSI TS 103 300-3 V2.1.1 (2020-11) Intelligent Transport Systems (ITS); Vulnerable Road Users (VRU) awareness; Part 3: Specification of VRU awareness basic service; Release 2, 2020.

[16] ETSI. ETSI TS 103 301 V1.3.1 (2020-02) Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Facilities layer protocols and communication requirements for infrastructure services, 2020.

[17] ETSI. Intelligent Transport Systems (ITS); Security; ITS communications security architecture and security management; Release 2, 2021.

[18] ETSI. Intelligent Transport Systems (ITS); Security; Security header and certificate formats; Release 2, 2021.

[19] ETSI. Intelligent Transport Systems (ITS); Security; Confidentiality services; Release 2, 2022.

[20] ETSI. Intelligent Transport Systems (ITS); Security; Trust and Privacy Management; Release 2, 2022.

[21] ETSI. ETSI TS 103 324 V2.1.1 (2023-06) Intelligent Transport System (ITS); Vehicular Communications; Basic Set of Applications; Collective Perception Service; Release 2, 2023.

[22] Marco Guerrieri. Smart roads geometric design criteria and capacity estimation based on av and cav emerging technologies. a case study in the trans-european transport network - scientific figure on researchgate. Online article or technical report, 2021. https://www.researchgate.net/publication/350976769_Smart_Roads_Geometric_Design_Criteria_and_Capacity_Estimation_Based_on_AV_and_CAV_Emerging_Technologies_A_Case_Study_in_the_Trans-European_Transport_Network.

[23] NMEA. NMEA website, 2025.

[24] Mohammed Obaid and Zsolt Szalay. A novel model representation framework for cooperative intelligent transport systems. *Periodica Polytechnica Transportation Engineering*, 48, 05 2019.

[25] Panos Papadimitratos and et al. Secure vehicular communication systems: Design and architecture. IEEE.

[26] pointonenav. Real time kinematics. https://pointonenav.com/news/rtk-survey/.

[27] ublox. Zed-f9r interface description. https://cdn.sparkfun.com/assets/learn_tutorials/1/1/7/2/ZED-F9R_Interfacedescription__UBX-19056845_.pdf.

[28] Princeton University. Global navigation satellite system (gnss) overview. Online article or technical report, 2007. https://www.princeton.edu/~alaink/Orf467F07/GNSS.pdf.

[29] vboxautomotive. Real time kinematics. https://www.vboxautomotive.co.uk/index.php/de/wie-funktioniert-es-rtk.

[30] Lei Wang and Michel Kadoch. Adaptive cam generation for efficient v2x communications. IEEE Internet of Things Journal, Vol. 9, No. 15, 2022, pp. 12450–12462, https://doi.org/10.1109/JIOT.2022.3143220.

[31] yan9a. Serial. https://github.com/yan9a/serial.