

POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica



**SELF-DRIVING CARS AND OPENPILOT: A COMPLETE
OVERVIEW OF THE FRAMEWORK**

Relatore: Prof. Luciano BARESI
Correlatore: Dott. Damian Andrew TAMBURRI

Tesi di Laurea di:
Francesco FONTANA
Matr. 919901

Anno Accademico 2020 / 2021

Ringraziamenti

A conclusione di questo lavoro di tesi, è doveroso porre un sincero ringraziamento a tutti coloro che, in momenti diversi e in vari modi, hanno prestato il loro aiuto e la loro assistenza nella realizzazione di questo lavoro.

Desidero innanzitutto ringraziare il professor Luciano Baresi, relatore di questa tesi, per la disponibilità e la cortesia dimostrata e per il supporto fornito durante la stesura. Un grazie va anche al dottor Damian Tamburri, per il tempo concessomi e per il gran sostegno allo sviluppo delle tesi esposte in questo lavoro.

Un ringraziamento va ai miei amici e colleghi che, in un modo o nell'altro, hanno condiviso con me tutti questi anni di gioie e sacrifici. In particolare, ringrazio Vincenzo, Filippo e Francesco, amici da una vita con i quali ho condiviso tempo, passioni ed esperienze.

Un sincero ringraziamento va poi a Danila, la mia più cara amica, che mi ha sempre supportato nei momenti difficili.

Ringrazio poi tutti i miei zii, punti di appoggio sui quali ho sempre potuto contare e sempre disposti ad aiutarmi in questa mia avventura da fuorisede.

Il più grande ringraziamento va ovviamente alla mia famiglia, a mia madre e mio padre, per l'amore incondizionato che mi hanno sempre dimostrato in qualsiasi situazione, per tutto il sostegno datomi in questi anni e i relativi sforzi e sacrifici, fatti senza mai pensarci due volte, ma soprattutto per non aver mai smesso di credere in me.

Abstract (IT)

Openpilot è un vasto progetto opensource, con più di 230 contributori e 50.000 righe di codice. La documentazione disponibile fornisce alcuni spunti su cosa è in grado di fare Openpilot, come installarlo e come le persone possono contribuire al suo sviluppo, mentre il team di sviluppo pubblica periodicamente aggiornamenti sul blog aziendale [1], informando i lettori sullo stato del progetto e sull'implementazione di nuove caratteristiche.

Queste informazioni, che potrebbero essere sufficienti per un utente che vuole solo utilizzare Openpilot e, di tanto in tanto, ricevere aggiornamenti sulle modifiche che verranno apportate nelle versioni future, potrebbe essere insufficienti per uno sviluppatore che vuole contribuire e non sa da dove cominciare. Sul blog sono presenti informazioni generali su cosa sono i diversi componenti del software e come interagiscono tra loro, ma non è disponibile documentazione tecnica.

La community di Openpilot è molto attiva e spesso proprio dalla community arrivano le risposte alle domande tecniche degli sviluppatori, che tuttavia possono non essere sempre affidabili.

Lo scopo di questo studio è analizzare Openpilot, decodificare il suo codice sorgente per capire come è strutturato il software, come i diversi componenti interagiscono e scambiano informazioni e quali sono gli aspetti teorici alla base di alcuni concetti chiave del software.

L'analisi verterà su:

- Le dipendenze che caratterizzano il progetto e come Openpilot sfrutta le sue librerie.
- Come sono stati testati i diversi componenti del software.
- Come si è evoluto il software.
- Come è stato organizzato il lavoro e come l'open sourcing ha contribuito allo sviluppo.

Lo scopo dell'analisi dei contributi è identificare i contributori chiave e quali funzionalità sono state introdotte nel corso dello sviluppo. Il punto di partenza è la cronologia dei cambiamenti presenti su GitHub per ogni modulo del software, che verrà poi filtrata e raggruppata per identificare i contributi chiave.

Per comprendere meglio come vengono perseguiti gli obiettivi di test di Openpilot, per ogni modulo verranno analizzati i test inclusi e le GitHub Action che vengono eseguite dopo ogni commit, per avere un quadro più chiaro di quali sono gli aspetti critici da monitorare per ogni repository.

La versione del progetto considerata per questa analisi sarà la versione 0.8.7, rilasciata il 01/07/2021.

Parole chiave: open source; guida autonoma; intelligenza artificiale; machine learning; analisi del software; test del software; qualità del software; contributi della community;

Abstract (EN)

Openpilot is a vast open-source project, with more than 230 contributors and 50.000 lines of code. The documentation available gives some insights on what Openpilot is capable of doing, how to install it and how people can contribute to it, while the development team posts periodically update on the company's blog [1] on the state of the project and implementation of new features.

This information, which could be enough for a user that only wants to use Openpilot and, from time to time, receive updates on the changes that will be made in future releases, could be insufficient for a developer that wants to contribute and doesn't know where to start. On the blog are present general information on what the different components of the software are and how they interact, but no technical documentation is available.

The community of Openpilot is very active and often from the community come the answers to the technical questions of the developers, but community members cannot be always reliable and give answers that are the results of their interpretation of the code.

This study aims to analyze the Openpilot, reverse engineering its codebase to understand how the software is structured, how the different components interact and exchange messages, and what are what is the theory behind some of the key aspects of the software.

The analysis will focus on:

- The dependencies that characterize the project and how Openpilot leverages its libraries.
- How the different components of the software have been tested.
- How each component of the software evolved.
- How the work was organized and how open sourcing contributed to the development.

The analysis of the contributions aims to identify the key contributors and what functionalities were introduced during the different releases. The starting point is the history of the Git commits of each package of the repository, which will then be filtered and grouped to identify the key contributions.

To better understand how the Openpilot testing goals are pursued, for each directory will be analyzed the tests that are included and the GitHub Actions that are executed after each commit, to have a clearer picture of what are the most critical aspects to monitor for each repository.

The project's version considered for this analysis will be version 0.8.7, released on 01/07/2021.

Keywords: open source; self-driving; artificial intelligence; machine learning; software analysis; software testing; software quality; community contributions;

Summary

1	Introduction	1
2	The Openpilot framework	2
2.1	What is Openpilot?	2
2.2	Structure of Openpilot and management of the workflow.....	3
2.3	Solution's architecture description	6
2.4	Research design	9
3	Openpilot's submodules: their purpose, the development process, and the community contributions	10
3.1	Cereal	10
3.1.1	Package structure	10
3.1.2	Implementation	11
3.1.2.1	Messaging.....	13
3.1.2.2	VisionIPC	16
3.1.3	Usage	18
3.1.3.1	Messaging - usage	19
3.1.3.2	VisionIPC - usage.....	21
3.1.4	Testing	23
3.1.5	Development and community contribution	25
3.2	Common.....	32
3.2.1	Package structure.....	32
3.2.2	Implementation	33
3.2.2.1	API	34
3.2.2.2	Kalman	35
3.2.2.3	Transformation	37
3.2.3	Usage	39
3.2.3.1	Api - usage	40
3.2.3.2	Kalman - usage.....	41

3.2.3.3	Transformations - usage	42
3.2.4	Testing	44
3.2.5	Development and community contribution	46
3.3	Laika	55
3.3.1	Package structure.....	57
3.3.2	Implementation.....	58
3.3.3	Usage	62
3.3.4	Testing	67
3.3.5	Development and community contribution	70
3.4	OpenDBC.....	77
3.4.1	Package structure.....	77
3.4.2	Implementation.....	77
3.4.3	Usage	88
3.4.4	Testing	91
3.4.5	Development and community contributions.....	92
3.5	Panda.....	100
3.5.1	Package structure.....	102
3.5.2	Implementation.....	102
3.5.3	Usage	111
3.5.4	Testing	113
3.5.5	Development and community contribution	123
3.6	Rednose.....	129
3.6.1	Package structure.....	129
3.6.2	Implementation.....	130
3.6.3	Usage	134
3.6.4	Testing	136
3.6.5	Development and community contributions.....	138
4	Self-driving cars: an overview of the Openpilot framework and its quality assessment	
	139	
4.1	Package structure	139
4.2	Implementation	143
4.2.1	AthenaD.....	143

4.2.2	BoardD/PandaD.....	145
4.2.3	CameraD	147
4.2.4	ControlsD	150
4.2.5	PlannerD	155
4.2.6	RadarD.....	159
4.2.7	CalibrationD	161
4.2.8	LocationD	164
4.2.9	UbloxD	166
4.2.10	ModelD.....	168
4.2.11	DMonitoringModelD	175
4.2.12	DMonitoringD	178
4.2.13	LoggerD.....	180
4.3	Testing	183
4.4	Development and community contributions	197
5	Conclusions and future work	209
6	References	210
7	Appendixes.....	215

List of Figures

Figure 1 - Flowchart diagram of the development process of Openpilot, supported by GitHub CI	3
Figure 2 - Motivation view of this thesis, showing the key stakeholders and the goals that this thesis wants to reach	6
Figure 3 - BPMN diagram showing the process of making new changes to the software, leveraging the artifacts of this thesis	7
Figure 4 - BPMN diagram showing the process of retrieving information related to the technologies used in self-driving cars, looking at the details of the technologies implemented by Openpilot	8
Figure 5 - Packages included in cereal main directory	10
Figure 6 - Messaging class diagram, showing the links between the PubSocket and SubSocket and the integration with the PubMaster and SubMaster components	13
Figure 7 - PubSub design pattern	15
Figure 8 - visionipc class diagram, with the VisionIPC server having a PubSocket for each type of VisionBuf and the VisionIPC client able to subscribe only to a type of VisionBuf at a time (for a maximum of 128 buffers of the same type)	16
Figure 9 - Import dependencies between selfdrive and cereal	18
Figure 10 - Import dependencies between selfdrive and cereal/messaging	19
Figure 11 - Import dependencies between selfdrive and cereal/visionipc.....	21
Figure 12 - Package diagram of the common directory	32
Figure 13 - Import dependencies between selfdrive and common	39
Figure 14 - Import dependencies between selfdrive and common/api.....	40
Figure 15 - Import dependencies between selfdrive and common/kalman	41
Figure 16 - Import dependencies between selfdrive and common/transformations.....	42
Figure 17 - Comparison between the cropped frame and the full-frame acquired by the device's camera	47
Figure 18 - Visualization of GLONASS and GPS orbits	56
Figure 19 - Receiver on Earth and the estimation of the position in space of multiple satellites	56
Figure 20 - Predictions of the Carrier Phase (CP) made by AstroDog.....	57

Figure 21 - Package diagram of Laika.....	57
Figure 22 - Import dependencies between selfdrive and laika	62
Figure 23 - get_ionex() activity diagram showing download, elaboration, and caching of IONEX files	64
Figure 24 - get_nav() activity diagram showing download, elaboration, and caching of NAV files	65
Figure 25 - get_orbit() activity diagram showing download, elaboration, and caching of ORBIT files	66
Figure 26 - Comparison of the measurements acquired with Laika and U-Blox. Laika shows a much higher concentration of data around 0 meters of altitude error.....	70
Figure 27 - Opendbc package diagram.....	77
Figure 28 - DBC class diagram. Each part of a DBC file has its implementation (Msg, Signal, Val).	79
Figure 29 - CANPacker class diagram. The component construct a message starting from a series of values.	81
Figure 30 - CANParser class diagram. The component parses the received message through the MessageState component.....	84
Figure 31 - Import dependencies between selfdrive and opendbc	88
Figure 32 - Comparison between the precision of ZSS and that of the standard sensor	97
Figure 33 - The White Panda.....	100
Figure 34 - The Grey Panda with its external GPS antenna.....	100
Figure 35 - The Black Panda	101
Figure 36 - The Red Panda.....	101
Figure 37 - Panda package diagram	102
Figure 38 - Panda class diagram. The handle attribute can be a either USB handle (standard from usllib) or a custom Wi-Fi handle, according to the connection method.	108
Figure 39 - Import dependencies between selfdrive and panda	111
Figure 40 - Rednose package diagram	129
Figure 41 - EKFSym class diagram. EKFSym leverages the Extended Kalman Filter EKF that is generated according to the selected measurement model and provides the prediction and update methods.	130
Figure 42 - Import dependencies between selfdrive and rednose	134
Figure 43 - Kinamatic EKF simulation plot	137
Figure 44 - Openpilot package diagram	140
Figure 45 - Dependencies of the selfdrive packages	140
Figure 46 - Openpilot deployment diagram	141

Figure 47 - processes taxonomy, organized by type and objective of the process	142
Figure 48 - Sequence diagram of the initialization of Openpilot. ManagerD daemon takes care of managing the choreography of processes that run during the execution of the software.	143
Figure 49 - Panda and Pigeon class diagram	146
Figure 50 - camerad class diagram. The two cameras only differ for the way they handle the processor instructions calls.....	147
Figure 51 - camerad activity diagram.....	148
Figure 52 - camerad data flow diagram.....	149
Figure 53 - controlsd inbound data flow diagram	151
Figure 54 - controlsd activity diagram	152
Figure 55 - CarInterface class diagram. The CarInterface leverages the CANParser component to elaborate the received messages.	153
Figure 56 - controlsd outbound data flow diagram	154
Figure 57 - lane change statechart diagram	155
Figure 58 - LateralPlanner class diagram	156
Figure 59 - LongitudinalPlanner class diagram.....	157
Figure 60 - plannerd data flow diagram	158
Figure 61 - RadarD class diagram	159
Figure 62 - RadarInterface class diagram.....	159
Figure 63 - RadarData message packet specifications	159
Figure 64 - radard data flow diagram	160
Figure 65 - calibrationd data flow diagram	163
Figure 66 - locationd class diagram. The EKFSym component from Rednose library helps smoothing the acquired data.	164
Figure 67 - locationd data flow diagram	165
Figure 68 - UbloxMsgParser class.....	166
Figure 69 - ubloxd data flow diagram	167
Figure 70 - modeld class diagram. RunModel component can use SNPEModel component, that leverages only the Snapdragon processor capabilities, or use the caching system provided by Thneed.	168
Figure 71 - supercombo model input.....	172
Figure 72 - supercombo model output.....	172
Figure 73 - Desire message specifications	173
Figure 74 - modeld data flow diagram	174

Figure 75 - dmonitoringd class diagram. Similarly to the path prediction model, also here there is the possibility for RunModel to use SNPEModel only or Thneed.....	175
Figure 76 - monitoring_model_q input	176
Figure 77 - monitoring_model_q output	176
Figure 78 - dmonitoringmodeld data flow diagram	177
Figure 79 - DriverStatus class diagram	178
Figure 80 - DMonitoringD data flow diagram	179
Figure 81 - LoggerdState class diagram	180
Figure 82 - VideoEncoder class diagram	181
Figure 83 - selfdrive data flow diagram	182
Figure 84 - Speed variation for different targets of cruise speeds.....	185
Figure 85 - Acceleration variation for reaching the target cruise speed.....	185
Figure 86 - Following distance at different lead speeds	186
Figure 87 - Steer ratios of two consecutive commits	194
Figure 88 – Difference of steer ratio in two consecutive commits.....	194
Figure 89 - Messages delays obtained through HIL testing, shown during COMMA_CON	195
Figure 90 - Comparison between a healthy boardd and a lagging one, from COMMA_CON	195
Figure 91 - Results of the study conducted by Consumer Report	196
Figure 92 - The Comma One	197
Figure 93 - EON dashcam DevKit	198
Figure 94 - The Comma Two device.....	199
Figure 95 - Diagram of the warp-simulator.....	200
Figure 96 - The Comma Three device.....	201
Figure 97 - output of the analysis of the contribution of Adeeb Shihadeh.....	205
Figure 98 - Openpilot's software entropy	206
Figure 99 - Total contribution made by the most active users	208

List of Tables

Table 1 - Lines of code of cereal, by programming language	11
Table 2 - TestPubSubSockets test case.....	23
Table 3 - TestMessaging test case	23
Table 4 - TestPoller test case.....	23
Table 5 - TestSubMaster test case	24
Table 6 - TestPubMaster test case	24
Table 7 - TestServices test case.....	24
Table 8 - Lines of code for the Common, by programming language	32
Table 9 - TestFileHelpers unit test	44
Table 10 - InterpTest unit test	44
Table 11 - TestParams test case.....	44
Table 12 - TestParams (xattr) test case.....	45
Table 13 - TestSimpleKalman test case	45
Table 14 - TestNED test case	45
Table 15 - TestOrientation test case	45
Table 16 - Lines of code of the Laika, by programming language	57
Table 17 - AstroDog class	58
Table 18 - TestDOP Test Case	67
Table 19 - TestAstroDog Test Case	67
Table 20 - TestFailCache Test Case	67
Table 21 - TestFetchSatInfo Test Case.....	68
Table 22 - TestPositioning Test Case	68
Table 23 - TestConstellationPRN Test Case	68
Table 24 - TestTime Test Case.....	69
Table 25 - TestTimeRangeHolder Test Case	69
Table 26 - Packages included in opendbc main directory	77
Table 27 - TestCanParserPackerExceptions test case	91

Table 28 - TestCANDefine test case	91
Table 29 - TestCanParserPacker test case	91
Table 30 - Packages included in Panda main directory	102
Table 31 - Common Test Cases.....	114
Table 32 - TestChryslerSafety	115
Table 33 - TestGmSafety Test Case	115
Table 34 - Honda Test Cases.....	116
Table 35 - Hyundai Test Cases.....	117
Table 36 - TestMazdaSafety Test Case	117
Table 37 - TestNissanSafety Test Case	118
Table 38 - TestSubaruLegacySafety Test Case	118
Table 39 - TestSubaruSafety Test Case.....	119
Table 40 - TestTeslaSafety Test Case	119
Table 41 - TestToyotaSatology Test Case	120
Table 42 - TestVolkswagenMqbSafety Test Case	120
Table 43 - TestVolkswagenPqSafety Test Case.....	121
Table 44 - Lines of code of Rednose, by programming language.....	129
Table 45 - TestCompare test case.....	136
Table 46 - TestKinematic Test Case.....	137
Table 47 - Lines of code of selfdrive, by programming language	139
Table 48 - TestValgrind test case	183
Table 49 – TestBoarddApiMethods	184
Table 50 - TestAlerts test case.....	184
Table 51 - TestClustering test case.....	184
Table 52 - TestCruiseSpeed test case	184
Table 53 - TestFollowingDistance test case	186
Table 54 - TestLateralMpc test case.....	186
Table 55 - TestStartup test case.....	187
Table 56 - TestMonitoring test case	187
Table 57 - TestUploader test case.....	188
Table 58 - TestDeleter test case.....	189
Table 59 - TestEncoder test case	189

Table 60 - TestLoggerd test case.....	190
Table 61 - TestCalibrationd test case	190
Table 62 - TestLocationdLib test case.....	190
Table 63 - TestLocationdProc test case.....	191
Table 64 - TestAthenadMethods test case	191
Table 65 - TestRegistration test case	192
Table 66 - TestPowerMonitoring test case.....	192
Table 67 - TestCarModel test case	193
Table 68 - Statistics of the main contributors.....	207
Table 69 - Averages and standard deviations of the selected statistics	207
Table 70 - Normalized statistics for each contributor	208
Table 71 - Discord server's channels description	219
Table 72 - Commits adding new parameters to capnp files in cereal.....	226
Table 73 - Contributions to the DBC files of Acura cars	226
Table 74 - Contributions to the DBC files of BMW cars	226
Table 75 - Contributions to the DBC files of Chrysler cars	227
Table 76 - Contributions to the DBC files of Ford cars	228
Table 77 - Contributions to the DBC files of GM cars	229
Table 78 - Contributions to the DBC files of Chevrolet cars	229
Table 79 - Contributions to the DBC files of Cadillac cars	229
Table 80 - Contributions to the DBC files of Honda cars	232
Table 81 - Contributions to the DBC files of Hyundai cars	233
Table 82 - Contributions to the DBC files of Luxgen cars.....	233
Table 83 - Contributions to the DBC files of Mazda cars	234
Table 84 - Contributions to the DBC files of Mercedes cars	234
Table 85 - Contributions to the DBC files of Nissan cars	234
Table 86 - Contributions to the DBC files of Subaru cars.....	236
Table 87 - Contributions to the DBC files of Tesla cars	236
Table 88 - Contributions to the DBC files of Toyota cars.....	240
Table 89 - Contributions to the DBC files of Lexus cars	240
Table 90 - Contributions to the DBC files of Volvo cars	241
Table 91 - Contributions to the DBC files of Volkswagen car	241

1 Introduction

The rapid development of the Internet economy and Artificial Intelligence (AI) has promoted the progress of self-driving cars. The offer of many car manufacturers, such as Tesla, focuses on delivering cars capable of autonomously deciding what is the safest path to follow and detecting possible danger coming from the road using a wide range of sensors.

Unlike what many would think, self-driving a car is not a hardware problem, instead is almost completely related to software. Modern cars are built in such a way that all the relevant information about the status of the car is exchanged over standardized channels, that if accessed allow controlling the car. Many cars that support, to some extent, automated drive use a set of sensors that can be reduced to cameras and radars, which allows the car to be aware of its surroundings. These data are then elaborated and through AI and machine learning.

Machine learning has two learning models: supervised and unsupervised. With unsupervised learning, a machine learning algorithm receives unlabeled data and no instructions on how to process it, so it has to figure out what to do on its own.

With the supervised model, an algorithm is fed instructions on how to interpret the input data. This is the preferred approach to learning for self-driving cars. It allows the algorithm to evaluate training data based on a fully labeled dataset, making supervised learning more useful where classification is concerned.

These machine learning algorithms, in the end, allow a car to collect data on its surroundings from cameras and other sensors, interpret it, and decide what actions to take. Machine learning even allows cars to learn how to perform these tasks as good as (or even better than) humans. This leads to the reasonable conclusion that machine learning algorithms and autonomous vehicles are the future of transportation. [2]

The approach of many car manufacturers is that of developing their machine learning algorithm and providing their cars with a precise set of sensors. However, in a trial to democratize self-driving cars and make them available to everyone, Comma.ai with Openpilot, the object of this analysis, offers a single device, able to bring the power of a machine learning algorithm trained on thousands of hours of drive in any compatible car. In the following pages, it will be analyzed how this is made possible and what are the limitation that comes with an open-source software, trying to provide a complete overview of all of its components.

2 The Openpilot framework

2.1 What is Openpilot?

Comma.ai is an AI startup founded by **George Hotz (@geohot)** in September 2015. The mission of the company is to “*solve self-driving cars while delivering shippable intermediaries*”.

To achieve this mission, in 2016 Comma.ai launched **Openpilot**, an open-source, semi-automated driving system. It is a comprehensive system of driver assistance features supporting a wide range of car models. Today, Comma.ai sold more than 7.000 devices and has more than 3500 active users.

The Society of Automotive Engineers (SAE) defines 6 levels of driving automation ranging from 0 (fully manual) to 5 (fully autonomous). These levels have been adopted by the U.S. Department of Transportation.

Openpilot allows reaching a *level 2* driving automation level, meaning that the vehicle on which the device is installed can control both steering and accelerating/decelerating. At a level 2 automation, the human still monitors all the tasks and can take control at any time.

It is substantially different from other automated driving systems since it can be installed by anyone that buys the **Comma Two** or **Comma Three** development kit available in their online shop and flashes the Openpilot software available on the public GitHub repository.

Openpilot offers many functionalities based on machine learning and computer vision, including:

- Automated lane-centering
- Adaptive cruise control
- Driver monitoring
- Assisted lane change

The user base grew from less than 275 users per week in the second half of 2018 to more than 2.750 users per week at the end of 2020, and so did the number of developers who contributed to the project, now more than 200.

In 2018, Comma.ai also published their proprietary development tools on Openpilot’s main repository, to help all developers willing to contribute to developing and testing the software with the same tools used internally by Comma.ai.

2.2 Structure of Openpilot and management of the workflow

The Openpilot repository is available at <https://github.com/commaai/Openpilot.git>, it counts more than 5.000 forks and is starred by more than 25.000 users. Developers that want to contribute to the project push their changes on the **master** branch, which is stripped and minified by CI (GitLab Continuous Integration) and pushed to **master-ci** automatically if the tests pass. When the version on the master branch is ready to be published on the release branch, master-ci is pushed to **devel-staging**, opening a pull request into **devel**. This pull request will be the spot for comments on the new release, and hotfixes at this point will be cherry-picked from master into devel-staging. **devel** is built by CI and pushes the built versions to **release-staging** and **dashcam-staging** signed with the official comma release key. After the -staging branches are tested by the community for a few days, the changes are pushed to **release** and **dashcam**. [3]

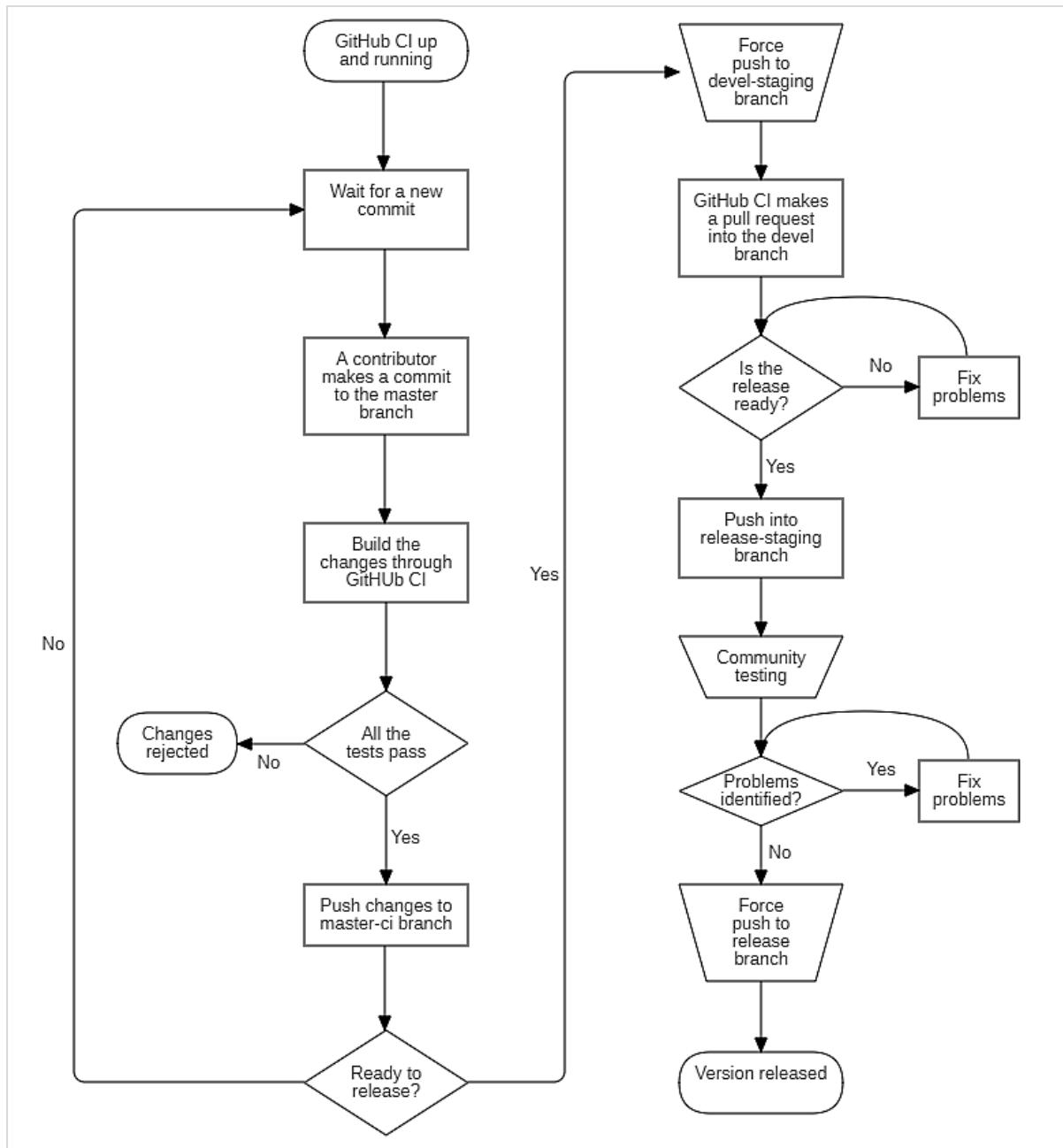


Figure 1 - Flowchart diagram of the development process of Openpilot, supported by GitHub CI

The release branch is structured as follows:

- **cereal**: the messaging spec and libs used for all logs
- **common**: library like functionality and utility functions
- **installer/updater**: manages updates of NEOS
- **opendbc**: files showing how to interpret data from cars
- **panda**: code used to communicate on CAN
- **phonelibs**: external libraries
- **pyextra**: extra python packages not shipped in NEOS
- **selfdrive**: code needed to drive the car
 - **assets**: fonts, images, and sounds for UI
 - **athena**: allows communication with the app
 - **board**: daemon to talk to the board
 - **camerad**: driver to capture images from the camera sensors
 - **car**: car specific code to read states and control actuators
 - **common**: shared C/C++ code for the daemons
 - **controls**: planning and controls
 - **debug**: tools to help you debug and do car ports
 - **location**: precise localization and vehicle parameter estimation
 - **logcatd**: Android logcat as a service
 - **loggerd**: logger and uploader of car data
 - **modeld**: driving and monitoring model runners
 - **proclogd**: Logs information from processes
 - **sensord**: IMU interface code
 - **test**: unit tests, system tests, and a car simulator
 - **ui**: the UI

Openpilot, from version 0.7 on, includes different submodules available on dedicated repositories that allow a completely open style of development. Modules that are in separate repositories include:

- cereal
- laika
- opendbc
- panda
- rednose

This allows developers to work on the same workflow as the Openpilot team and to allow to merge pull requests much easier [3]. Another important aspect to consider is the managerial aspect: Openpilot is entirely managed through **GitHub discussions** and a **Discord server**. The GitHub discussions, as well as the **Openpilot wiki**, gather some general information about the most important aspects of the project. Discussed topics regard porting new cars, discussion about simulations on different hardware, and porting Openpilot on unofficial hardware, but also include FAQs from new users that want to collaborate and developers that want to share their thoughts and doubts.

The Discord server, **Comma.ai community**, is organized into different *channels*, that have the purpose of giving the users a place to discuss different topics. Channels are organized into *categories*, which allow giving access to certain channels only to people that have a specific *role*. A role is a defined set of permissions, identified by a name. Each role has different permissions, that will allow the users having that role to perform certain actions rather than others.

The roles on the server include:

- **admin**: the user is an administrator of the server.
- **Comma.ai staff**: the user is a member of the Comma.ai staff.
- **COMMA_CON**: a user participating to the COMMA_CON, the official convention of Comma.ai where the participants will be able to meet the members of the company and talk about the future of self-driving cars.
- **comma prime**: a user that purchased the comma prime service, the comma's mobile service which allows EON and comma two to connect to the internet through a SIM card and upload the route data in real-time.
- **moderator**: the user is a moderator of the Discord server and has the privileges needed to mute or ban other users.
- **Openpilot contributor**: a user who contributed to the development of Openpilot, by pushing some changes to a branch of the repository.
- **Openpilot tester**: an official tester of Openpilot. Is a user who purchased Openpilot and contributed to making it run on both supported and unsupported car models.
- **community member**: identifies a user that has joined the community. When a user joins the server, this role is given automatically.
- **dev**: a user that has accepted the terms and conditions of the development channel. After accepting, this role will be assigned to the users, allowing them to write, and read messages on the development channels.

The channels on the server are classified into different categories, each of which can be accessed only by users with the right set of privileges. More details on the channels can be found in *Table 71*.

2.3 Solution's architecture description

The aim of this thesis is to support all the contributors in making effective changes to the project, but also give more technical information about the software to anyone that is interested in having more details on the implementation of Openpilot.

The analysis of both the submodules in chapter 3 and selfdrive, where the Openpilot software lives, in chapter 4, aim to reach four key goals:

- Analyze the software implementation, identifying the main classes and functionalities provided by each component. These details are available in the paragraphs **Package structure** and **Implementation** both for each submodule and for selfdrive.
- Analyze the submodules' dependencies, and in particular what are Openpilot's key processes that use the submodules. These details can be found in the paragraph **Usage** of each submodule analyzed in chapter 3.
- In the paragraph **Testing** is an assessment of the testing strategy of each component, what are the tests executed and what is their output.
- In the paragraph **Development and community contribution** An overview of all the main contributions to the project, both internal and external to Comma.ai

The main stakeholders to which this thesis is directed are the community contributors, the users of Openpilot that own a Comma device, and the researchers of the self-driving cars field, while the drivers of this research are three: supporting the contributors with a complete software walkthrough that can give insights on the actual functioning of each component; discuss the main technologies adopted, the theoretical basis and the design choices that are behind the implementation; understand what is the development process through which Openpilot went through and how evolved over time.

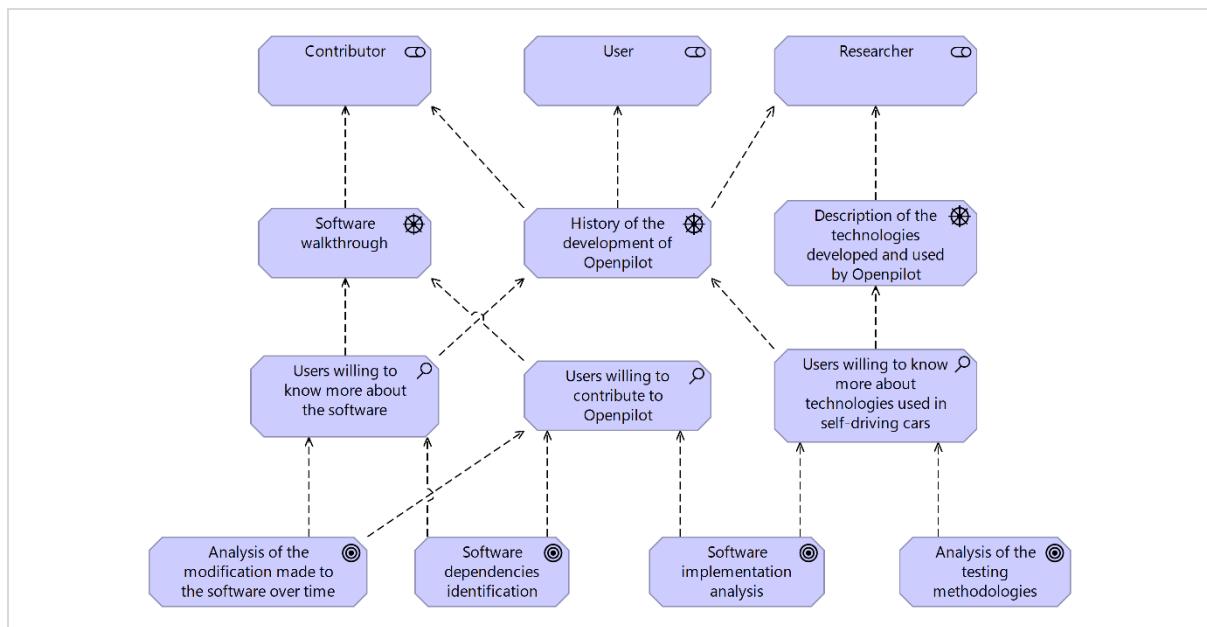


Figure 2 - Motivation view of this thesis, showing the key stakeholders and the goals that this thesis wants to reach

A stakeholder wanting to consult this thesis could adopt different approaches according to what are the specific needs. To have a general view of how Openpilot works, the chapter 4 provides all the information about the processes that are involved in the execution of the software.

The process taxonomy of the software [Figure 47] provides a general classification of all the processes, while each paragraph of the chapter describes them in detail, providing details on the operations that they perform and what other components of the software are involved. To have more insight on the submodules that the processes they use, the stakeholders can find the explanation of their implementation in chapter 3.

A developer can also find relevant information in the same chapters, including technical descriptions of the different components that will support the development of the software features. In Figure 3 is shown how a stakeholder could leverage the information contained in the different paragraphs to improve its productivity and quality of changes made.

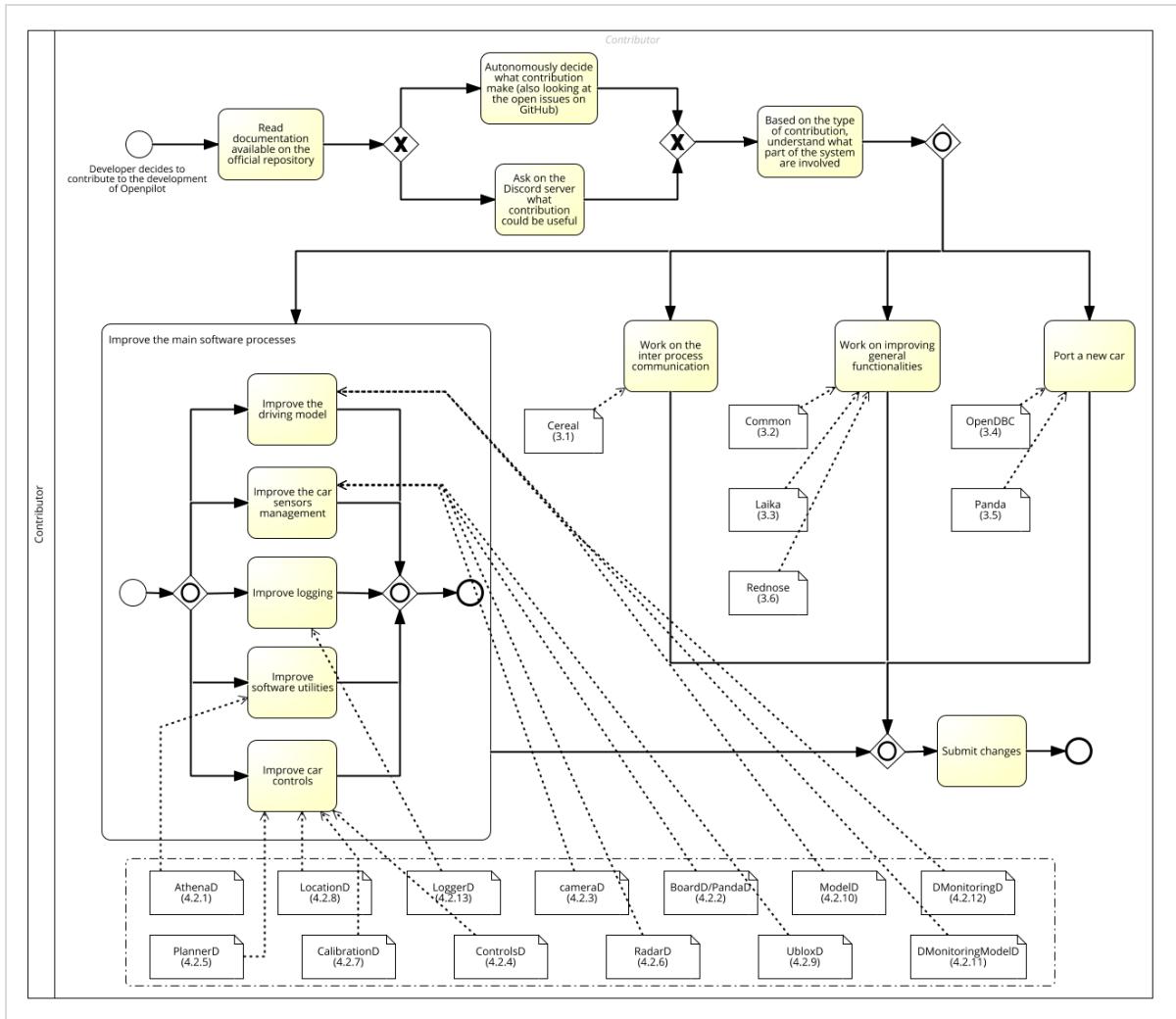


Figure 3 - BPMN diagram showing the process of making new changes to the software, leveraging the artifacts of this thesis

If a stakeholder is instead interested to the development process of Openpilot, each section of chapter 3 and 4 has a paragraph dedicated to the improvement, corrections, and additions made over the years. All the changes are listed in chronological order and come with the list of commits that influenced a specific change. A general analysis of all the contributions and contributors is also available in paragraph 4.4, where are analyzed in a qualitative and quantitative way the contributions made by the most productive contributors, including both the members of the community and Comma.ai employees.

A stakeholder could be interested in this document to have more insights on the technologies that are currently used in the field of self-driving cars and what makes possible to convert the external input coming from outside in data that can then be elaborated and used to predict the path that the car should follow. The paragraph dealing with the implementation of the different components also give insight on the theoretical basis that are behind a certain design or implementation choice. The most relevant technologies that a researcher of the field may be interested into may be the predictive model of Openpilot and the interface that the device uses to communicate with the car.

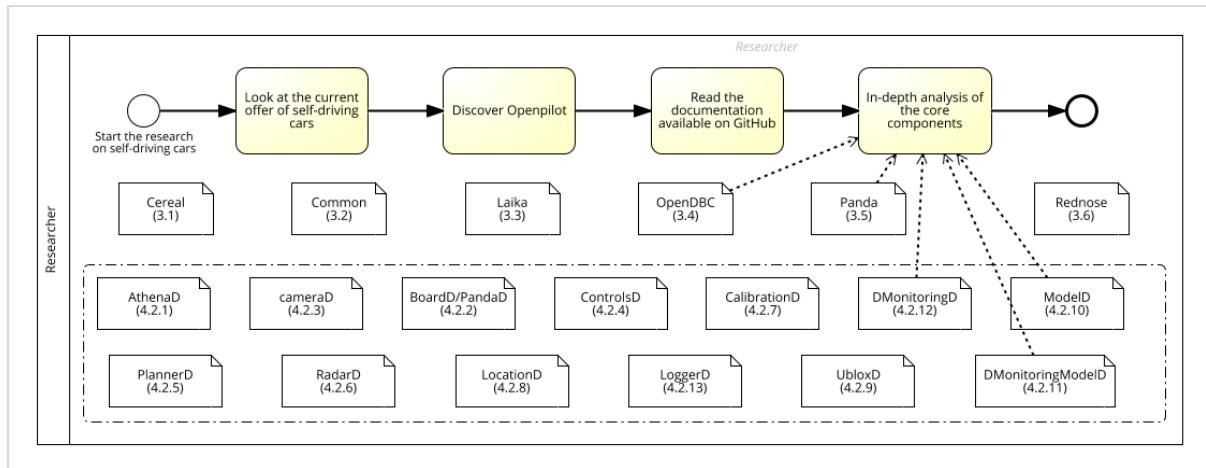


Figure 4 - BPMN diagram showing the process of retrieving information related to the technologies used in self-driving cars, looking at the details of the technologies implemented by Openpilot

2.4 Research design

This thesis will tackle the problem of the lack of a unified source of knowledge for what concerns the more technical details of the Openpilot framework, that for an open-source project of this size could represent an entry barrier for many developers that are willing to contribute but don't have the necessary information to correctly interpret and understand the many modules and processes of which the project is composed.

The available literature that partially mitigates the problem is limited to an article analyzing only the driving model of Openpilot [4] and other articles, posted by the Comma.ai team, giving general information on the different processes. Other general information is present in the main GitHub repository and includes instructions on the installation of the software and on how to contribute, while on the official Discord server the community is very active, and many members often share their knowledge and experiences.

The main source of information of this overview of the Openpilot framework is the source code of Openpilot, available on GitHub, for what concerns the description of the different components, what is their purpose, and how they work and interact with the other components of Openpilot. A lot of information is also found in the Discord server, since the less trivial topics are frequently asked by many users, and in the Pull Requests conversations, which often provided more details than the simple commit description. From the output of the execution of the GitHub Actions, instead, are retrieved the details of the execution of the test, what are the expected results of the tests, and what are the main components that are important to test the reliability of the software.

The analysis of the code base will follow a top-down approach, starting from the classes and methods that expose the functionalities to the other components and decomposing them into their key and atomic functionalities.

This thesis also wants to provide the information needed to assess the quality of the software. Two key points that will be considered will be the compliance of the system with the current regulations and the improvement of the provided functionalities from a release to the successive.

The main tools used to support the analysis are Visual Studio Code, that thanks to its "*Find all references*" functionality make it easy to discover the usage of methods, attributes, and classes throughout the code, even in files written using different programming languages, and different tools used to draw the UML diagrams. The history of the GitHub commits provides useful insights on what changes are made by the users, which functionalities are introduced or corrected, and Visual Studio Code is again a useful tool to examine the changes made, comparing the modified pieces of code in two successive commits.

The structure of this document will see a first chapter, containing the details on the libraries used by Openpilot, how they are structured, how they work, and how they are leveraged by Openpilot, and a second chapter analyzing the source code of Openpilot, focusing on its constellation of processes.

3 Openpilot's submodules: their purpose, the development process, and the community contributions

Openpilot needs many components that allow the software to interface with the car and exchange messages with it. These components, after the open-sourcing of the software, were organized in different repositories, allowing to better manage them and have a clear distinction of what role each component plays. In this excursus of the submodules that are available in the main Openpilot repository will be analyzed the functionalities that each one of them provides, how they were tested to ensure the required levels of reliability, and what was the development process they went through.

3.1 Cereal

Cereal is both a messaging specification for robotics systems as well as a generic high-performance inter-process communication protocol enabling the communication among a single Publisher and multiple Subscribers (IPC pub/sub) for all the components that implement it. One of its main purposes is to enable easy and effective logging of all the events that occur during the usage of *Openpilot*, as well as enabling the different modules of the software to communicate with each other. The main components that *cereal* provides are *messaging*, which is the actual messaging specification library, and *VisionIPC*, which allows exchanging visual data.

3.1.1 Package structure

Cereal is composed of different packages [Figure 5], the main of which is *messaging*. Here are located the methods needed to instantiate the Publisher and the different Subscribers, as well as the methods needed to send and receive messages.

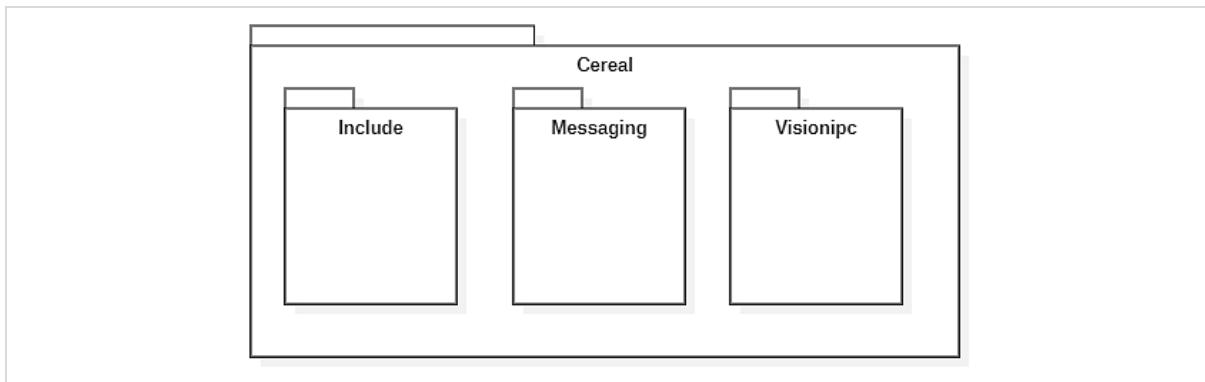


Figure 5 - Packages included in cereal main directory

Both *messaging* and *visionipc* are written in C++, but the package also uses Cython to allow using the library's functionalities in Python code.

Language	Files	Blank	Comment	Code
C++	14	441	79	1.550
C/C++ Header	9	77	4	422
Python	3	65	13	265
Cython	2	55	9	139
SUM:	28	638	105	2.376

Table 1 - Lines of code of cereal, by programming language

3.1.2 Implementation

Cereal supports a wide range of events and states, all defined through Cap'n Proto, a data exchange format optimized for client-server communication. Through the Cap'n Proto specifications can be defined the structs corresponding to different classes of events and messages, and for each of them the relative typed parameter. In particular, these specifications are described in the files *log.capnp*, *legacy.capnp*, and *car.capnp*.

car.capnp contains the specifications for messages that the car needs to send and receive to communicate with Openpilot, such as information on the speed of the car, the transmission, the steering wheel, the gears, the wheels' speed, and so on. There are five main structs defined in *car.capnp*.

- CarEvent, containing the messaging specifications of the events corresponding to a state change of the car. An example of events that are defined in the specifications are:

```
canError @0;
steerUnavailable @1;
brakeUnavailable @2;
wrongGear @4;
```

- CarState, that is the specifications of the messages communicating the state of the car and its components. For instance, the wheels' speed can be encapsulated in a message generated using the following struct.

```
struct WheelSpeeds {
    f1 @0 :Float32;    # front left
    fr @1 :Float32;   # front right
    rl @2 :Float32;   # rear left
    rr @3 :Float32;   # front right
}
```

- RadarData, specifying the structure of the messages related to the radar state and activity. A message transmitting the position retrieved by the radar will be generated by using the following specification:

```
struct RadarPoint {
    trackId @0 :UInt64;      # no trackId reuse
    dRel @1 :Float32;        # m from the front bumper of the car
    yRel @2 :Float32;        # m
    vRel @3 :Float32;        # m/s
    aRel @4 :Float32;        # m/s^2
    yvRel @5 :Float32;       # m/s
    measured @6 :Bool;
```

- CarControl, which contains the messaging specifications corresponding to the different actions on the actuators of the car, such as pedals and steering wheel. The basic struct of this class of messages is that of *Actuators*, which allows communicating the basic car actuator's status.

```
struct Actuators {
    gas @0: Float32;                      # range from 0.0 - 1.0
    brake @1: Float32;                     # range from 0.0 - 1.0
    steer @2: Float32;                     # range from -1.0 - 1.0
    steeringAngleDeg @3: Float32;          # range from -1.0 - 1.0
}
```

- CarParams, that describes the parameters used by the car to calibrate the actuators its actuators while performing an action. One of the most important sets of parameters are the *tuning* parameters, which include lateral and longitudinal tuning messages specification.

```
struct LateralINDITuning {
    outerLoopGainBP @4 :List(Float32);
    outerLoopGainV @5 :List(Float32);
    innerLoopGainBP @6 :List(Float32);
    innerLoopGainV @7 :List(Float32);
    timeConstantBP @8 :List(Float32);
    timeConstantV @9 :List(Float32);
    actuatorEffectivenessBP @10 :List(Float32);
    actuatorEffectivenessV @11 :List(Float32);
}
```

Tuning a car means adjusting the different parameters defined in these messaging specifications, needed to make the car steer, break, and accelerate as desired [5]. Tuning topics are discussed in a dedicated channel of the Discord server.

log.capnp describes the specifications of all the events that are logged during the execution of Openpilot. The events also include that described in **car.capnp** and **legacy.capnp**. The message structures defined include that for the actuators, the lateral and longitudinal tuning, the predictive model output, and all the other components state and relative events allowed.

legacy.capnp includes the specifications of old messages that are not used anymore or deprecated.

3.1.2.1 Messaging

The functionalities provided with the *messaging* package allow the car to communicate with the device, sending the messages triggered by the different events that can occur while driving.

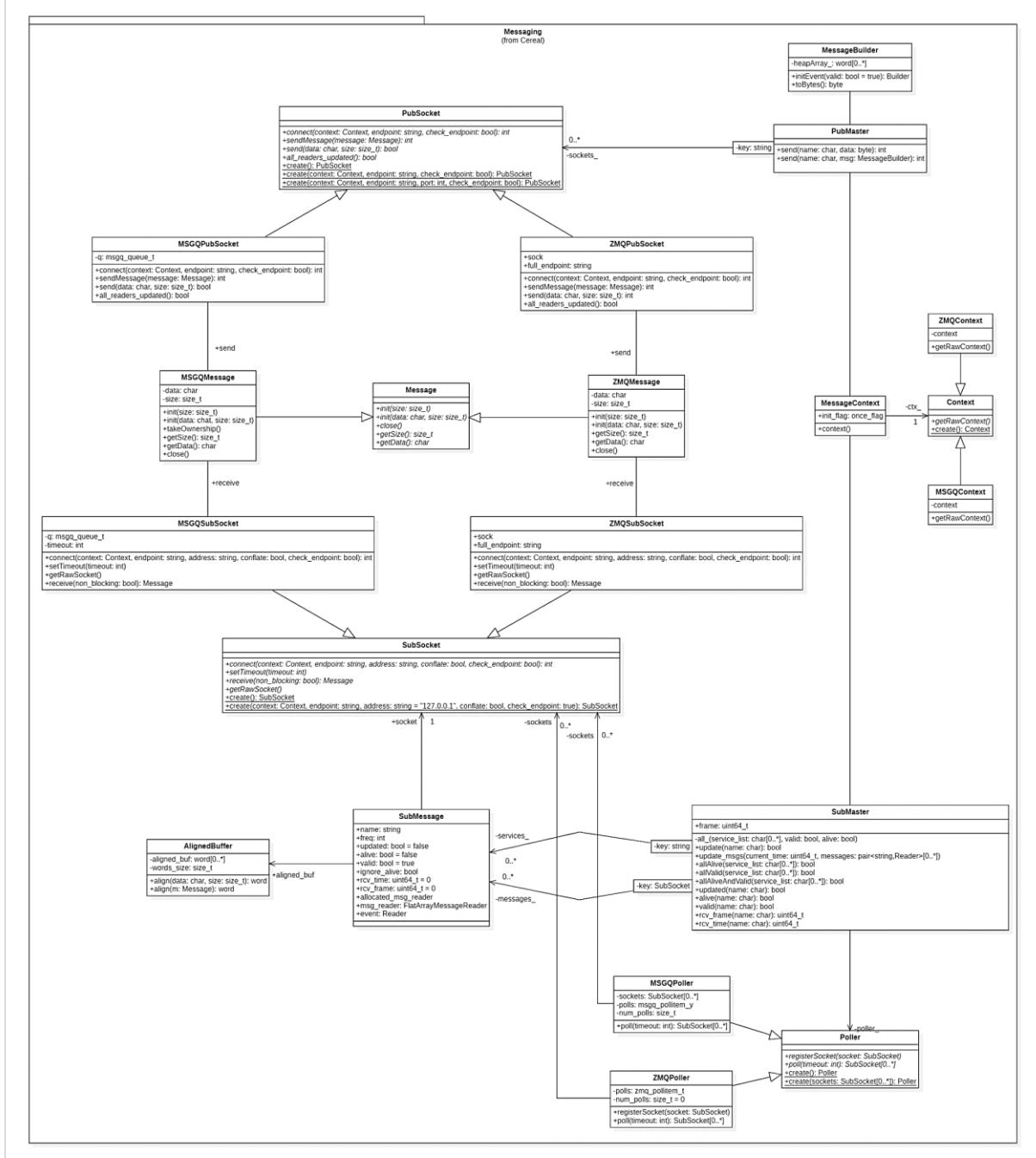


Figure 6 - Messaging class diagram, showing the links between the *PubSocket* and *SubSocket* and the integration with the *PubMaster* and *SubMaster* components

The messages are first created thanks to the method *new_message()* and are initialized accordingly to the different services that require the creation of the message. The *service* parameter will change how the new message is initialized.

```

def new_message(service: Optional[str], size: Optional[int]):
    dat = log.Event.new_message()
    dat.logMonoTime = int(sec_since_boot() * 1e9)
    dat.valid = True
    if service is not None:
        if size is None:
            dat.init(service)
        else:
            dat.init(service, size)
    return dat

```

The methods `pub_sock()` and `sub_sock()` return respectively a PubSocket and a SubSocket object, which correspond to the message publisher and the message receiver, respectively. Multiple subscribers can connect to a socket and receive responses from the publisher, while PubSocket can send the messages to the different subscribers connected to the same its same socket. Each parent class (PubSocket and SubSocket) has two child classes, that are instantiated according to the adopted exchange protocol, ZeroMQ or MSGQ.

ZeroMQ is a high-performance asynchronous messaging library, aimed at use in distributed or concurrent applications. It provides a message queue, but unlike message-oriented middleware, a ZeroMQ system can run without a dedicated message broker. ZeroMQ supports common messaging patterns (pub/sub like the one used in Cereal, request/reply, client/server, and others) over a variety of transports (TCP, in-process, inter-process, multicast, WebSocket, and more), making inter-process messaging as simple as inter-thread messaging.

MSGQ is a system to pass messages from a single producer to multiple consumers. All the consumers need to be able to receive all the messages. It is designed to be a high-performance replacement for ZMQ-like SUB/PUB patterns. It uses a ring buffer in shared memory to efficiently read and write data. Each read requires a copy. Writing can be done without a copy, as long as the size of the data is known in advance.

The library provides the methods to send and receive messages using both ZeroMQ and MSGQ, as shown in the class diagram in Figure 6.

If we consider a typical execution of the program, the Context is first initialized. A Poller is created, according to the adopted messaging protocol. The role of the Poller is that of efficiently wait for new messages, saving resources for more important tasks. When a new message is sent on a socket, the receivers are notified by sending a SIGUSR2 signal.

The SubSocket and the PubSocket are then instantiated, with multiple SubSocket able to connect to the same endpoint and one single PubSocket for each endpoint. The connection is performed through the method `connect()`.

In the case of the PubSocket, it takes as parameters the context, the endpoint, a boolean indicating if the PubSocket should conflate, and a boolean indicating if it should first check the endpoint before sending data. On the other hand, the SubSocket requires the same parameters, with the addition of the address on which the Publisher will send data, which by default is 127.0.0.1.

This is a classical implementation of the Pub/Sub design pattern [Figure 7] [6], in which a client subscribes to a certain topic and the publisher sends messages to all the subscribers subscribed to a certain topic.

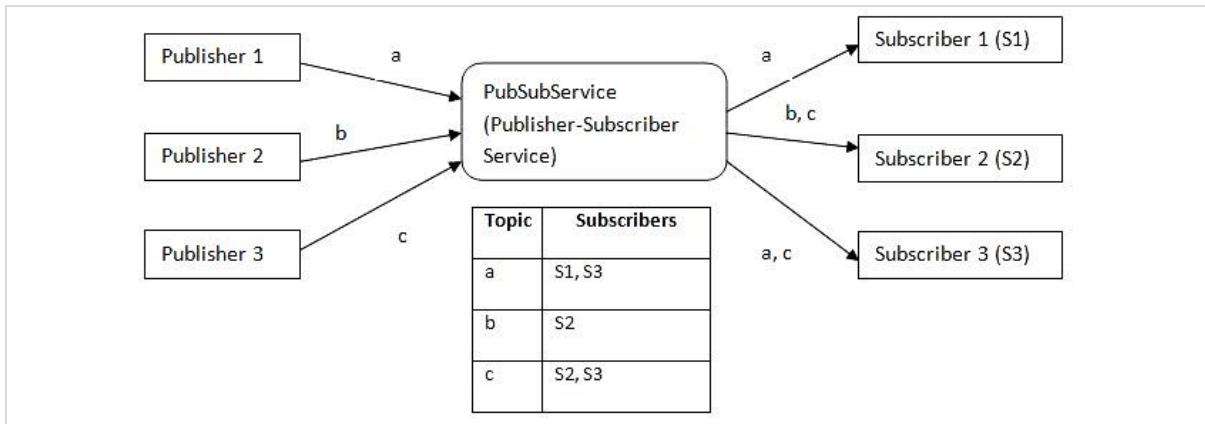


Figure 7 - PubSub design pattern

The exposed classes that can be used to make the Publisher and the Subscribers interact are located in the *socketmaster.cc* file, and there include PubMaster and SubMaster classes. They can be used as shown in the example reported in the official documentation:

```
sm = messaging.SubMaster(['sensorEvents']) # In subscriber
while 1:
    sm.update()
    print(sm['sensorEvents'])

pm = messaging.PubMaster(['sensorEvents']) # In publisher
dat = messaging.new_message('sensorEvents', size=1)
dat.sensorEvents[0] = {"gyro": {"v": [0.1, -0.1, 0.1]}}
pm.send('sensorEvents', dat)
```

The subscriber keeps listening on the selected socket and the *update()* refreshes the messages for all the sockets of the current poll.

```
def update(self, timeout: int = 1000) -> None:
    msgs = []
    for sock in self.poller.poll(timeout):
        msgs.append(recv_one_or_none(sock))

    for s in self.non_polled_services:
        msgs.append(recv_one_or_none(self.sock[s]))
    self.update_msgs(sec_since_boot(), msgs)
```

3.1.2.2 VisionIPC

Inter-process communication (IPC) refers specifically to the mechanisms that allow the processes to manage shared data. In the applications that use IPC, there is typically a client and a server, where the client requests data and the server responds to the client's requests. The module VisionIPC implements this mechanism, providing the methods and classes needed to establish a connection between the client (VisionIpcClient) and the server (VisionIpcServer), and exchange data containing information regarding the acquired images, that can be encoded in RGB or YUV, on the created channel.

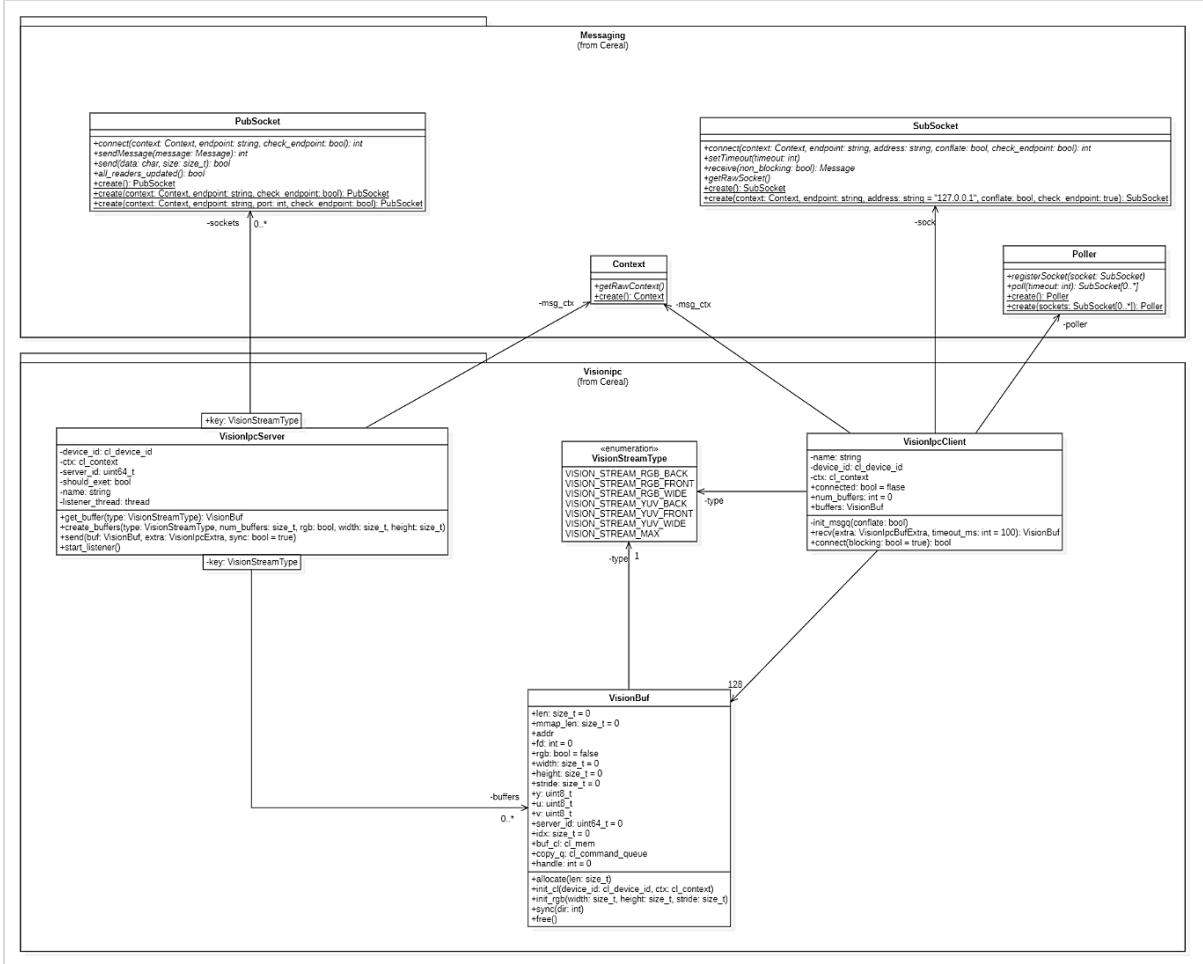


Figure 8 - visionipc class diagram, with the VisionIPC server having a PubSocket for each type of VisionBuf and the VisionIPC client able to subscribe only to a type of VisionBuf at a time (for a maximum of 128 buffers of the same type)

The communication is enabled by the messaging components present in the messaging package: the VisionIPC server has a PubSocket through which publishes the acquired frames, while the VisionIPC client has a SubSocket that connects to the specified PubSocket and receives the frames. The communication between the client and the server happens in the following way:

- The server is first initialized, and a name is given to it.
- The VisionIpcServer then creates a buffer (VisionBuf), specifying the stream type and the dimension of the images (width and height).
- The client has to connect to the server, specifying its name and the type of stream that is going to receive.

- After specifying the content of the VisionBuf and VisionIpcBufExtra, the two objects are then sent to the client

```
VisionIpcServer server("camerad");
server.create_buffers(VISION_STREAM_YUV_BACK, 1, true, 100, 100);
server.start_listener();

VisionIpcClient client = VisionIpcClient("camerad",
VISION_STREAM_YUV_BACK, false);
REQUIRE(client.connect());
zmq_sleep();

VisionBuf * buf = server.get_buffer(VISION_STREAM_YUV_BACK);
REQUIRE(buf != nullptr);

*((uint64_t*)buf->addr) = 1234;

VisionIpcBufExtra extra = {0};
extra.frame_id = 1337;

server.send(buf, &extra);

VisionIpcBufExtra extra_recv = {0};
VisionBuf * recv_buf = client.recv(&extra_recv);
```

On the client side, a Message object is received on the socket to which the client is connected. From the Message object, a VisionIpcPacket is extrapolated and then converted into a VisionBuf that will be equal to the VisionBuf sent by the server.

```
VisionIpcPacket *packet = (VisionIpcPacket*)r->getData();
VisionBuf * buf = &buffers[packet->idx];
```

The VisionBuf represents the main abstraction of the camera frame. The image is encoded using the YUV color encoding, and the VisionBuf provides the details on the three channels Y, U, and V of the image, the width, and height, and provides the methods to convert the YUV image in RGB and vice versa.

3.1.3 Usage

Cereal main directory exposes the files containing the definitions of the different messages, and, together with methods to retrieve the information related to available services. The different components of *selfdrive* use the structs defined in log.capnp and car.capnp to serialize the different data streams.

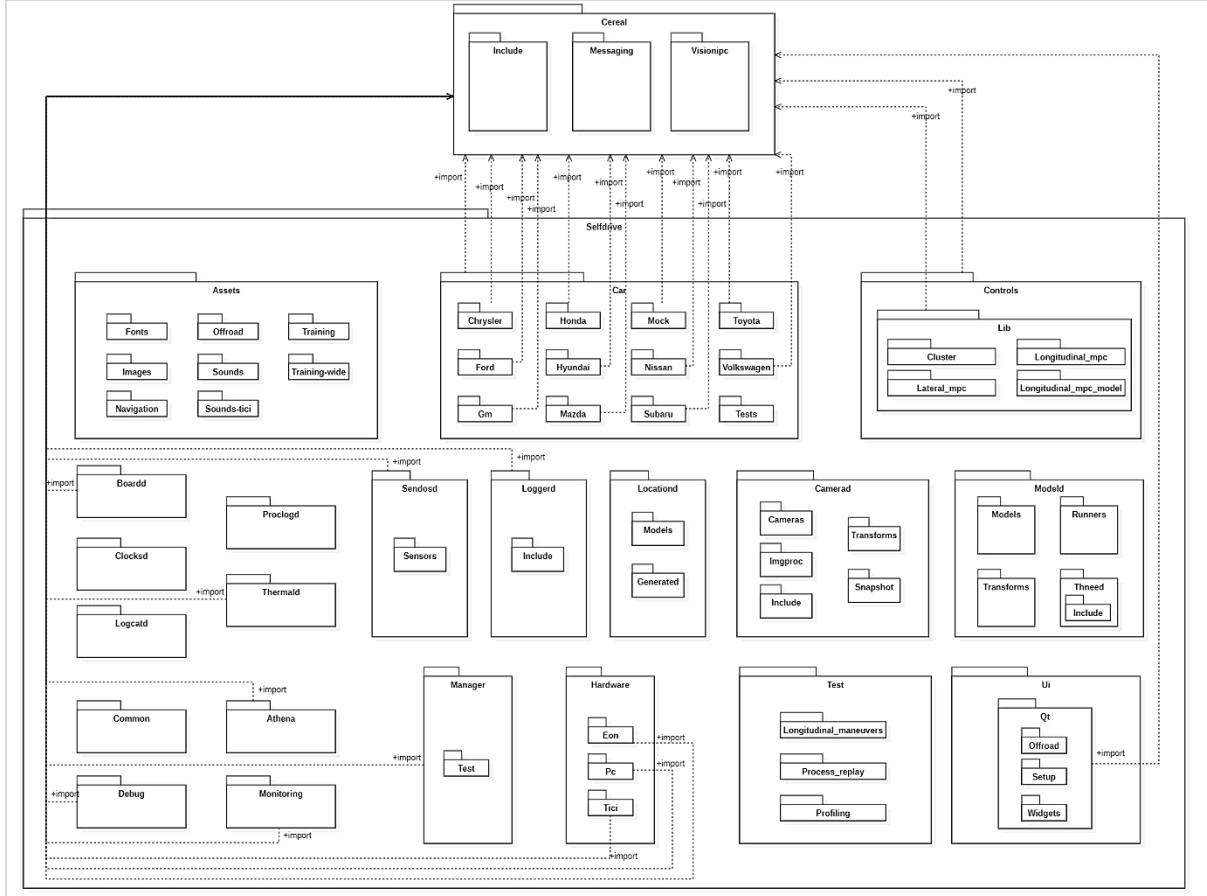


Figure 9 - Import dependencies between *selfdrive* and *cereal*

In particular, the service daemons like *boardd*, *locationd*, *loggerd*, *sensorsd*, and *thermal*, leverage this communication protocol to transmit the data acquired by the different sensors and the message specifications defined in the Cap'n Proto files allow building well-structured messages. For instance, in the case of sensors, managed by the process *sensorsd*, each sensor has a `get_event()` method, through which it retrieves the value recorded by the specified sensor.

```
event.setSource(cereal::SensorEventData::SensorSource::RPR0521);
event.setVersion(1);
event.setSensor(SENSOR_LIGHT);
event.setType(SENSOR_TYPE_LIGHT);
event.setTimestamp(start_time);
event.setLight(value);
```

3.1.3.1 Messaging - usage

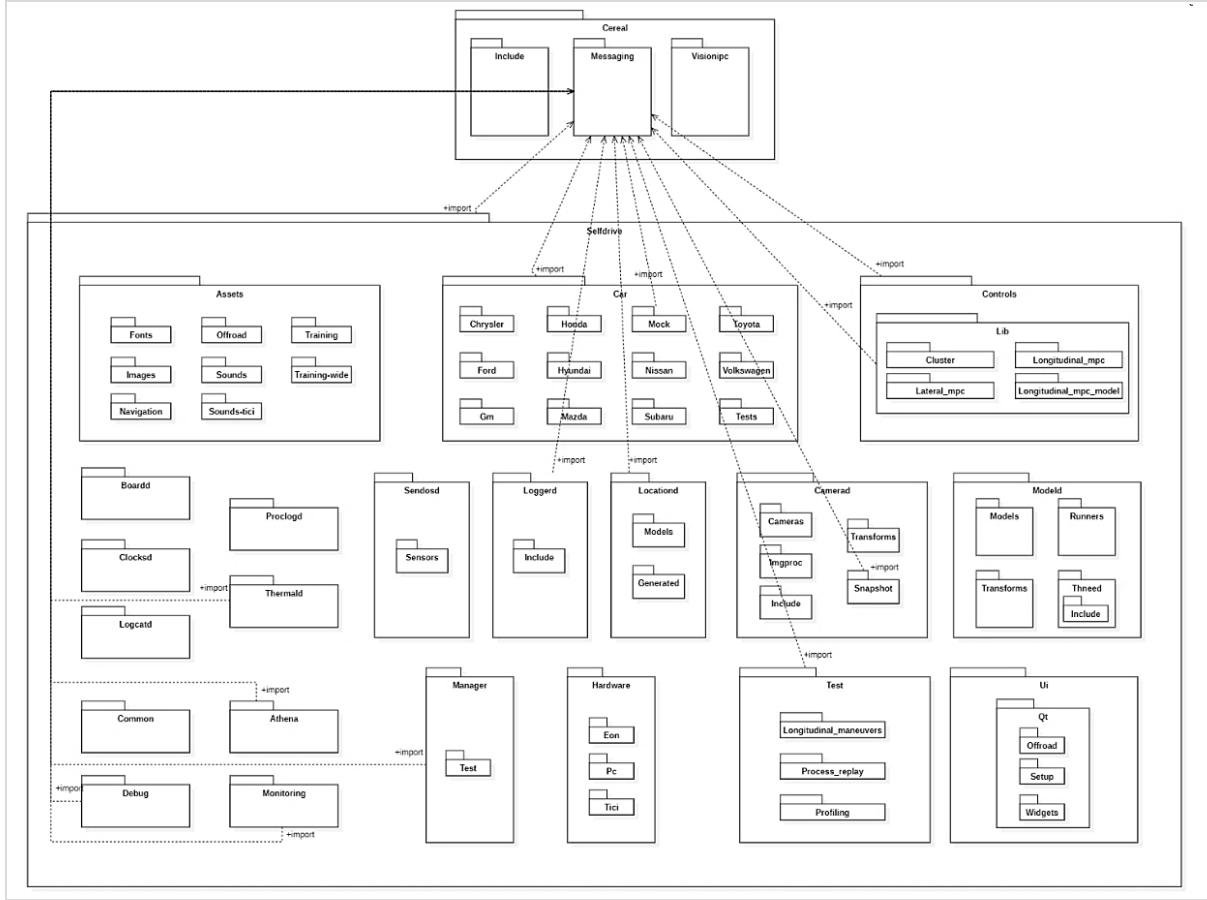


Figure 10 - Import dependencies between selfdrive and cereal/messaging

As we can see from the diagram in *Figure 10*, the messaging functionalities are largely used in the other component of Openpilot. All the software components that need to send and receive messages to other software components use cereal to establish a Pub-Sub connection. To simplify the instantiation of the PubSocket and SubSocket, the library provides two components, the PubMaster and SubMaster, that automatically instantiate and initialize the publisher and the subscribers for all the specified services.

In camerad, for instance, the component that manages the acquisition of the camera frames, are instantiated both a PubMaster and a SubMaster.

```
s->sm = new SubMaster({"driverState"});
s->pm = new PubMaster({"roadCameraState",
                      "driverCameraState",
                      "thumbnail"});
```

This means that the camera component listens to data arriving on the socket “*driverState*”, containing details on the state of the diver, and publishes the data acquired by the road camera and the driver camera on the sockets *roadCameraState* and *driverCameraState*, respectively.

The data sent on the sockets by the PubMaster are then received by the SubMaster subscribed to the same sockets. For instance, data sent on socket *roadCameraState* are received by *controlsds*, *modeld*, the component managing the alerts and the one managing the snapshots.

```
# controlsd.py
self.camera_packets = ["roadCameraState", "driverCameraState"]
self.sm = messaging.SubMaster(['deviceState', 'pandaState', 'modelV2',
                               'liveCalibration', 'driverMonitoringState',
                               'longitudinalPlan', 'lateralPlan',
                               'liveLocationKalman', 'managerState',
                               'liveParameters', 'radarState'] +
                               self.camera_packets + joystick_packet,
                           ignore_alive = ignore,
                           ignore_avg_freq = ['radarState', 'longitudinalPlan'])
```

```
# cycle_alerts.py
sm = messaging.SubMaster(['deviceState', 'pandaState',
                           'roadCameraState', 'modelV2',
                           'liveCalibration', 'driverMonitoringState',
                           'longitudinalPlan', 'lateralPlan',
                           'liveLocationKalman'])
```

```
// modeld.cc
PubMaster pm({"modelV2", "cameraOdometry"});
SubMaster sm({"lateralPlan", "roadCameraState"});
```

```
# snapshot.py
def get_snapshots(frame="roadCameraState",
                  front_frame="driverCameraState",
                  focus_perc_threshold=0.):
    sockets = []
    if frame is not None:
        sockets.append(frame)
    if front_frame is not None:
        sockets.append(front_frame)

    sm = messaging.SubMaster(sockets)
```

3.1.3.2 VisionIPC - usage

VisionIPC is used by all the components that require to access the frames acquired by the device's cameras [Figure 11]. In particular, the processes accessing the frame data are *camerad* and *modeld*. Moreover, there are also tools and common shared libraries which use VisionIPC to send and retrieve camera frames.

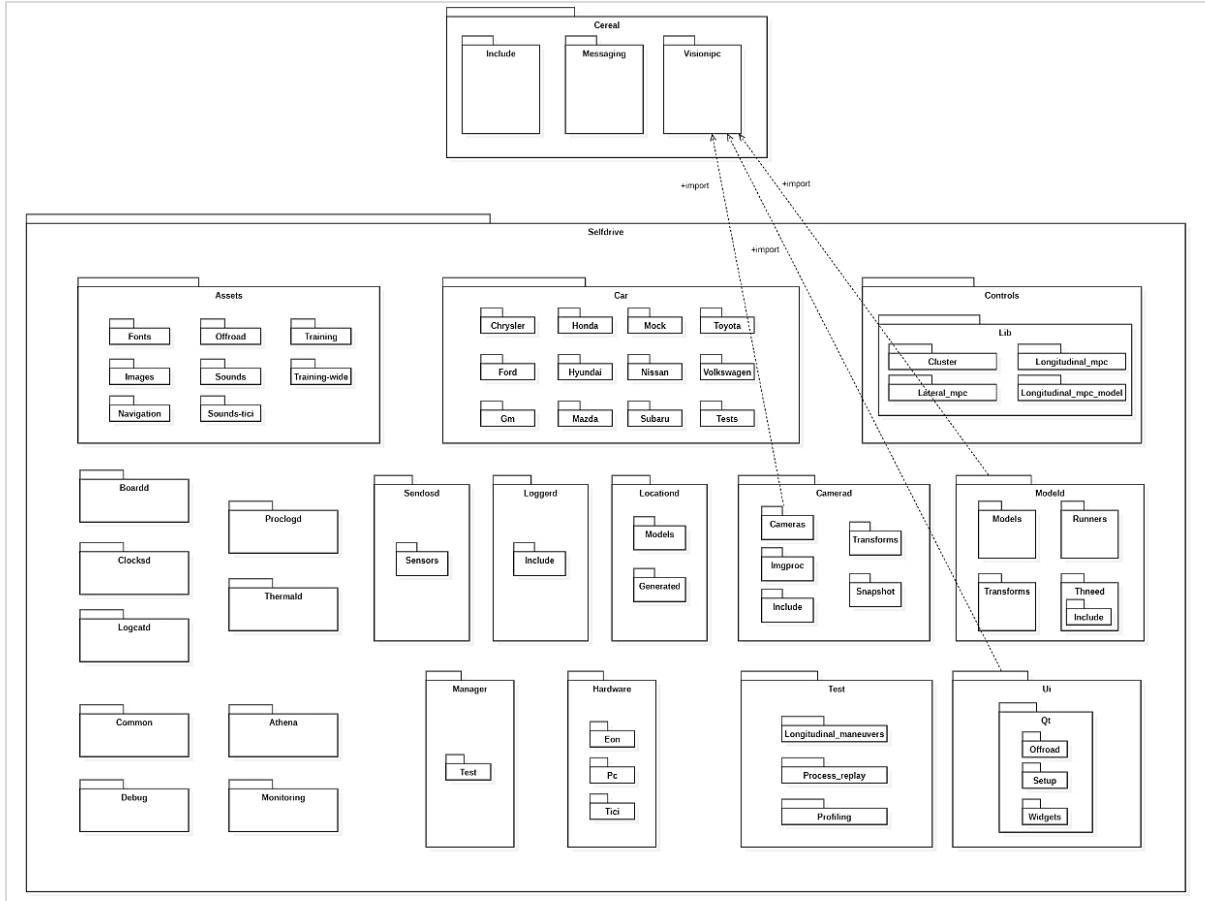


Figure 11 - Import dependencies between selfdrive and cereal/visionipc

In the case of the camera driver, for instance, each camera is represented by a CameraBuf object, which provides the methods to acquire the image and send it to the different VisionIpcClients. The *acquire()* method reads data from the cameras and the instantiated VisionIpcServer transmits them using MSGQ or ZMQ, indifferently.

```
void VisionIpcServer::send(VisionBuf* buf, VisionIpcBufExtra* extra,
    bool sync){
    if (sync) buf->sync(VISIONBUF_SYNC_FROM_DEVICE);

    assert(buffers.count(buf->type));
    assert(buf->idx < buffers[buf->type].size());

    VisionIpcPacket packet = {0};
    packet.server_id = server_id;
    packet.idx = buf->idx;
    packet.extra = *extra;

    sockets[buf->type]->send((char*)&packet, sizeof(packet));
}
```

Both RGB and YUV data are sent using this approach, but data is sent on different sockets.

```
vipc_server->send(cur_rgb_buf, &extra);
vipc_server->send(cur_yuv_buf, &extra);
```

The method *release()* calls a callback which closes the connection.

```
void CameraBuf::release() {
    if (release_callback) {
        release_callback((void*)camera_state, cur_buf_idx);
    }
}
```

The method *queue()*, instead, pushes the data on in the buffer's queue.

```
void CameraBuf::queue(size_t buf_idx) {
    safe_queue.push(buf_idx);
}
```

On the other hand, the model daemon receives the camera data as a VisionIpcClient and passes the received data to the modeler.

```
VisionIpcClient v ipc_client = VisionIpcClient(
    "camerad",
    wide_camera
    ? VISION_STREAM_YUV_WIDE
    : VISION_STREAM_YUV_BACK,
    true,
    device_id,
    context);

while (!do_exit && !v ipc_client.connect(false)) {
    util::sleep_for(100);
}
if (v ipc_client.connected) {
    const VisionBuf *b = &v ipc_client.buffers[0];
    LOGW("connected with buffer size: %d (%d x %d)",
        b->len,
        b->width,
        b->height);
    run_model(model, v ipc_client);
}
```

3.1.4 Testing

messaging is tested by using the *unittest* framework. A unit test is a small test that checks that a single component operates in the right way, and it does that by comparing the result of the component with the expected result. To test the behavior with the different types of events, parametrized tests are widely used: they allow passing a list of tuples and running the test for each one of the elements in the list.

- **TestPubSubSockets:** it contains three tests that aim to test the connection between a publisher and a subscriber, instantiating the different sockets and checking the robustness of the library in case of conflated or delayed behaviors. [Table 2]

TestPubSubSockets
+ setup()
+ test_pub_sub()
+ test_conflate()
+ test_receive_timeout()

Table 2 - *TestPubSubSockets* test case

- **TestMessaging:** it includes eight tests that check the capability of the sockets to receive different types of messages and their capability to handle errors. It also makes use of the parameterized test to test the socket for each service available. [Table 3]

TestMessaging
+ setUp()
+ test_new_message()
+ test_pub_sock()
+ test_sub_sock()
+ test_drain_sock()
+ test_recv_sock()
+ test_recv_one()
+ test_recv_one_or_none()
+ test_recv_one_retry()

Table 3 - *TestMessaging* test case

- **TestPoller:** its five tests verify the ability of the Poller to send one or more messages to one or more subscribers. The Poller also has to ensure that there is only one publisher sending messages on a certain socket, otherwise, it has to raise an exception. [Table 4]

TestPoller
+ test_poll_once()
+ test_poll_and_create_many_subscribers()
+ test_multiple_publishers_exception()
+ test_multiple_messages()
+ test_conflate()

Table 4 - *TestPoller* test case

- **TestSubMaster:** this test case comprehends nine tests that check the functioning of the component of the library responsible for the connection of the different subscribers to a socket and of the reception of messages. Three of these tests, however, are only needed to check if the components are still running and alive. [Table 5]

TestSubMaster
+ setUp()
+ test_init()
+ test_init_state()
+ test_getitem()
+ test_update()
+ test_update_timeout()
+ test_alive()
+ test_ignore_alive()
+ test_valid()
+ test_conflate()

Table 5 - TestSubMaster test case

- **TestPubMaster:** it contains only two tests, one that initializes the PubMaster and one that checks the sending of a response to different sockets, using the different protocols. [Table 6]

TestPubMaster
+ setup()
+ test_init()
+ test_send()

Table 6 - TestPubMaster test case

- **TestServices:** its three tests check that there cannot be duplicate ports and that the generated files are valid C files for all the services. [Table 7]

TestServices
+ test_services()
+ test_no_duplicate_port()
+ test_generated_header()

Table 7 - TestServices test case

The unit tests are executed with GitHub Actions, which executed the tests whenever a new commit is made, to ensure that each change doesn't modify the behavior of the software and work as expected.

For what concerns VisionIPC, it is tested through Catch2, a unit testing framework for C++. Catch2 allows defining tests easily and naturally, without the need of naming them with valid identifiers or registering them. *visionipc_tests.cc* include six different unit tests: connecting, check buffers, check YUV/RGB, send single buffer, test no conflate, test conflate.

3.1.5 Development and community contribution

Many users contributed to the development of cereal. In particular, the possibility to test the messaging specs on different car models and compare their behavior speeded up the whole development process. Cereal was made a public GitHub repository on the 13th of June 2019.

initial commit, internal from 6/13/19 (13/06/2019) <George Hotz>

Currently, *cereal* repository counts more than 100 forks and over 30 contributors. When published, the community, as well as the Comma.ai staff, focused on increasing the events supported by the messaging specifications. The allowed modifications and additions had to follow precise guidelines.

The development of *cereal* started before Comma.ai made the repository public, and new developers would find the messaging specification already in place, but they were free to modify and add new specifications and parameters according to their needs and findings during the tests on the road.

Not only external contributors, but also the Comma.ai team at the beginning focused on adding different parameters to support the events that were logged during the execution of Openpilot [Table 72]. If we consider the commits which modify the files *car.capnp*, *log.capnp*, and *legacy.capnp*, the ones containing the specifications of different types of messages corresponding to the events, represent more than half of the total amount of commits of this repository.

The addition of new parameters was more frequent when the repository was made public, while now the community focuses more on stability and performance issues.

After the publication of the specifications of the events in the public repository, also the messaging services were made public and moved to cereal.

import messaging and services (01/11/2019) <George Hotz>

fixups (01/11/2019) <George Hotz>
--

fix internal refs (01/11/2019) <George Hotz>

The first problems that the team had to deal with were related to the remote connection to the messaging services. This included managing the connection and deletion of the SubSockets and make possible to pass a specific address to which the socket could connect.

```
def connect(self, Context context, string endpoint,
           string address=b"127.0.0.1",
           bool conflate=False):
    self.socket.connect(context.context, endpoint, address, conflate)
def connect(self, Context context, string endpoint,
           string address=b"127.0.0.1",
           bool conflate=False):
    self.socket.connect(context.context, endpoint, address, conflate)
```

remove extra underscore from __dealloc__ (03/11/2019) <andyh2>

only delete subsocket when created by same object (04/11/2019) <Willem Melching>

remote address support (04/11/2019) <Willem Melching>
--

Don't delete context from python side only (04/11/2019) <Willem Melching>
--

A big improvement in terms of performance was brought by the introduction of MSGQ. MSGQ was disabled when first introduced and then made the default in version 0.7.1.

The implementation of MSGQ required some time, with the staff focusing on improving the protocol stability and performances.

add all msgq files, but don't use as default (05/11/2019) <Willem Melching>
Fix service list path in bridge (05/11/2019) <Willem Melching>
zmq already sets the errno correctly (05/11/2019) <Willem Melching>
add c exports for jni usage (07/11/2019) <andyh2>
fix export prefix and make shared library world readable (07/11/2019) <andyh2>
default to msgq (08/11/2019) <Willem Melching>
MSGQ stability improvements when opening and closing lots of queues (15/11/2019) <Willem Melching>
also remove the FIFO from disk (16/11/2019) <Willem Melching>
msgq: don't clean up uninitialized sockets (16/11/2019) <Willem Melching>
msgq: make sure read_fifos is initialized so we don't close random fds (16/11/2019) <Willem Melching>
msgq: try again when no timeout on poll but also no message (18/11/2019) <Willem Melching>
Default to zmq (19/11/2019) <Willem Melching>
msgq: don't block when fifo does not exists (19/11/2019) <Willem Melching>
add msgq tests (19/11/2019) <Willem Melching>
Switch from polling on FIFOs to signal (#12) (22/11/2019) <George Hotz>
Switch default to msgq (#21) (13/01/2020) <Willem Melching>

New tests were also created to validate the new functionalities added. The tests were also added to GitHub CI, to make them run automatically when a user makes a new commit.

Run scons in CI (#14) (21/11/2019) <Willem Melching>
run python unittest in ci (21/11/2019) <Willem Melching>
add test with multiple subscribers (21/11/2019) <Willem Melching>

It has also been implemented an error handling mechanism and two new types of exceptions, *MultiplePublishersError* and *MessagingError*, thrown when the connection fails.

Implement error handling and exceptions (#18) (04/12/2019) <Willem Melching>

```
SubSocket * create(Context * context, string endpoint, string address,
bool conflate) {
    SubSocket *s = SubSocket::create();
    int r = s->connect(context, endpoint, address, conflate);
    if (r == 0) {
        return s;
    } else {
        delete s;
        return NULL;
    }
}
```

The first external contribution to this library was made by a user trying to make cereal work on macOS. **Paweł Goliński** pointed out the fact that MSGQ was not supported by macOS due to the absence of a shared memory filesystem, therefore he proposed to use ZMQ as default for the operative system [7].

Use ZMQ on MacOS (#46) (21/05/2020) <Paweł Goliński>

The same user also explicated a silent failure causing a segmentation fault while using Openpilot tools [8].

Fix potential segfault in MSGQPubSocket:connect (21/05/2020) <Pawel Goliński>
--

An important change, aimed to simplify and improve how cereal could allow exchanging messages among the master and its subscribers, was made by an external contributor, **Dean Lee**, with the support of **Willem Melching**, a member of the Comma.ai team. The introduction of a SubMaster and a PubMaster allowed to create a SubSocket for all the specified sockets, and the PubMaster, sending the latest messages to all its subscribers [9].

C++ implementation of SubMaster and PubMaster (21/05/2020) <Dean Lee>
--

submaster always conflates (22/05/2020) <Willem Melching>
--

Other changes were made to lint the code through *flake8*, a command-line utility for enforcing style consistency across Python projects. It was added to the workflow of the GitHub Actions also a static analysis check, executed before each commit.

add pre-commit static analysis (#48) (29/05/2020) <Willem Melching>
--

two spaces before inline comment (31/05/2020) <Adeeb Shihadeh>

fix flake8 complaint about too many blank lines (01/06/2020) <Adeeb Shihadeh>
--

whitespace fix (01/06/2020) <Adeeb Shihadeh>

enable almost all flake8 checks (01/06/2020) <Adeeb Shihadeh>
--

fix dereferencing of full_path after free (08/06/2020) <Adeeb Shihadeh>
--

After the revision of the previous changes from the Comma.ai staff, George Hotz pointed out some issues related to the implementation of the new Master classes. In particular, the file containing those classes was placed in the main folder of */cereal* and not in */cereal/messaging*. The author of the changes initially placed the file socketmaster.cc in */cereal* because it would make the execution of offroad.apk if placed in */cereal/messaging*, as specified in pull request #58 [10]. Both the problems were then fixed by the same author [11].

fix mac build (#50) (12/06/2020) <George Hotz>

More tests had to be added to validate the new classes and lower-level functions, and smaller fixes were also made to clean up the code and be compliant with the constraint imposed by *flake8*.

Messaging unit tests (#66) (27/07/2020) <Adeeb Shihadeh>

Update catch2 and move to Dockerfile (#71) (27/07/2020) <Adeeb Shihadeh>

Tests for lower level messaging functions (#73) (28/07/2020) <Adeeb Shihadeh>
--

remove get_one_can (29/07/2020) <Adeeb Shihadeh>

simple service tests (29/07/2020) <Adeeb Shihadeh>

test generated services.h (29/07/2020) <Adeeb Shihadeh>
--

fix test file path (29/07/2020) <Adeeb Shihadeh>

Smaller fixes to the socket class constructor were also made to improve their usability and reduce their memory usage. For instance, the parameter types and the return types were made explicit for all the methods available in *messaging.h* and parallel execution was enabled when building Cython extension.

expose frame and recv_frame in C++ submaster (02/08/2020) <Adeeb Shihadeh>

parallelize Cython extension build (02/08/2020) <Adeeb Shihadeh>

Add type hints to messaging (#82) (23/08/2020) <Adeeb Shihadeh>

larger shared memory size on computer for frames (28/08/2020) <Willem Melching>

It was possible to further reduce the CPU usage by using only one Poller for each SubMaster and by making the request without a specified service non-blocking. This is particularly useful to reduce the usage of processes that should run at 10Hz but instead run at 100Hz when subscribing to the socket *carState*. [12]

allow prioritization of services in SubMaster (#84) (02/09/2020) <Adeeb Shihadeh>

Smaller fixes were then made, one of which was made by an external contributor, **Gregor Kikelj**, and fixed an issue opened by **Willem Melching**, a member of the Comma.ai team.

The issue [13] was about finding a way to show a warning when subscribing/publishing to a service queue that was not in the service list, and it was solved by checking if the considered service was in the list of available services.

```
bool service_exists(string path) {
    for (const auto& it : services) {
        if (it.name == path) {
            return true;
        }
    }
    return false;
}
```

fix the issue 41 (#87) (11/09/2020) <Gregor Kikelj >

To better manage the creation of new messages, it was introduced a MessageManager class to simplify the code and utilize the same class rather than calling the same methods multiple times each time that a message had to be created.

Custom message builder (#72) (13/09/2020) <Dean Lee>

In particular, a community member, **Alexander Litzenberger**, was able to improve the performances and brought a speed up by changing the type of messages exchanged using the method *send()* from *string* to *byte*.

Switch send to using bytes (#93) (07/10/2020) <Alexander Litzenberger>

A code refactoring was made due to the violation of some best practices that made it hard to understand what the different parameters were actually needed for.

In many cases, the parameters did not mention the System of Unit, used deprecated methods, or had misleading names, and the refactor made by **Harald Schafer** in his pull request [14] aimed to fix those bad practices.

Best practice (#107) (17/02/2021) <Harald Schafer>

msgq: fixup larger queue size for frames (22/02/2021) <Willem Melching>

Comparably to what happened with the creation of the class of MessageManager, it was also created the class AlignedBuffer. AlignedBuffer allows to simply manage a buffer, made by an Array of words, without the need of managing the creation and deallocation of it every time that is needed.

Added new util class AlignedBuffer (#125) (15/03/2021) <Dean Lee>

AlignedBuffer remove operator() to prevent out of scope usage (17/03/2021) <Willem Melching>

To ensure that the update frequency was adequate it was introduced a method to check if the average frequency, and if higher than 90% of the expected value then the subscriber could be considered *alive*.

Add check for average frequency (#128) (06/04/2021) <Willem Melching>

don't check avg freq in simulation (08/04/2021) <Willem Melching>

add list for average frequency ignore (#132) (08/04/2021) <Willem Melching>

To simplify the process of verifying if all the subscribers to a certain socket are *alive* and *updated* a new getter function that performed this check was introduced. The pull request with the additions, as well as other small fixes [15], was made by a community member, **Joost Wooning**.

some fixes and small changes for locationid in C++ (#135) (19/04/2021) <Joost Wooning>

More improvements were then made to the part dealing with the connection of the services, making the service assign the ports dynamically rather than statically. Also in this case, the pull request was made by a community member, **Shane Smikol** [16].

Automatically generate service ports (#136) (20/04/2021) <Shane Smiskol>

The last optimizations made to the library regards mostly the consideration of some particular cases that could lead to some unexpected behaviors.

For example, the method `msgq_all_readers_updated()` was modified to make it return the value *true* only if there is at least one connected user, and the liveness of a subscriber can be checked only when not in a simulation.

SubMaster: split socket recv and update (#133) (23/04/2021) <iejMac>

msgq_all_readers_updated: only return true when at least one reader is connected (05/05/2021) <Willem Melching>

SubMaster: Don't check alive when SIMULATION env variable is set (05/05/2021) <Willem Melching>

C++ SubMaster: make readers always valid (#139) (15/05/2021) <Dean Lee>

Also, the types of some variables were changed to save space and increase the performance. Other changes were then made to increase the number of subscribers that can connect to the same socket.

construct PubMaster with vector instead of initializer list (15/05/2021) <Adeeb Shihadeh>

SubMaster - allow dynamic service lists (#150) (18/05/2021) <Shane Smiskol>

msgq: bump num readers to 10 (21/05/2021) <Adeeb Shihadeh>

Unbridge - for republishing msgs sent from your laptop to Openpilot (#160) (08/06/2021) <Shane Smiskol>

SubMaster: set traversalLimitInWords to max (#188) (17/08/2021) <Dean Lee>

For what concerns VisionIPC, the library has not been modified in many years from its first implementation since it was the most effective way to transmit frame data.

After the introduction of MSGQ, multiple attempts of removing VisionIPC were made, and the head of Openpilot George Hotz even offered a prize of \$500 to the author of the pull request which fulfilled the objective [17]. The bounty was then closed, since removing VisionIPC could not bring any tangible performance improvement.

VisionIPC was included in the *cereal* package only in a second moment when it was rewritten and updated to improve performances and usability. In the beginning, it was located directly in the *selfdrive* directory and it was not open source. The rewriting of VisionIPC required 64 commits made by **Adeeb Shihadeh** and **Willem Melching** [18]. The main changes included the introduction of the classes currently used for managing the client and the server, VisionIpcClient and VisionIpcServer respectively, and VisionBuf to manage the buffer of the acquired frames. It was also adopted the messaging specification included in the cereal package to create the sockets and transfer data among the publisher and the different subscribers.

Visionipc v2.0 (#101) (08/01/2021) <Willem Melching>

More minor changes were made, including the addition of a timeout to the client-side after which the connection, if not receiving any data, could be dropped, and of parameters to the server-side which help to manage the persistence of the connection, for instance, it has been forced that the server buffer can be freed only by the server itself and invalid buffer type no longer stop the execution of the server.

only free ION buffer in server (11/01/2021) <Willem Melching>

visionipc add timeout param (19/01/2021) <Willem Melching>

don't crash on invalid buffer type (19/01/2021) <Robbe Derkx>

To improve the usability of the VisionIPC server functionalities has been included a Cython wrapper for the VisionIpcServer class, to give access to the C++ implementation of the server also from Python code and use its main feature like if it was a Python library.

Cython wrapper for VisionIpc server (#117) (09/02/2021) <Willem Melching>

```
cdef extern from "visionipc_server.h":  
    cdef cppclass VisionIpcServer:  
        VisionIpcServer(string, void*, void*)  
        void create_buffers(VisionStreamType, size_t, bool, size_t, size_t)  
        VisionBuf * get_buffer(VisionStreamType)  
        void send(VisionBuf *, VisionIpcBufExtra *, bool)  
        void start_listener()
```

Small fixes were then made to the header files of the package: to unify the style with the rest of the packages, the headers extension has been modified from *hpp* to *h*, and the path of inclusion of the headers has been fixed by making explicit also the folder where the different headers are located.

rename headers: hpp -> h (04/05/2021) <Adeeb Shihadeh>

fix cython visionipc for qcom (29/05/2021) <Adeeb Shihadeh>

cleanup include paths (#165) (10/06/2021) <Dean Lee>

The previous changes on the management of the free of the buffer were reverted [19] due to some details found on a dedicated forum which suggested to always free the buffer, not considering the author of the free request or failure while freeing the buffer [20].

visionipc: increase max fds to 128 (10/07/2021) <Adeeb Shihadeh>

Always free ION buffer (#183) (26/07/2021) <Willem Melching>

ignore failures on ION buffer free (28/07/2021) <Willem Melching>

As pointed out by many users, in the *cereal* repository there is not a clear guide of how the messages are structured and how *Openpilot* makes use of it to communicate with the car. It is not rare that a new developer, willing to collaborate, is pulled down by the complexity of the project and by the absence of a clear guide on how to use each component correctly.

Fortunately, the community is full of developers that have a lot of experience with the project and can help the newcomers to better understand how the software is structured and point out the key aspect to consider when contributing to the project.

Also, this allows responding more quickly and effectively than it would be if the only people able to reply to technical questions were the Comma.ai staff.

Overall, many developers who were not part of the staff of Comma.ai gave their contribution to *cereal* by testing the library, trying to make *Openpilot* communicate with different car models, fixing bugs, and adding new features.

3.2 Common

The common package contains methods, variables, and processes that are used by all the other packages to perform common operations.

3.2.1 Package structure

The package [Figure 12] includes a simple Kalman filter, the methods needed to call the REST API, proprietary of Comma.ai, and transformation utilities, needed to manipulate the acquired camera frames.

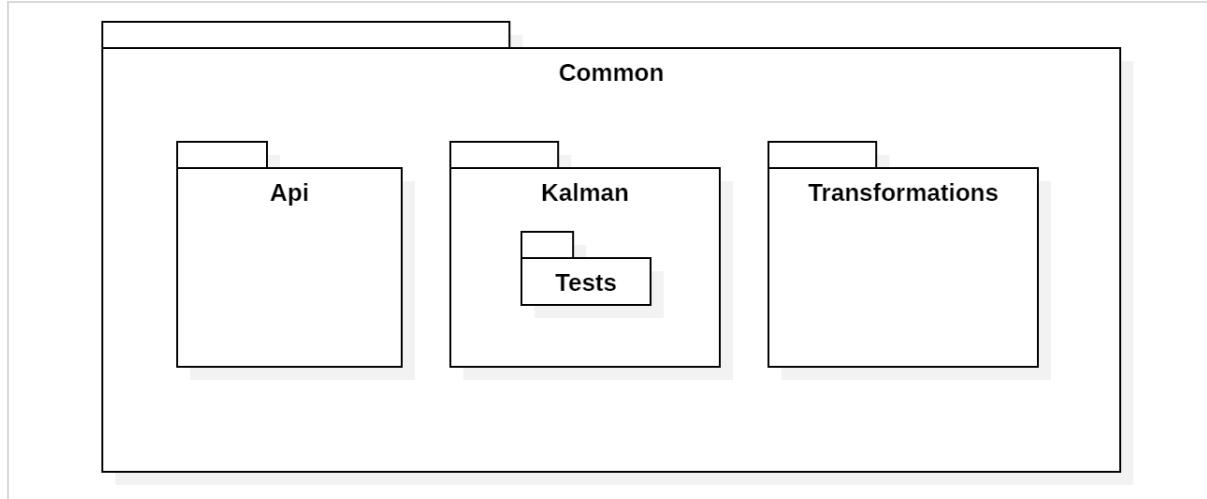


Figure 12 - Package diagram of the common directory

The utility functions included in the common directory are mainly written in Python, but most of the code, represented by the implementation of *Kalman* and *transformation*, is written in C++.

Language	Files	Blank	Comment	Code
C++	6	1.531	4.392	23.117
Python	25	288	138	1.064
Cython	4	72	17	255
C/C++ Header	2	10	0	48
SUM:	37	1.901	4.547	24.484

Table 8 - Lines of code for the Common, by programming language

3.2.2 Implementation

Among the different utility functions that come with the *common* package, we can find functions that help to manipulate or generate files, for example using CFFI to generate C code to import in Python modules and compile the generated code.

```
def compile_code(name,
                 c_code,
                 c_header,
                 directory,
                 cflags = "",
                 libraries = None):
    if libraries is None:
        libraries = []

    ffibuilder = FFI()
    ffibuilder.set_source(name,
                          c_code,
                          source_extension='.cpp',
                          libraries=libraries)
    ffibuilder.cdef(c_header)
    os.environ['OPT'] = "-fwrapv -O2 -DNDEBUG -std=c++1z"
    os.environ['CFLAGS'] = cflags
    ffibuilder.compile(verbose=True, debug=False, tmpdir=directory)
```

Other utilities provide timing information, including the time since the boot of the device and the real-time.

```
cdef double readclock(clockid_t clock_id):
    cdef timespec ts
    cdef double current

    clock_gettime(clock_id, &ts)
    current = ts.tv_sec + (ts.tv_nsec / 1000000000.)
    return current

def sec_since_boot():
    return readclock(CLOCK_BOOTTIME)
```

Many functions supporting the logging of the different events are available, and they aim to both format correctly the logs and record the different events. Logging is supported by the *logging* library available for Python, which provides a standard library module so that all the Python modules can participate in logging: in this way the application log can include both custom messages and messages from third-party modules.

The package also includes different processes directly used by selfdrive, for instance, showing the loading spinner and dealing with the timeout exceptions.

3.2.2.1 API

Comma.ai exposes on the website <https://api.commandotai.com/> a series of web APIs that can be accessed to get different types of information. More in general, a web API is a programmatic interface consisting of one or more publicly exposed endpoints to a defined request-response message system, in this case, expressed in JSON. The available endpoints for Comma.ai API are:

- Account
 - **Profile:** returns information about the authenticated user.
 - **Devices:** list devices owned or readable by the authenticated user.
- Device
 - **Device info:** returns an object representing a comma device.
 - **Update device properties:** update device alias and/or vehicle_id.
 - **Device location:** returns a gpsLocation ZMQ packet from the device. The API server queries Athena and caches the location for the Device Info response.
 - **Pair EON:** pair a comma EON to the authenticated user's account.
 - **Unpair device:** Unpair a device. The authenticated user must be the device owner to perform.
 - **Grant device read permissions to user:** grant read permissions to a user by email. The authenticated user must be the device owner to perform.
 - **Remove device read permissions from user:** remove read permissions from a user by email. The authenticated user must be the device owner to perform.
 - **Device driving statistics:** returns aggregate driving statistics for a device.
 - **Device users:** list users with access to a device.
 - **Device boot logs:** returns most recent boot logs uploaded from a device.
 - **Device crash logs:** returns most recent crash logs uploaded from a device.
- Routes
 - **Segments:** returns time-sorted list of segments, each of which includes basic metadata derived from Openpilot log.
 - **Route Info:** returns information about the provided route. The authenticated user must have ownership of or read access to the device from which the route was uploaded.
 - **Route Segments:** returns a list of segments comprising a route. The authenticated user must have ownership of or read access to the device from which the route was uploaded.
 - **Route Manifest:** lists files uploaded for a route.
- Raw driving data
 - **Files:** retrieve uploaded files for a route.
 - **Qlogs:** retrieve uploaded qlogs for a route.
- Video Stream
 - **Rear Camera Stream:** returns rear camera HLS stream index of MPEG-TS fragments.
 - **Front Camera Stream:** returns front camera HLS stream index of MPEG-TS fragments.
- Derived Data
 - **GPS Path:** after Openpilot uploads a log, a JSON array of GPS coordinates interpolated at 1hz is exposed at a signed URL in the cloud. As segments are uploaded, route.coords will be appended atomically.
 - **Video Frames (every 5s):** JPEGs are extracted every 5s from road camera video and are exposed at a signed URL in the cloud.

- Vehicles
 - **Vehicle by ID:** retrieve vehicle metadata by ID.
 - **List Vehicle Makes:** list available vehicle makes and years.
 - **List vehicles by make:** returns JSON array of vehicle metadata
- Openpilot
 - **Openpilot auth:** authenticate and return the *dongle_id* and *access_token*.
 - **Athena WebSocket:** Listen for inbound JSON-RPC requests.
 - **Upload URL:** Request a URL to which an Openpilot file can be uploaded via PUT request. This endpoint only accepts tokens signed with a device's private key.
- Athena
 - **Athena HTTP:** send JSON-RPC requests to the active web socket client for a given device identified by Dongle ID.
- Annotations
 - **My Annotations:** list annotations contributed by the authenticated user.
 - **Device Annotations:** list annotations contributed to drives from a device that authenticated user has ownership of or read access to.
 - **Annotation by ID:** returns annotation object. The authenticated user must have contributed the annotation.
 - **Create Annotation:** create a new annotation.
 - **Delete annotation:** delete an annotation by ID. The authenticated user must have contributed the annotation.
 - **Update annotation:** update "data" object. The authenticated user must have contributed the annotation.
- Leaderboard
 - **List top leaders:** returns overall and last-week comma point leaders.
- Billing
 - **Subscription Status:** retrieve subscription status for a device
 - **Activate Subscription:** activate the subscription for a device, optionally with a compatible SIM card.
 - **Get Payment Source:** get the current payment source
 - **Set Payment Source:** update payment source associated with an account. Stripe securely stores all card information.
 - **Cancel Subscription:** cancel the subscription for the device.

3.2.2.2 Kalman

The *kalman* package includes the implementation of a simple Kalman filter, needed to smooth the series of values acquired from the different actuators. Kalman filtering, also known as linear quadratic estimation (LQE), is an algorithm that uses a series of measurements observed over time, containing statistical noise and other inaccuracies, and produces estimates of unknown variables that tend to be more accurate than those based on a single measurement alone, by estimating a joint probability distribution over the variables for each timeframe. The variables involved in the calculation of the smoothed values are:

- **A:** state transition matrix, which applies the effect of each parameter of the previous state on the next state. (3.2)
- **K:** Kalman gain. It is calculated from the state covariance matrix and the measurement covariance matrix. The higher the noise measurement, the lower the value of K, and vice versa. (3.2)

- **x0**: initial state, it contains the state of the system i.e., the parameters that uniquely describe the current position of the system. (3.4)
- **C**: state covariance matrix, it models uncertainty in the system. It models the uncertainty of the state vector 'x'. Each of the diagonal elements contains the variance (uncertainty in position) of each of those respective state variables in the state vector. For initialization for this matrix, if the state variable's initial location is known to a high degree, the corresponding diagonal element in P is small. Vice-versa in case the state variable's initial location is not known well. (3.3)

The first step of the algorithm is the prediction step, in which the values of the next system state are predicted. By multiplying the (3.2) and (3.3) we obtain the (3.5). This result is then subtracted from the state matrix A to obtain (3.7).

$$A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \quad (3.1)$$

$$K = \begin{bmatrix} k_{00} \\ k_{10} \end{bmatrix} \quad (3.2)$$

$$C = [c_{00} \ c_{01}] \quad (3.3)$$

$$x_0 = \begin{bmatrix} x_{00} \\ x_{10} \end{bmatrix} \quad (3.4)$$

$$K * C = \begin{bmatrix} K_{00}C_{00} & K_{00}C_{01} \\ K_{10}C_{00} & K_{10}C_{01} \end{bmatrix} \quad (3.5)$$

$$A - K * C = \begin{bmatrix} A_{00} - K_{00}C_{00} & A_{01} - K_{00}C_{01} \\ A_{10} - K_{10}C_{00} & A_{11} - K_{10}C_{01} \end{bmatrix} \quad (3.6)$$

$$\begin{bmatrix} AK_0 & AK_1 \\ AK_2 & AK_3 \end{bmatrix} = \begin{bmatrix} A_{00} - K_{00}C_{00} & A_{01} - K_{00}C_{01} \\ A_{10} - K_{10}C_{00} & A_{11} - K_{10}C_{01} \end{bmatrix} \quad (3.7)$$

```
def __init__(self, x0, A, C, K):
    self.x0_0 = x0[0][0]
    self.x1_0 = x0[1][0]
    self.A0_0 = A[0][0]
    self.A0_1 = A[0][1]
    self.A1_0 = A[1][0]
    self.A1_1 = A[1][1]
    self.C0_0 = C[0]
    self.C0_1 = C[1]
    self.K0_0 = K[0][0]
    self.K1_0 = K[1][0]
    self.A_K_0 = self.A0_0 - self.K0_0 * self.C0_0
    self.A_K_1 = self.A0_1 - self.K0_0 * self.C0_1
    self.A_K_2 = self.A1_0 - self.K1_0 * self.C0_0
    self.A_K_3 = self.A1_1 - self.K1_0 * self.C0_1
```

The second step is the update step, in which the value is corrected using the actual measurement.

$$(A - KC) * x_0 = \begin{bmatrix} AK_0 * x_{00} + AK_1 * x_{10} \\ AK_2 * x_{00} + AK_3 * x_{10} \end{bmatrix} \quad (3.8)$$

$$K * meas = \begin{bmatrix} k_{00} * meas \\ k_{10} * meas \end{bmatrix} \quad (3.9)$$

$$(A - KC) * x_0 + (K * meas) = \begin{bmatrix} AK_0 * x_{00} + AK_1 * x_{10} + k_{00} * meas \\ AK_2 * x_{00} + AK_3 * x_{10} + k_{10} * meas \end{bmatrix} \quad (3.10)$$

$$\begin{bmatrix} x_{00} \\ x_{10} \end{bmatrix} = \begin{bmatrix} AK_0 * x_{00} + AK_1 * x_{10} + k_{00} * meas \\ AK_2 * x_{00} + AK_3 * x_{10} + k_{10} * meas \end{bmatrix} \quad (3.11)$$

The matrix resulting from this operation (3.11) can be used to directly apply the transformation and the result is computed by the method *update()* which returns a state that is updated using the Kalman filter starting from a measurement ‘meas’. [21]

```
def update(self, meas):
    double x0_0 = self.A_K_0 * self.x0_0 +
                  self.A_K_1*self.x1_0 +
                  self.K0_0*meas

    double x1_0 = self.A_K_2*self.x0_0 +
                  self.A_K_3*self.x1_0 +
                  self.K1_0*meas
    self.x0_0 = x0_0
    self.x1_0 = x1_0

    return [self.x0_0, self.x1_0]
```

3.2.2.3 Transformation

Openpilot has to deal with different types of reference frames and is helpful to have the possibility to convert them into a different type of reference frame since all of them are widely diffused throughout the codebase.

This folder contains all helper functions needed to transform generate the different types of reference frames and orientations. Generally, this is done by generating a rotation matrix and multiplying the reference frame. The reference frame adopted by Openpilot are:

- **Geodetic:** is a reference frame for precisely measuring locations on Earth or other planetary body. In geodetic coordinates, Earth's surface is approximated by an ellipsoid, and locations near the surface are described in terms of latitude, longitude, and height.
- **ECEF (Earth-Centered, Earth-Fixed):** represents positions as X, Y, and Z coordinates. The origin (point 0, 0, 0) is defined as the center of mass of Earth, hence the term geocentric coordinates. The distance from a given point of interest to the center of Earth is called the geocentric distance, $R = \sqrt{(X^2 + Y^2 + Z^2)}$, which is a generalization of the geocentric radius, not restricted to points on the ellipsoidal surface.

- **NED (North, East, Down)**: are similar to ECEF in that they are Cartesian, however, they can be more convenient due to the relatively small numbers involved, and because of the intuitive axes.
- **Device**: the frame aligned with the road-facing camera used by Openpilot.
- **Calibrated**: calibrated frame is defined to be aligned with the car frame in pitch and yaw and aligned with the device frame in roll. It also has the same origin as the device frame.
- **Car**: the reference frame defined as aligned with the vehicle. The origin of car frame is defined to be directly below the device frame (in car frame), such that it is on the road plane. The position and orientation of this frame are not necessarily always aligned with the direction of travel or the road plane due to suspension movements and other effects.
- **View**: like the device frame, but also considers the camera conventions.
- **Camera**: like the view frame, but 2D on the camera image.
- **Normalized camera**: generated from the camera frame after applying a normalization process. The normalization is done by dividing the measured data by the focal of the camera.
- **Model**: the sampled rectangle of the full camera frames the model uses.
- **Normalized model**: generated from the model frame after applying a normalization process. The normalization is done by dividing the measured data by the focal of the camera.

The orientation of the different reference frames can be expressed in different ways: quaternions, rotation matrices, and Euler angles are three equivalent representations of orientation and all three are used throughout the code base.

For Euler angles the preferred convention is roll (longitudinal axis), pitch (transverse axis), and yaw (normal axis) which corresponds to rotations around the x, y, and z axes. All Euler angles should always be in radians or radians/s unless for plotting or display purposes.

For quaternions, the Hamilton notations are preferred, which are [qw, qx, qy, qz]. All quaternions should always be normalized with a strictly positive qw. These quaternions are a unique representation of orientation whereas Euler angles or rotation matrices are not.

To rotate from one frame into another with Euler angles the convention is to rotate around the roll, then around pitch, and then around yaw, while rotating around the rotated axes, not the original axes.

The *transformation* folder provides functionalities to transform coordinates, orientations, and models among the different used by Openpilot. For the orientations, the transformation methods available are:

```
Quaternion euler2quat(Vector3)
Vector3 quat2euler(Quaternion)
Matrix3 quat2rot(Quaternion)
Quaternion rot2quat(Matrix3)
Vector3 rot2euler(Matrix3)
Matrix3 euler2rot(Vector3)
Matrix3 rot_matrix(double, double, double)
Vector3 ecef_euler_from_ned(ECEF, Vector3)
Vector3 ned_euler_from_ecef(ECEF, Vector3)
```

The different orientations, instead, can be transformed using the methods:

```

NED ecef2ned(ECEF)
ECEF ned2ecef(NED)
NED geodetic2ned(Geodetic)
Geodetic ned2geodetic(NED)

```

3.2.3 Usage

The *common* directory provides many utility functionalities used by Openpilot and its processes.

- **spinner**: gives information on the progress of a given operation.
- **profiler**: provides a set of statistics that describe how often and for how long various parts of the program are executed.
- **ffi_wrapper**: a foreign function interface (FFI) is a mechanism by which a program written in one programming language can call routines or make use of services written in another. This wrapper allows to compile and use C++ code.
- **text_window**: get a predefined text corresponding to methods return codes.
- **xattr**: is a workaround for the EON/termux build of Python having os.*xattr removed. Extended attributes extend the basic attributes of files and directories in the file system.
- **Filter_simple**: it is a simple implementation of a first-order high-pass filter
- **stat_live**: tracks real-time mean and standard deviation without storing any data
- **dict_helpers**: utility function that removes all keys that end in DEPRECATED.
- **realtime**: get the time passed since when the application is booted.

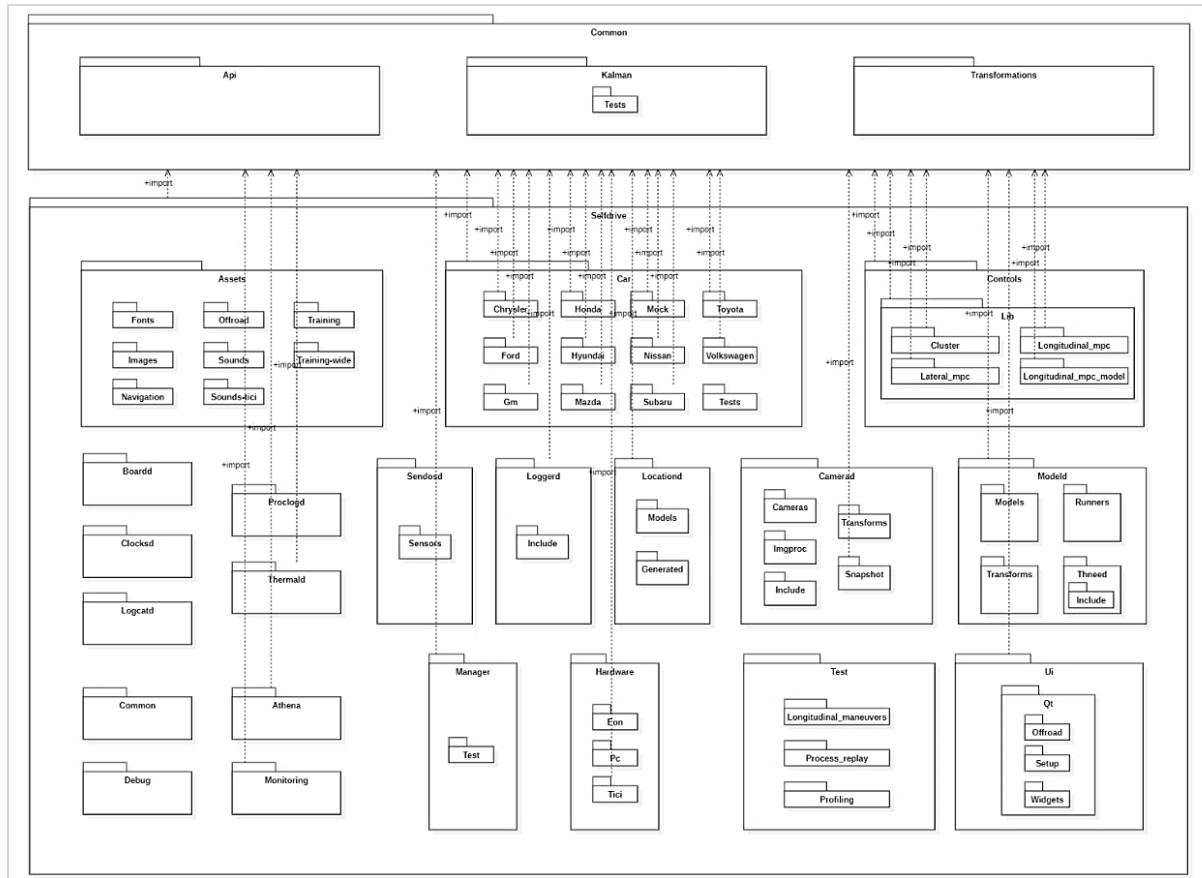


Figure 13 - Import dependencies between selfdrive and common

3.2.3.1 Api - usage

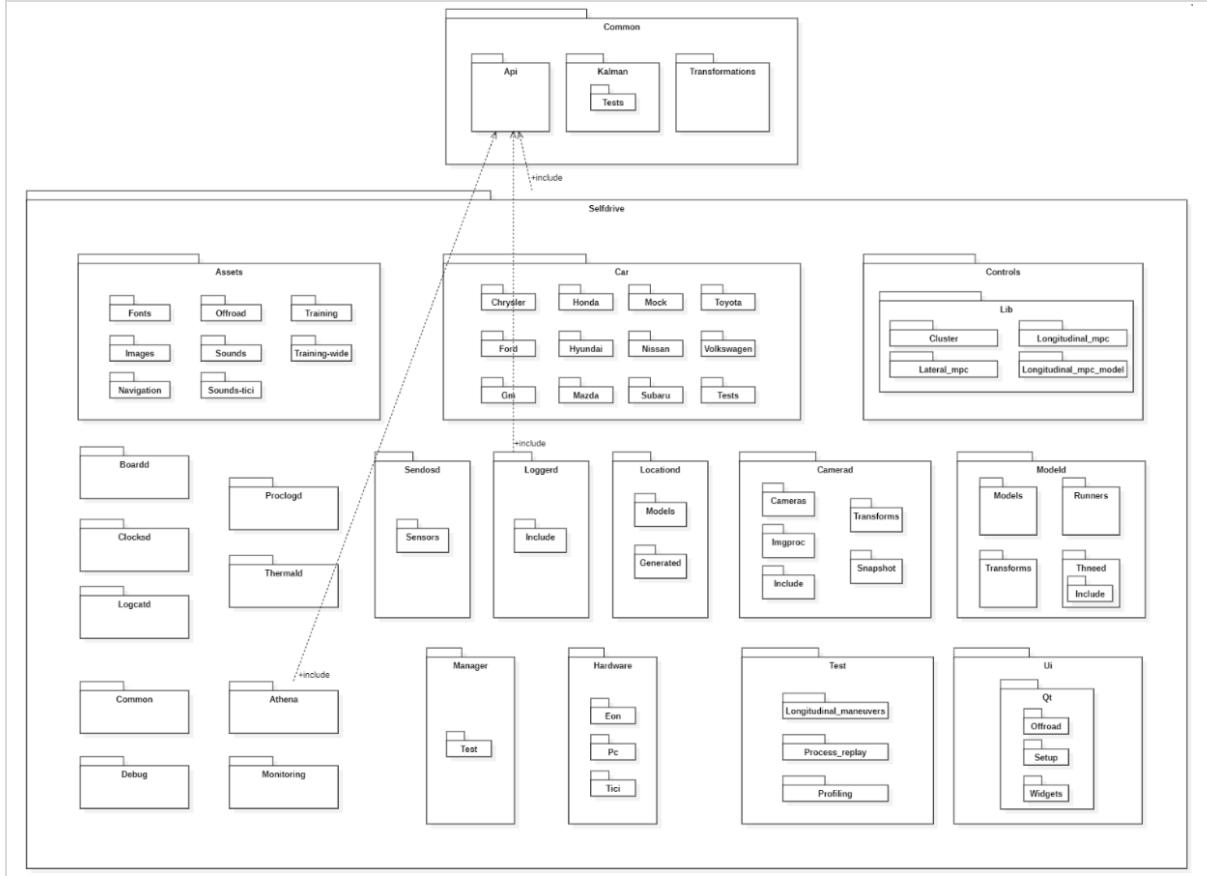


Figure 14 - Import dependencies between selfdrive and common/api

The different processes that leverage the APIs make use of methods to both query the endpoint and receive responses. A request can be made through the method `api_get()`.

```
api_get(endpoint,method='GET',timeout=None,access_token=None,**params)
```

The request is sent to <https://api.commandotai.com/endpoint>. Here is reported an example of a request generated and the relative response from the server:

```
GET /v1/devices/:dongle_id/location

curl 'https://api.commandotai.com/v1/devices/02ec6bea180a4d36/location' \
-H 'Authorization: JWT {{token}}'

Response
{
    "dongle_id": "02ec6bea180a4d36",
    "lat": 32.0,
    "lng": -117.0,
    "time": 1558595762000,
    "speed": 0.0,
    "bearing": 0.0
}
```

The response is a JSON object that is decoded when received by the client.

The method `get_token()`, generates a token by encoding a payload using RSA Signature with SHA-256. The encoding algorithm is an asymmetric algorithm that requires two keys, a public key, and a private key. The private key is stored in an internal folder inside the main directory. The token is then used to establish a connection with the server (also used by the method `api_get()` as `access_token`).

The Comma APIs are largely used by *Athena*, which is the Openpilot component that enables to access remotely the car parameters detected by the Comma device mounted in the vehicle. Athena connects to the API base URL and returns a web socket that manages the exchange of requests and responses.

```
ws = create_connection(ws_uri,
                      cookie="jwt=" + api.get_token(),
                      enable_multithread = True,
                      timeout = 30.0)
```

When Athena receives a request, it sends a response to the client that generated it.

3.2.3.2 Kalman - usage

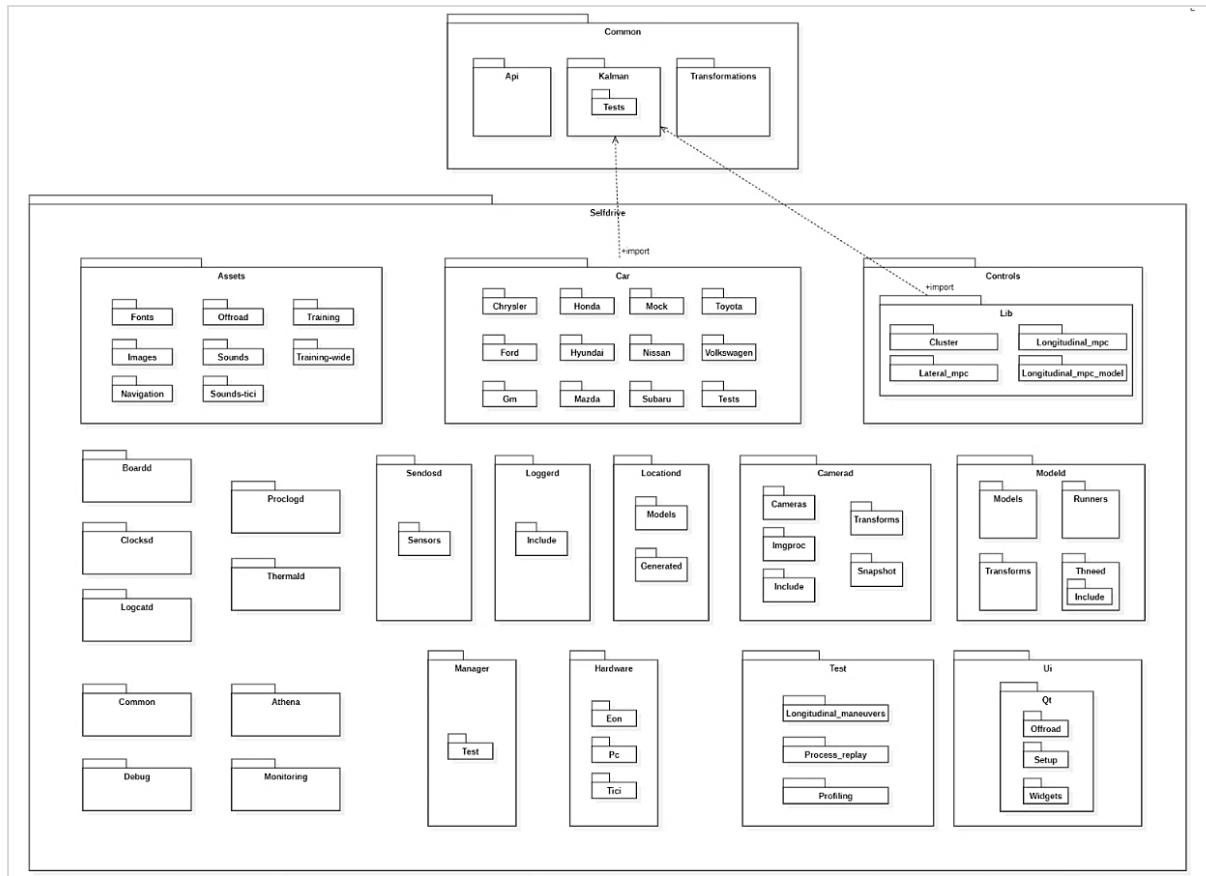


Figure 15 - Import dependencies between selfdrive and common/kalman

The Kalman filter available in `common` folder is used by the process managing the car radar information for Openpilot (`radard`) to precisely estimate the speeds and rotations of the car.

The measured speed is corrected by computing the matrixes using pre-computed values for the state matrixes and the Kalman gain.

```
self.v_ego_kf = KF1D(x0=[[0.0], [0.0]],
                      A=[[1.0, DT_CTRL], [0.0, 1.0]],
                      C=[1.0, 0.0],
                      K=[[0.12287673], [0.29666309]])

def update_speed_kf(self, v_ego_raw):
    if abs(v_ego_raw - self.v_ego_kf.x[0][0]) > 2.0:
        self.v_ego_kf.x = [[v_ego_raw], [0.0]]

    v_ego_x = self.v_ego_kf.update(v_ego_raw)
    return float(v_ego_x[0]), float(v_ego_x[1])
```

In the showed example, the `v_ego_kf` array is first initialized using the Kalman filter and is then used to correct the raw measure. The check on the difference between the raw data and the computed data has the purpose to prevent large accelerations when the car starts at non-zero speed. `v_ego_x` array represents the corrected state and is calculated by applying the `update` function to the raw velocity measurement '`v_ego_raw`'.

3.2.3.3 Transformations - usage

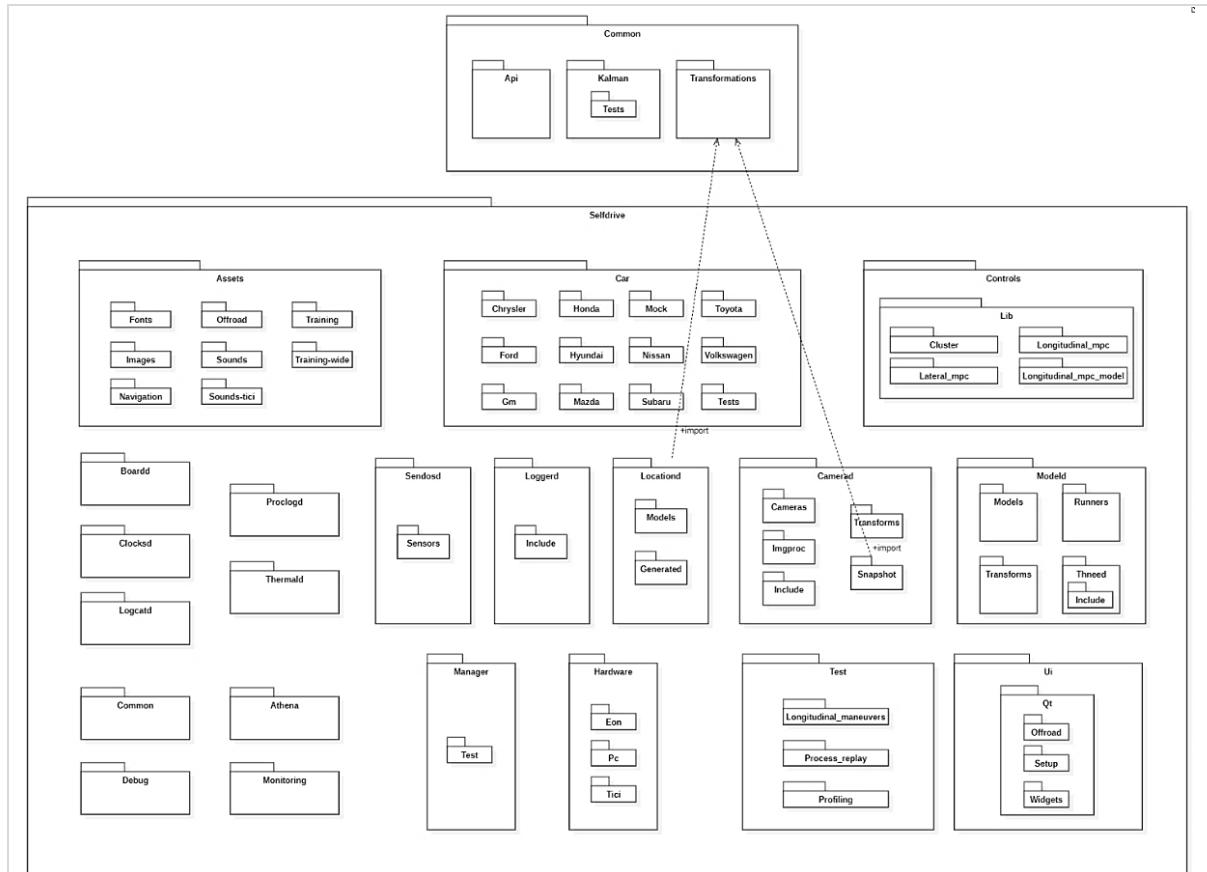


Figure 16 - Import dependencies between selfdrive and common/transformations

The snapshot functionalities, included in *camerad* process, provide methods that return the images acquired by the camera, converting them in JPEG format. The size of the different types of frames that are acquired by the camera are specified in the Transformations package:

```
from common.transformations.camera import eon_f_frame_size,
                                         eon_d_frame_size,
                                         leon_d_frame_size,
                                         tici_f_frame_size
eon_f_frame_size = (1164, 874)
eon_d_frame_size = (1152, 864)
leon_d_frame_size = (816, 612)
tici_f_frame_size = tici_e_frame_size = tici_d_frame_size = (1928, 1208)
```

The precise sizes of the different frames are needed to extract the images from the buffer. R, G, and B arrays are reshaped according to the image's height and width, resulting in bidimensional arrays with each pixel of a given color (red, green, or blue) corresponding to the position [h][w] of the corresponding array. *Np.vstack()* stacks arrays in sequence depth-wise (along the third axis). In this way, each pixel in the position [h][w] will correspond to an RGB color code.

```
def extract_image(dat, frame_sizes):
    img = np.frombuffer(dat, dtype=np.uint8)
    w, h = frame_sizes[len(img) // 3]
    b = img[::3].reshape(h, w)
    g = img[1::3].reshape(h, w)
    r = img[2::3].reshape(h, w)
    return np.vstack([r, g, b])
```

In this way is possible to extract the images captured by the front and rear camera.

```
rear = extract_image(sm[frame].image, frame_sizes)
if frame is not None
else None

front = extract_image(sm[front_frame].image, frame_sizes)
if front_frame is not None
else None
```

The calibration daemon (*calibrationd*) also leverages the transformation methods to calibrate the acquired frames.

Calibrating the acquired images is helpful for Openpilot's driving model since it can be more accurate and make more accurate predictions if the images taken as input by the model look similar also when acquired by other devices mounted differently in different cars. To achieve this, the images are "calibrated" by transforming them into calibrated frames.

3.2.4 Testing

The functionalities of this library are tested through unit testing, and in particular, using the *unittest* framework. There are tests specific for the different packages both for the main package *common* and for the *Kalman* and *transformations* packages. The *common* package includes four test cases and a total of twenty-one tests, also executed through GitHub Actions.

- **TestFileHelpers:** it includes three tests [*Table 9*] that check atomic writes in a temporary file and a specific directory. The test writes a string and checks that the result of the write is the one expected.

TestFileHelpers
+ run_atomic_write_func(atomic_write_func) + run_atomic_write_on_fs_tmp() + run_atomic_write_in_dir()

Table 9 - TestFileHelpers unit test

- **InterpTest:** it includes one test [*Table 10*] which tests the function designed to perform the one-dimensional linear interpolation in a fast and efficient way and compares the result of the interpolation with the result of the one-dimensional linear interpolation calculated using *numpy*.

InterpTest
+ test_correctness_controls()

Table 10 - InterpTest unit test

- **TestParams:** this test case comprehends twelve tests [*Table 11*] that check the behavior when trying to set and retrieve different types of parameters. This includes particular cases such as non-ASCII parameters, parameters with an unknown key, or persistent parameters.

TestParams
+ test_params_put_and_get() + test_persist_params_put_and_get() + test_params_non_ascii() + test_params_get_cleared_panda_disconnect() + test_params_get_cleared_manager_start() + test_params_two_things() + test_params_get_block() + test_params_unknown_key_fails() + test_params_permissions() + test_delete_not_there() + test_get_bool() + test_put_non_blocking_with_get_block()

Table 11 - TestParams test case

- **TestParams(xattr)**: it includes five tests [Table 12] that perform operations on the extended attributes and check the behavior of the library while it manages them. Extended attributes extend the basic attributes of files and directories in the file system.

TestParams
+ test_getxattr_none() + test_listxattr_none() + test_setxattr() + test_listxattr() + test_removexattr()

Table 12 - TestParams (xattr) test case

Kalman tests compare two versions of the Kalman filter that are used in the library. The taste case is **TestSimpleKalman** compares the results and the performances of the two.

TestSimpleKalman
+ test_getter_setter() + update_returns_state() + test_old_equal_new() + test_new_is_faster()

Table 13 - TestSimpleKalman test case

Transformation tests focus on testing the different transformations that can be performed using the library, from and to Euler, Quaternions, and Rotation matrixes.

- **TestNED**: this test case [Table 14] comprehends five tests that test the conversion of data to the NED coordinates system and vice versa.

TestNED
+ test_small_distances() + test_ecf_geodetic() + test_ned() + test_ned_saved_results() + test_ned_batch()

Table 14 - TestNED test case

- **TestOrientation**: it includes four tests [Table 15] that check the result of different transformations performed on a set of data.

TestOrientation
+ test_quat_euler() + test_rot_euler() + test_rot_quat() + test_euler_ned()

Table 15 - TestOrientation test case

3.2.5 Development and community contribution

Common folder was made a public repository on the 17th of January 2020.

common folder (17/01/2020) <George Hotz>

Even if it was made public, most of the contributions still come from the Comma.ai staff. In the case of the Kalman filter, for example, no modifications were made by the members of the community and in one year only six commits have been made, three of which are part of general modifications made to the entire *common* folder. Also, the modifications made only updated the package dependencies and the builder file to be compliant with the new standard adopted by the project, which changed over time.

Rebuild cython extensions when dependency version changes (#1886) (16/07/2020)

<Adeeb Shihadeh>

update pipfile.lock (#1896) (21/07/2020) <Willem Melching>

One of the reasons why this package is not frequently updated like the other libraries is the introduction of an ad-hoc library for Kalman filtering, called *rednose*, and over time a lot of the dependencies from the Kalman filter were removed and replaced by *rednose*. For what concerns the API package, six commits have been made after that the common folder was published for the first time. Most of the fixes were about correcting the indentation of the code, which was creating conflicts with the rules enforced by flake8 and solving some issues that could occur during the installation process on Mac and PC systems.

manager runs on Mac, and other Openpilot for PC fixes (02/02/2020) <George Hotz>

After the update of the library PyJWT, used to generate the web token, some users experience problems when trying to call the function *get_token()*, since in version 2.0 of PyJWT the method *decode()* returns a string, while in the older versions returned a type *bytes*. To solve this issue, a community member opened a pull request proposing a solution in which the token is converted into a string when the method *decode()* returns a *bytes* value, otherwise, the token is returned as it is.

<pre>token = jwt.encode(payload, self.private_key, algorithm='RS256') if isinstance(token, bytes): token = token.decode('utf8') return token</pre>
--

Fix jwt.encode return type (#19776) (#19958) (02/02/2021) <Shubham Dhumal>

The *transformation* functionalities, instead, are updated and changed as new functionalities are introduced in Openpilot, since a lot of data acquired by the cameras have to be pre-processed and calibrated before they can be used. After some initial clean-up of the deprecated functions that were no longer needed and removed, it was also adjusted to support the introduction of a new driving model.

deprecated (18/01/2020) <Harald Schafer>

delete unused code (24/04/2020) <George Hotz>
--

The BIG model, introduced in April 2020, is a model trained with a resolution close to that of the full camera frames, instead of the cropped frame that is adopted with other models. This allows to get a lot of data and driving information, but it is also more costly to run it.

What enabled the implementation of the BIG model was the release of Comma Two, which had more powerful hardware compared to EON, and the various optimizations made, especially the refactor of the Python code into C++ code which allowed to reduce the CPU usage.

who is ready for big model? (22/04/2020) <George Hotz>

Compared to the small model, the frames are four times bigger and almost the same size as the full camera frame.

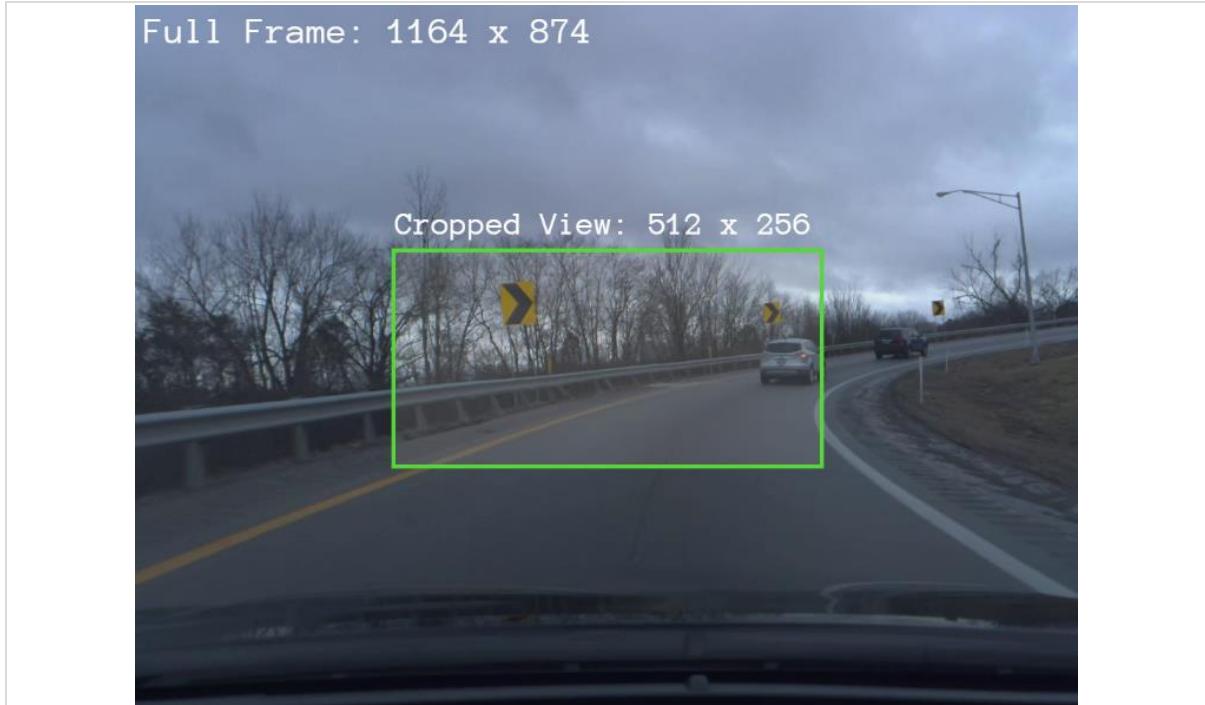


Figure 17 - Comparison between the cropped frame and the full-frame acquired by the device's camera

Other fixes, in line with the previous adjustments on the frames management, were then made.

transform_img_M (23/04/2020) <George Hotz>
run coordinate tests (24/04/2020) <Willem Melching>

```
def transform_img(base_img,
                  augment_trans=np.array([0,0,0]),
                  augment_eulers=np.array([0,0,0]),
                  from_intr=eon_intrinsics,
                  to_intr=eon_intrinsics,
                  output_size=None,
                  pretransform=None,
                  yuv=False,
                  alpha=1.0,
                  beta=0,
                  blur=0):
```

After some small improvements to the naming and the initialization of the variables, the Comma.ai team decided to refactor the code and convert it from Python to C++, to include more functionalities while keeping the same level of performance.

pure init (#1569) (27/05/2020) <Harald Schafer>

More rigorous definition of calibration (06/06/2020) <Harald Schafer>

The conversion was made by **Willem Melching**, and it took 37 commits to write the C++ classes and the different optimizations.

Write orientation & transform in C++ (#1637) (10/06/2020) <Willem Melching>

Like for the other libraries, a Cython wrapper has also been written to allow to call the generated C++ code of the transformation and orientation classes from other Python processes.

Later on, were added also the getter for the transformation matrices, which enable to transform NED data to ECEF data.

add getter for LocalCoord transformation matrices (10/06/2020) <Willem Melching>

More models were then added and the relative information about the size of their input frame had to be included in the *common* folder.

add calmodel (06/07/2020) <George Hotz>

needed in pipeline (29/07/2020) <Harald Schafer>

makes more sense (05/08/2020) <Harald Schafer>

sbigmodel, a bigmodel with the size of a smallmodel (27/10/2020) <George Hotz>

Other improvements were then made to the function and parameters relative to the camera information, making them more general and not platform-specific (they were tailored to EON in the first place).

generalize camera assumptions (#2423) (05/11/2020) <ZwX1616>

clip arcsin to prevent locationd orientation NaN (#20868) (11/05/2021) <Harold Schafer>

remove oneplus camera params (#21047) (27/05/2021) <Adeeb Shihadeh>

The common package, among the other functionalities, offers Android-specific methods and attributes that can be easily accessed by the other components of Openpilot. Especially in the beginning, the Comma.ai team focused on the maintenance and improvement of the Android functionalities included in *common*, since NEOS, the operative system of the devices on which Openpilot is executed, is based on a modified version of Android. With the first improvements made, they were introduced parameters to better monitor the status of the phone.

allow non android to be identified differently (18/01/2020) <George Hotz>

apk lib: Grant offroad access to TelephonyManager (20/01/2020) <andyh2>

Cache FW query (#1025) (31/01/2020) <Willem Melching>

Add network_type to thermal (#1030) (01/02/2020) <Andrew Valish>

Handle get_network_type exception (03/02/2020) <Willem Melching>

get_network_type: Sort, correct cell network lookup and fix for pc (04/02/2020) <Andy Haden>

Only show update alert if updater failed once since reboot (#1078) (11/02/2020) <Willem Melching>

Add LaneChangeEnabled param and settings toggle (#1093) (15/02/2020) <Andrew Valish>

Disable Power Down option for desk devices (#1117) (18/02/2020) <George Hotz>

apk: Fix permission to read /sdcard/ (20/02/2020) <andyh2>
add network strength logging to thermal (#1211) (07/03/2020) <Andrew Valish>
use unknown networkstrength, not none (#1222) (09/03/2020) <Andrew Valish>
revise wifi signalstrength dumpsys query (#1224) (10/03/2020) <Andrew Valish>
Remove legacy AccessToken param (23/03/2020) <Andy Haden>
Sidebar Connectivity Status (#1268) (05/04/2020) <Andy>
RHD support for driver monitoring (#1299) (16/04/2020) <ZwX1616>

Some changes to the structure were then made, which lead to the removal of some unused methods and files and the inclusion of a geocoder. The geocoder, then removed in later commits, offered two main functionalities: get the name of a city starting from its coordinates, and tell if in a certain city the drive is on the left lane.

Remove unused logging_es (28/01/2020) <Andy Haden>
logging: Vendor findCaller for correct stack frame info in logs (28/01/2020) <Andy Haden>
logging: imports for stack info (28/01/2020) <Andy Haden>
reverse geocoder (04/02/2020) <Harald Schafer>

Other fixes were about showing error messages and better managing the exceptions that could happen during the execution or at boot time.

dm offsetshould only care about the valid counts (23/03/2020) <ZwX1616>
URLFile: include url in bad status code error (24/03/2020) <Andy Haden>
compress option for dict column store writer (28/03/2020) <Greg Hogan>
Add binary to display text (#1301) (01/04/2020) <Willem Melching>
Boot-loop testing script for EON/C2 device sensors (#1303) (01/04/2020) <Jason Young>
Show manager startup failures using TextWindow (#1310) (03/04/2020) <Willem Melching>
fix AttributeError: 'FakeSpinner' object has no attribute 'close' (#1317) (05/04/2020) <DeanLee>
ColumnStoreWriter/Reader support for dictionary of arrays in single file (14/04/2020) <Greg Hogan>
fix UnicodeDecodeError in get_network_strength (#1385) (17/04/2020) <Dean Lee>
ColumnStoreReader support for intermediate keys of flat dictionaries (21/04/2020) <Greg Hogan>
ColumnStoreWriter.add_dict() support for multiple types (21/04/2020) <Greg Hogan>

A lot of unused code and functions, including the recently added geocode, were removed to reduce the total number of lines of the project. The company internally set the target of 40.000 lines of Python code, and the cleanup, optimizations, and porting some modules to C++ allowed it to reach this goal.

remove unused code (24/04/2020) <George Hotz>
more unused code (24/04/2020) <George Hotz>
delete more unused, now under 40k lines of python. framereader needs to be ported to C++ (24/04/2020) <George Hotz>
remove geocode (24/04/2020) <Willem Melching>

On the line of the previous changes, the Comma.ai team kept optimizing the code, but also making improvements to how the message was displayed to the final user and updating some of the adopted libraries. To fix many minor code indentation and unused import problems, it was also used <https://lgtm.com/>, which provides tools to analyze the code and prevent critical vulnerabilities.

improve printing in profiler (27/04/2020) <George Hotz>
add simple usage example to window.py (01/05/2020) <George Hotz>
Using lgtm.com and fixing found alerts (#1452) (02/05/2020) <George Hotz>

To statically analyze the code, it was adopted <https://pre-commit.ci/>, a framework for managing and maintaining multi-language pre-commit hooks. Git hooks are programs that can be placed in a hooks directory to trigger actions at a certain point during git's execution and are useful for identifying simple issues before submission to code review. pre-commit.ci run its hooks on every commit to automatically point out issues in code such as missing semicolons, trailing whitespace, and debug statements. This allows the code reviewers to focus only on what is important and not on smaller fixes that can be solved automatically.

pre-commit pylint (#1580) (28/05/2020) <Willem Melching>
flake8 in pre-commit (#1583) (28/05/2020) <Willem Melching>
Run mypy commit hook (#1591) (29/05/2020) <Willem Melching>
Running pre-commit in CI (#1590) (29/05/2020) <Willem Melching>

Since the methods and attributes in *common* can be called by all the other components of Openpilot, is important to consider unexpected behaviors that may be caused by passing the wrong parameter. That is the case of *makedirs_exists_ok(path)*, which returns the absolute path of the variable *path*. Since also URLs have the same format as a directory path and the automatic checks could not identify the error, it has been added another check that made the method return an error in the case the path starts with *http://* or *https://*.

ensure.makedirs_exists_ok is not called for URLs (09/06/2020) <Greg Hogan>
makedirs_exists_ok more specific URL detection (09/06/2020) <Greg Hogan>

Yet in another cleanup of the code, the CPU usage tests were also refactored, and they were moved from the common folder directly in the tests of the selfdrive package, and consequently, the manager_helpers.py file was removed.

fix CPU usage test for thermald and dmonitoringd (24/06/2020) <Adeeb Shihadeh>
fail CPU usage test if can't get usage for a process (24/06/2020) <Adeeb Shihadeh>
Refactor CPU usage test (#1802) (04/07/2020) <Adeeb Shihadeh>
remove duplicate logging context (04/07/2020) <Greg Hogan>
remove manager_helpers.py after CPU usage test refactor (04/07/2020) <Adeeb Shihadeh>

New common attributes and functions were then added to support the new functionalities that were being developed. In particular, two more parameters were added to the file *params.py* to get additional information about the system update, and a method checking the status of the sound card of the devices was added to the provided Android-specific methods.

support code for NEOS update alert (25/06/2020) <Adeeb Shihadeh>
Add an SSH param to disable updates (#1807) (01/07/2020) <George Hotz>
Sound test (#1820) (06/07/2020) <Adeeb Shihadeh>

A community member, **Mufeed VH**, pointed out a vulnerability, specifically an insecure temporary file creation vulnerability (CWE-377), that affected a deprecated library that was

still used. The issue was solved by substituting the vulnerable function `mktemp()` with the more secure `NamedTemporaryFile()`. [22]

Fix insecure temporary file creation (#1890) (18/07/2020) <Mufeed VH>
fix params permissions after #1890 (19/07/2020) <Adeeb Shihadeh>

Other small additions were made to consider more variables that could be relevant for Openpilot, such as the car battery capacity. It has also made explicit to the user the last exception detected when the update of Openpilot fails 15 times, and this exception is saved in a dedicated persistent parameter.

add test for params permissions (19/07/2020) <Adeeb Shihadeh>
don't init sound (31/07/2020) <Harald Schafer>
Alert when updated consistently fails (#2013) (12/08/2020) <Adeeb Shihadeh>

An important refactor of the power management system was made to solve an issue [23] opened to solve a battery drain problem of the comma device that users could experience in the case in which they would use the car for short periods daily. The Comma Two was set to be on for 30 hours and then turn off, to avoid syncing problems with Comma Connect. If used repeatedly for short periods, the Comma Two would be on for days and continuously drain the battery, with the car battery not able to charge the device completely during the short drive. Many proposals were discussed in the relative pull request [24] and a solution was found by logging the car battery level and shutting the device down if it reaches a certain threshold while the car is off.

Car power integrator + power management refactor (#1994) (17/08/2020) <robbederks>

One of the main refactor of the common package was made to overcome the hardcoded dependencies from the android platform and abstract the hardware layer, by creating different hardware classes that could be instantiated according to the different platforms on which Openpilot would be run. This implied moving the Android-specific methods and attributes into a dedicated Android class and was also created a TICI class that included all the methods and attributes relative to the new Comma Three that was being developed.

Pandad: turn on panda power (#2073) (24/08/2020) <Willem Melching>
Hardware abstraction class (#2080) (26/08/2020) <Willem Melching>
hardware.py: PC is Wi-Fi so uploader works (26/08/2020) <Willem Melching>
tici: fix set_realtime_priority (#2124) (02/09/2020) <Willem Melching>
clean up old params (04/09/2020) <Adeeb Shihadeh>
handle exception in android service call (11/09/2020) <Adeeb Shihadeh>
Tici hardware abstraction layer (#2183) (17/09/2020) <Willem Melching>
tici: reboot (15/10/2020) <Willem Melching>

Unused parameters were also removed, and other modifications were made to improve the performance and reliability, especially for the utility functions needed to retrieve the real-time information.

UI vision refactor (#2115) (04/09/2020) <Adeeb Shihadeh>
TextWindow Enhancements (#2114) (04/09/2020) <Shane Smiskol>
Improve realtime performance on NEOS (#2166) (17/09/2020) <Adeeb Shihadeh>
Realtime shield (#2194) (18/09/2020) <Adeeb Shihadeh>

Params: use a multiple-reader / single-writer flock to improve concurrency (#2207)
(24/09/2020) <Dean Lee>
Run all driving processes on cores 2-3 (#2257) (03/10/2020) <Adeeb Shihadeh>

More optimizations were also made to speed up the boot times, simplify the code, and replace unused libraries. In particular, the optimization on the permission check allowed to save 2.5 seconds when booting Openpilot.

Speedup android permissions (#2331) (13/10/2020 15:35) <Gregor Kikelj>
Params refactor, simplified (#2300) (13/10/2020 16:23) <Willem Melching>
fix params on PC and when reading path from env (#2340) (14/10/2020) <Adeeb Shihadeh>
set default path for put_nonblocking helper (14/10/2020) <Adeeb Shihadeh>
Params path only in one place (#2344) (15/10/2020) <Willem Melching>
revert apk launch thread (16/10/2020) <Adeeb Shihadeh>
fix build warnings (#2355) (17/10/2020) <Adeeb Shihadeh>
store params in ~/.comma on PC (#2369) (20/10/2020) <Willem Melching>

More changes to the structure were made and more components were moved to the common folder, so that general information about the hardware could be accessed from everywhere in the code.

Scons builder for cython extensions (#2485) (11/11/2020) <Gregor Kikelj>
Simple improvements for quality gate (#2517) (12/11/2020) <Jon Ander Oribe>
Make the DSP work everywhere (#2621) (30/11/2020) <George Hotz>
Move thermald hardware calls into HW abstraction layer (#2630) (02/12/2020) <robbederks>
hardware.py: get network info over dbus (#2727) (14/12/2020 14:19) <Willem Melching>
Set GPU priorities + improved modeld priorities (#2691) (16/12/2020) <Adeeb Shihadeh>
HW abstraction layer (#19530) (17/12/2020) <Adeeb Shihadeh>

With the introduction of the driving statistics, displayed on the main Qt UI, more parameters and utility functions were also added to the *common* folder. This included the parameters to keep track of the status of the vision toggle and the hardware details of the device, as well as parameters to store the time zone and the last time the GPS was updated, needed to show that information when no connection was available or with the car not moving.

```
b"HardwareSerial": [TxType.PERSISTENT],
b"IMEI": [TxType.PERSISTENT],
b"VisionRadarToggle": [TxType.PERSISTENT],
b"LastGPSPosition": [TxType.PERSISTENT],
b"LastUpdateTime": [TxType.PERSISTENT],
b"Timezone": [TxType.PERSISTENT]
```

Qt Offroad stats (#19498) (18/12/2020) <Gregor Kikelj>
Mpc rework2 (#19660) (15/01/2021) <Harald Schafer>
Qt offroad: pairing (#19675) (18/01/2021) <Gregor Kikelj>
params helpers (#19788) (19/01/2021) <Greg Hogan>
vision-only radar toggle (#19849) (27/01/2021) <Greg Hogan>

Timezoned (#19960) (06/02/2021) <Willem Melching>

fixup ui (#20049) (10/02/2021) <Harald Schafer>

Spinner: wait for UI to start (#20279) (09/03/2021) <Shane Smiskol>

The development of new features required also more parameters to be stored, to save the state of the new relevant information needed. In particular, with the possibility to force power down the device and to SSH the operative system, a parameter telling if any of these actions are allowed to be performed is needed. The same is true for the introduction of the possibility to lock and unlock the car remotely, while the toggle to change the lane while driving was removed, together with the corresponding parameter.

```
b"SshEnabled": [TxType.PERSISTENT],  
b"ForcePowerDown": [TxType.CLEAR_ON_MANAGER_START],  
b"RecordFrontLock": [TxType.PERSISTENT], # for the internal fleet
```

agnos 0.6 (#20077) (19/02/2021 03:14) <Adeeb Shihadeh>

Fix tici powerdown and add support for forcing (#20132) (23/02/2021) <robbederks>

record front lock (#20400) (19/03/2021 04:15) <Adeeb Shihadeh>

Remove lane change toggle and default LDW to off (#20442) (23/03/2021) <Adeeb Shihadeh>

In release 0.8.3 it was introduced a new model to support the laneless drive, the *KALE* (or *KL*) model. The laneless drive is not the default mode of drive, so it has been added a toggle to turn this mode on. The corresponding parameter can be accessed through the key “*EndToEndToggle*”.

```
b"EndToEndToggle": [TxType.PERSISTENT]
```

New KL model + laneless toggle (#20454) (24/03/2021) <Harold Schafer>

To further increase the performances, as well as the maintainability of the project, caching and logging have been introduced to different modules of *selfdrive*, and to support them, different parameters to keep stored the state of the different components have been introduced. To also increase usability, it has been introduced a series of toggles to enable the wide camera and the LTE connection, and the relative parameter to keep track of the user actions have been introduced as well.

```
b"GitDiff": [TxType.PERSISTENT],  
b"GithubUsername": [TxType.PERSISTENT],  
b"ApiCache_DriveStats": [TxType.PERSISTENT],  
b"ApiCache_Device": [TxType.PERSISTENT],  
b"ApiCache_Owner": [TxType.PERSISTENT],  
b"EnableWideCamera": [TxType.PERSISTENT],  
b"EnableLteOnroad": [TxType.PERSISTENT],  
b"ControlsReady": [TxType.CLEAR_ON_MANAGER_START,  
TxType.CLEAR_ON_PANDA_DISCONNECT]
```

log to file and send through athena (#20250) (25/03/2021) <Greg Hogan>

updated: log git diff on overlay init (#20476) (26/03/2021) <Adeeb Shihadeh>

Qt: cache home screen state (#20395) (26/03/2021) <Dean Lee>

Cache prime/points widget (#20497) (26/03/2021) <Willem Melching>

logging cleanup (#20502) (27/03/2021) <Greg Hogan>
Qt: show username for current SSH keys (#20508) (29/03/2021) <Dean Lee>
Params: python-like interface (#20506) (30/03/2021) <Dean Lee>
add putBool/getBool wrappers to cython params class (#20611) (07/04/2021) <Willem Melching>
ecam toggle (#20597) (07/04/2021) <Willem Melching>
Params: new class FileLock (#20636) (09/04/2021) <Willem Melching>
LTE toggle (#20683) (15/04/2021) <Adeeb Shihadeh>
delay controls start (#20761) (30/04/2021) <Adeeb Shihadeh>

To grant quick access to the common parameters keys list, it has been ported from Python to C++ and moved from the *common* folder to *selfdrive/common/*. In this way, the Python files only provide utility function caller more rarely, while the parameters key can be accessed and managed using quicker C++ methods.

Params: move keys from cython to cc (#20814) (04/05/2021) <Dean Lee>

After that, the parameter type indicating that a parameter should be reset after the car engine is started was added.

The *enum* ParameterKeyType, in which are enumerated all the parameters type available, after the refactor and the last additions become the following.

```
enum ParamKeyType {
    PERSISTENT = 0x02,
    CLEAR_ON_MANAGER_START = 0x04,
    CLEAR_ON_PANDA_DISCONNECT = 0x08,
    CLEAR_ON_IGNITION_ON = 0x10,
    CLEAR_ON_IGNITION_OFF = 0x20,
    ALL = 0x02 | 0x04 | 0x08 | 0x10 | 0x20
};
```

add CLEAR_ON_IGNITION param type (#20810) (04/05/2021) <Adeeb Shihadeh>
Add CLEAR_ON_IGNITION_OFF param type (#21121) (04/06/2021) <Shane Smiskol>

Refactoring the main methods to manipulate the parameters and translating them in C++ allowed to leverage the threaded execution, since Cython code can release the *global interpreter lock* (GIL) if no Python objects are manipulated by the specific fragment of code where the GIL has been released. From the official documentation, we read that GIL is the mechanism that assures that only one thread executes Python bytecode at a time. By releasing it, the performances can be increased by leveraging multi-threading.

Release GIL when calling C++ params functions (#21257) (15/06/2021) <Willem Melching>
--

After the last clean-up of the code and a fix to the log of the error, version 0.8.7 of Openpilot was officially released.

Add type hints, small cleanups (#21080) (03/06/2021) <Josh Smith>
only log errors once (15/07/2021) <Greg Hogan>

3.3 Laika

Laika is an open-source library for processing GNSS. The Global Navigation Satellite System (GNSS) refers to the set of constellations of satellites that include Europe's **Galileo**, the USA's **NAVSTAR Global Positioning System (GPS)**, Russia's **Global'naya Navigatsionnaya Sputnikovaya Sistema (GLONASS)**, and China's **BeiDou Navigation Satellite System**. The different constellations of satellites provide signals from space that include positioning and timing data and the GNSS receivers use all these data to determine the exact location.

Laika can process raw GNSS observations with data gathered online from various analysis groups to produce data ready for position/velocity estimation, producing accurate results, readable and easy to use. One of the possible methodologies that can be adopted to determine the position of a GNSS receiver is that of the Time-Of-Arrival. The different satellites have known orbits, so it is possible to determine their positions at any time. Since the signals travel at the speed of sound, by measuring the time that passes from when the signal is sent to when is received is possible to calculate the distance of the receiver from the satellite (3.12).

$$R = \text{range}$$

$$R = v_{\text{sound}} * \Delta t \quad v_{\text{sound}} = \text{velocity of sound} \quad (3.12)$$

$$\Delta t = \text{transmit time} - \text{receive time}$$

This makes determining the receiver's position a basic 3-dimensional trilateration problem. In practice, observed distances to each satellite will be measured with some offset that is caused by the receiver's clock error. This offset also needs to be determined, making it a 4-dimensional trilateration problem.

Laika is designed to process the received data and transform the raw GNSS data into usable distance measurements and satellite positions ready to be used to estimate the position. This is possible thanks to the class AstroDog, which is the main component of Laika.

Running the Jupyter Notebook file "walkthrough.ipynb" gives some important insight on what operations this library allows to perform, which include retrieving different pieces of information about the satellites such as their accurate position, velocity, and clock error.

```
Sattelite's position in ecef(m):  
[-19501731.52999999; -7580440.43899999; 16817823.82499999]  
Sattelite's velocity in ecef(m/s):  
[1993.27528306; -613.07074562; 2016.32438942]  
Sattelite's clock error(s): 0.00025521380390669117  
Sattelite's delay correction (m) in San Fransisco: 3.1049571045513886
```

Laika supports different constellations of satellites, and is possible to get their orbits' information that is publicly available to download. The files containing this information are broadcasted by Crustal Dynamics Data Information System (CDDIS), the official NASA's archive of Space Geodesy Data, and can be downloaded at <https://cddis.nasa.gov>. The official format for exchanging raw satellite navigation system data I known as Receiver Independent Exchange Format (RINEX).

RINEX enables storage of measurements of pseudo-range, carrier-phase, Doppler, and signal-to-noise. After parsing these data, is possible to extract all this information and use them for many purposes, such as plotting them to show the actual orbits of the satellites in the constellation of interest.

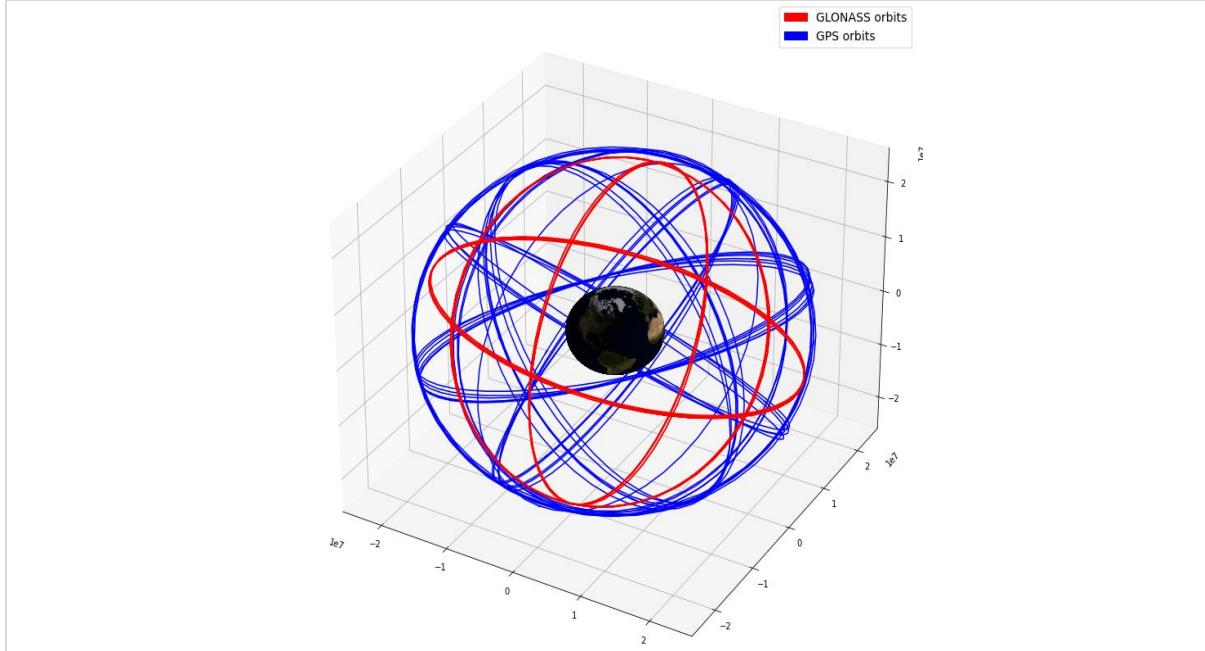


Figure 18 - Visualization of GLONASS and GPS orbits

Is also possible to track the number of satellites that are in view for each time frame (or epoch) and to calculate their distance from the receiver, so that is possible to plot the exact position of each satellite and position it precisely in their orbits.

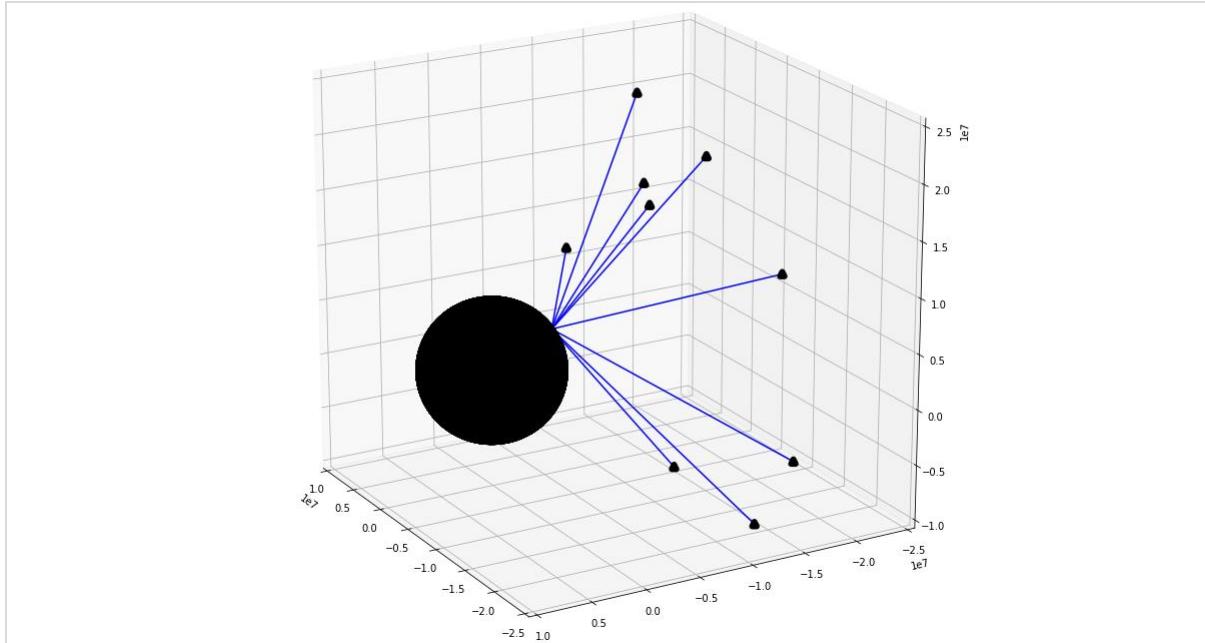


Figure 19 - Receiver on Earth and the estimation of the position in space of multiple satellites

AstroDog can also make predictions on positioning measurement based on the previous measurement. The predictions are not always precise, but still, they are close to the actual measurement.

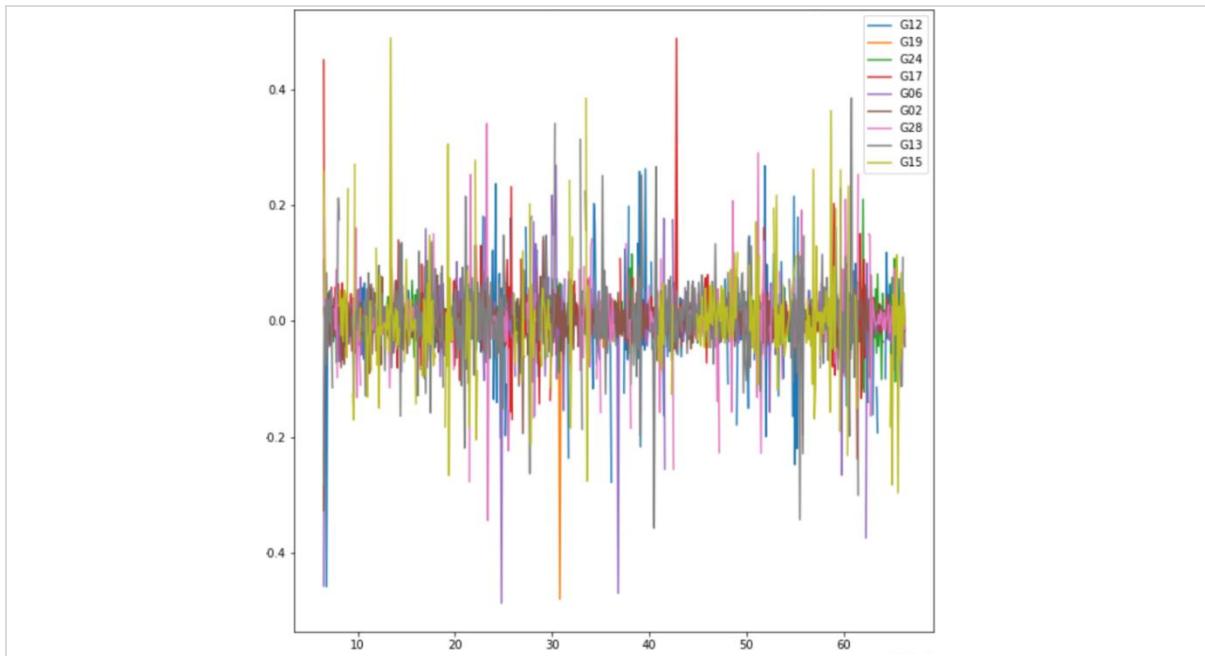


Figure 20 - Predictions of the Carrier Phase (CP) made by AstroDog

3.3.1 Package structure

The main components providing the Laika library functionalities are included in the `/laika` directory, but the repository also comes with tests and examples which showcase the library features and performances.

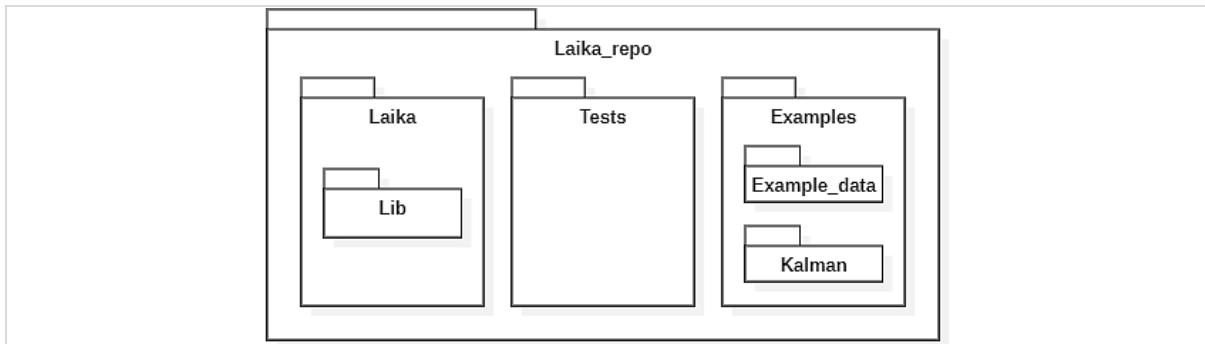


Figure 21 - Package diagram of Laika

Laika library is written in Python and has a strong focus on readability, usability, and easy integration with other optimizers. It runs in Python 3.8.2 and has only been tested on Ubuntu 20.04.

Language	Files	Blank	Comment	Code
Python	29	743	452	3556
HTML	1	9	0	1823
Jupyter Notebook	3	0	1029	329
Markdown	1	31	0	59
YAML	2	2	0	54
SUM:	36	785	1481	5821

Table 16 - Lines of code of the Laika, by programming language

3.3.2 Implementation

More in detail, the AstroDog class provides methods and attributes to retrieve, store and manage different types of localization data and from the different satellite constellations.

AstroDog
<ul style="list-style-type: none"> + auto_update: bool + cache_dir: String + dgps: bool + dgps_delays: List + ionex_maps: List + pull_orbit: bool + valid_const: List<String> + cached_ionex: List<Any> + cached_dgps: List<Any> + orbit_fetched_times: TimeRangeHolder + nav_fetched_times: TimeRangeHolder + dcbs_fetched_times: TimeRangeHolder + orbits + nav + dcbs + cached_orbits + cached_nav + cached_dcbs
<ul style="list-style-type: none"> + get_ionex(time) + get_nav(prn, time) + _select_valid_temporal_items(item_dict, time, cache) + get_navs(time) + get_orbit(prn, time) + get_orbits(time) + get_dcb(prn, time) + get_dgps_corrections(time, recv_pos) + add_ephem(new_ephem, ephems) + get_nav_data(time) + get_orbit_data(time) + get_dcb_data(time) + get_ionex_data(time) + get_dgps_data(time, recv_pos) + get_tgd_from_nav(prn, time) + get_sat_info(prn, time) + get_all_sat_info(time) + get_glonass_channel(prn, time) + get_frequency(prn, time, signal) + get_delay(prn, time, rcv_pos, no_dgps, signal, freq)

Table 17 - AstroDog class

The methods provided by AstroDog and their purpose will be analyzed in the following pages, giving also details on the different types of data.

- **get_ionex(time)**: the Ionosphere map Exchange (IONEX) is a format adopted to communicate effectively the total electron content (TEC) of the ionosphere. These data are stored in IONEX files downloadable from <https://cddis.nasa.gov/archive/gnss/products/ionex/> and composed of a header and a data section. IONEX supports the exchange of 2- and 3-dimensional TEC maps given a certain geographic grid. The method `get_ionex()` returns a list of objects that have a time that is closest to a given time (operation performed by the method `get_closest()` in `\laika\helpers.py`). The set of data considered is retrieved by the method `get_ionex_data()`, which downloads the IONEX map and returns the file path where it is stored. Data are then parsed and the head and the body of the file are extracted.
- **get_nav(prn, time)**: all the satellites in the different constellations transmit information about their location and timing via what is known as ephemeris data. This data is used by the receivers to estimate their position on earth and is retrieved by the method `get_nav(prn, time)`. The pseudo-random noise (`prn`) is the satellite ID in the RINEX convention, while the time is the epoch for which the data has to be retrieved. The method `get_nav_data(time)` retrieves the Ephemeris data for the desired constellation of satellites and parses them to generate a corresponding Ephemeris object (GLONASSEphemeris or GPSEphemeris), which easily allow access to the satellites key information. The Ephemeris is then checked and only the one with a time that is closest to a given time is cached. In this way it is possible to access the cached ephemeris knowing that it will be the most recent available. According to the time at which the request for the Ephemeris is made, data on a daily or an hourly basis may be available. The CDDIS creates daily broadcast ephemeris files from the GNSS site-specific files transmitted by the stations and for both GPS and GLONASS data they can be downloaded at the URL <https://cddis.nasa.gov/archive/gnss/data/daily/>. A similar file is created at the start of the UTC day and updated on an hourly basis from the hourly broadcast navigation files and these files are available at <https://cddis.nasa.gov/archive/gnss/data/hourly/>.
- **add_ephem(new_ephem, ephem)**: add the `new_ephem` object, recorded by a satellite with a certain `prn`, to the map of Ephemeris already stored. At each key of the map, equal to the `prn`, will correspond an array of Ephemeris objects acquired by the satellite with that same `prn`.
- **_select_valid_temporal_items(item_dict, time, cache)**: it returns only valid temporal item for a specific time from currently fetched data, by verifying that the absolute value of the difference between the time of the measure and that of the Ephem data is less than a maximum difference.
- **get_navs(time)**: uses the method `_select_valid_temporal_items(item_dict, time, cache)` to validate the cached navigation data for a specific time. If the considered time does not have a corresponding Ephemeris file, the method calls `get_nav_data(time)` to get a valid Ephemeris file that corresponds to that time.
- **get_orbit (prn, time)**: satellite orbit solutions are made available using pre-determined schedules, e.g., sub-daily, daily, or weekly, depending upon the data product. All orbit solution files utilize the Extended Standard Product- 3 (SP3c) format. All operational IGS GNSS products (i.e., orbits, station positions, EOP, clock solutions) are available in subdirectories, divided by GPS week, and are retrieved by the method `download_orbits()`, which downloads the file corresponding to the desired GPSTime. Files can be accessed via <https://cddis.nasa.gov/archive/gnss/products/> for GPS and <ftp://ftp.glonass-iac.ru/MCC/PRODUCTS/> for GLONASS. The files are then parsed by the method `parse_sp3_orbits()`, which generates a PolyEphemeris object holding the navigation information for easy access. The newest ephemeris is then cached.

- **get_orbits(*time*)**: uses the method `_select_valid_temporal_items(item_dict, time, cache)` to validate the cached navigation data for a specific time. If the considered time does not have a corresponding Ephemeris file, the method calls `get_orbits_data(time)` to get a valid Ephemeris file that corresponds to that time.
- **get_dcb(*prn*, *time*)**: Differential Code Biases (DCBs) are the systematic errors, or biases, between two GNSS code observations at the same or different frequencies. DCBs are required for code-based positioning of GNSS receivers, extracting ionosphere total electron content (TEC), and other applications. The method `get_dcb(prn, time)` retrieves the DCB of the satellite that has the ID equals to the *prn* and it caches the result in a map containing all the most recent DCB for each prn. If no DCB is already cached the method `get_dcb_data` is invoked to initialize the DCBs map.
- **get_dcb_data(*time*)**: the method tries to download the DCB files from the repository at <https://cddis.nasa.gov/archive/gnss/products/bias/> for the time specified and if the file is available is downloaded and cached, then the file path of the file is returned. Data is retrieved for many days since a lot of data are missing in the database.
- **get_tgd_from_nav(*prn*, *time*)** the term T_{GD} corresponds to the group delay differential. TGD values are referenced to an empirical absolute instrumental (satellite) bias, whereas DCB values are in a relative sense to reflect the differential hardware (the satellite or receiver) delay between two different code observations obtained on the same or two different frequencies. T_{GD} values can be easily accessed through the method `get_tgd_from_nav()`, which returns the *tgd* parameter of the GPSEphemeris object with the specified *prn*.
- **get_sat_info(*prn*, *time*)**: it allows to retrieve information recorded by a satellite, with the identifier equal to the specified *prn* and recorded at a particular *time*. Information includes the position, velocity, and clock error.
- **get_all_sat_info(*time*)**: it fetches the information recorded by all the satellites of the constellation at the specified time and it creates a map, with the key equals to the satellite's *prn* and the value equal to an array containing the satellite's information (position, velocity, and clock error), for all the satellites available.
- **get_dgps_data(*time*, *recv_pos*)**: A Differential Global Positioning System (DGPS) is an enhancement to the Global Positioning System (GPS) which provides improved location accuracy, in the range of operations of each system, from the 15-meter nominal GPS accuracy to about 1-3 centimeters in case of the best implementations. These data are acquired by the network of CORS stations. The Continuously Operating Reference Station (CORS) network is a multi-purpose cooperative endeavor involving more than 220 government, academic, commercial, and private organizations. Although participation in the CORS network is voluntary site operators must adhere to certain basic standards and conventions. CORS sites have a fundamental role in establishing and giving access to the National Spatial Reference System (NSRS). This network aims to minimize GNSS signal distortion and maximize the quality of calculated positions, following models used in processing GNSS data, to obtain centimeter to sub-centimeter accuracy. The list of available CORS is retrieved and the set of stations that are closest to the specified position *recv_pos* is then identified. For each station, the data is downloaded from <ftp://geodesy.noaa.gov/cors/rinex/> and cached in RINEX format. If such file exists, is parsed and a DGPSDelay object is generated and appended to the array of available objects. A DGPSDelay object contains information on the station id, station position, transmission delays, and maximum distance from the receiver.
- **get_dgps_corrections(*time*, *recv_pos*)**: it returns the cached DGPSDelay object that is the closest to the given time, if available, otherwise it fetches the DGPS data through the

method `get_dgps_data()` and caches the one that has the time that is closest to the specified time.

- **get_delay(prn, time, rcv_pos, no_dgps, signal, freq):** it returns the transmission delay of the specified satellite. The satellite information is first retrieved using the method `get_sat_info()` and if these data are available the Elevation and Azimuth for the satellite's position are calculated. The Elevation describes the angle of a satellite relative to the horizontal plane, while the Azimuth is the angle between the satellite and true North. If the Elevation is less than 18° the function terminates, otherwise, it calculates the delay according to the type of satellite. In the case of a CORS satellite, the delay is calculated through `get_dgps_corrections()`, while in the other cases the method calculates the ionospheric and tropospheric delays, also making corrections using the differential code biases. In the ionosphere the speed of propagation of radio signals depends on the number of free electrons in the path of a signal, causing the delay, while in the troposphere the speed of propagation of GPS signals in the troposphere is lower than that in free space and, therefore, the apparent range to a satellite appears longer, typically by 2.5-25 m depending on the satellite elevation angle.
- **get_glonass_channel(prn, time):** All GLONASS satellites transmit the same code as their standard-precision signal; however, each transmits on a different frequency using a 15-channel frequency division multiple access (FDMA) technique spanning either side from 1602.0 MHz, known as the L1 band. The method `get_glonass_channel()` returns the channel on which the satellite with the specified *prn* is transmitted at a given *time*.
- **get_frequency(prn, time, signal):** Each constellation of satellites transmits signals on very precise frequencies. GPS satellite program started to transmit right-hand circularly polarized signals to the earth at two frequencies, designated L1 (at 1575.42MHz) and L2 (at 1227.6 MHz). Moreover, a new radio frequency link (L5 at 1176.45 MHz) for civilian users has been included in a radio band reserved exclusively for aviation safety services. Unlike GPS and the other GNSS, GLONASS uses Frequency Division Multiple Access (FDMA) rather than Code Division Multiple Access (CDMA) for its legacy signals transmission. The open signals are located at 1602 MHz (L1 band) and 1246 MHz (L2 band) for civil uses. A modernized GLONASS-K satellite, GLONASS-KM also transmits on the L5 frequency at 1176.45 MHz, the same as the modernized GPS signal L5 and Galileo signal E5a. Galileo satellites transmit permanently three independent CDMA and Right-Hand Circularly Polarised (RHCP) signals, named E1, E5, and E6. The E5 signal is further sub-divided into signals denoted E5a and E5b. The precise frequency at which a satellite transmits can be identified by the method `get_frequency()` by getting the constellation to which the satellite belongs, calculated using the *prn*, and the code corresponding to the transmission band.

3.3.3 Usage

Laika library is used to acquire and process GNSS data from the satellites and determine a set of information including the satellite's position, velocity, and frequency.

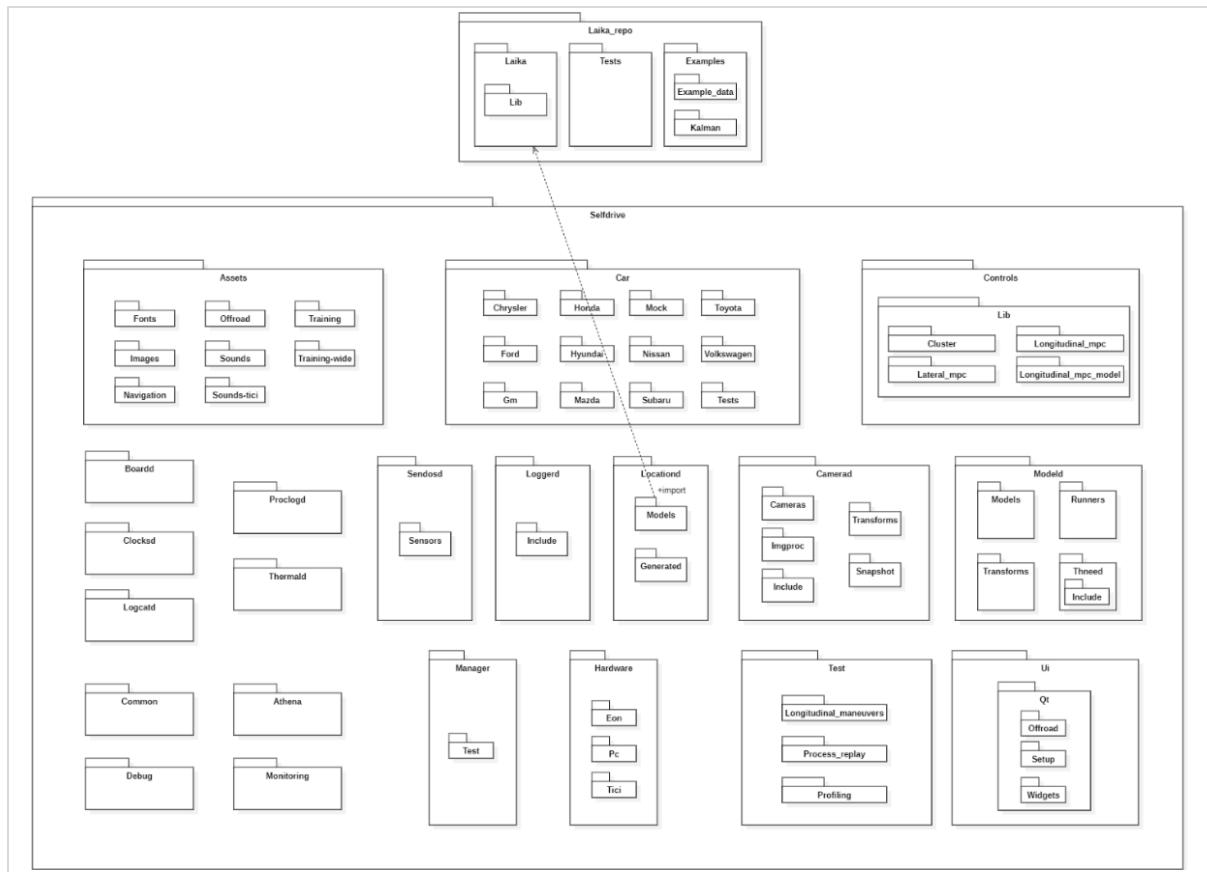


Figure 22 - Import dependencies between selfdrive and laika

The results obtained using the data processed by Laika are much more precise than those produced using raw GNSS data acquired by the u-blox M8 GNSS module.

The process *locationd* combines GNSS, INS, and vision data to precisely estimate the vehicle positioning in space. Through an Extended Kalman Filter, which is provided by the Rednose library, these data are smoothed and used by Openpilot. Rednose allows using many measurements models, which in the case of Laika is based on processed GNSS acquisitions, and combines them with the other measurements model to let the Extended Kalman Filter consider all the possible sources of noise of the system.

```
bs_eqs = [[h_speed_sym, ObservationKind.ODOMETRIC_SPEED, None],  
          [h_gyro_sym, ObservationKind.PHONE_GYRO, None],  
          [h_phone_rot_sym, ObservationKind.NO_ROT, None],  
          [h_acc_sym, ObservationKind.PHONE_ACCEL, None],  
          [h_pos_sym, ObservationKind.ECEF_POS, None],  
          [h_vel_sym, ObservationKind.ECEF_VEL, None],  
          [h_orientation_sym, ObservationKind.ECEF_ORIENTATION_FROM_GPS, None],  
          [h_relative_motion, ObservationKind.CAMERA_ODO_TRANSLATION, None],  
          [h_phone_rot_sym, ObservationKind.CAMERA_ODO_ROTATION, None],  
          [h_imu_frame_sym, ObservationKind.IMU_FRAME, None]]
```

GNSS processing requires getting data from the internet from various analysis groups such as NASA's CDDIS. AstroDog downloads and caches files from FTP servers from these groups when it needs them. These files are then parsed by AstroDog and kept in memory. Every one of these parsed objects (DCBs, ionospheric models, satellite orbit polynomials, etc.) has a valid location area and/or a valid time window. Within those windows, these objects can provide information relevant to GNSS processing.

The following activity diagrams show how some of the main methods provided by the library communicate with the different components. The methods considered are `get_ionex()` [Figure 23], `get_nav()` [Figure 24], and `get_orbit()` [Figure 25].

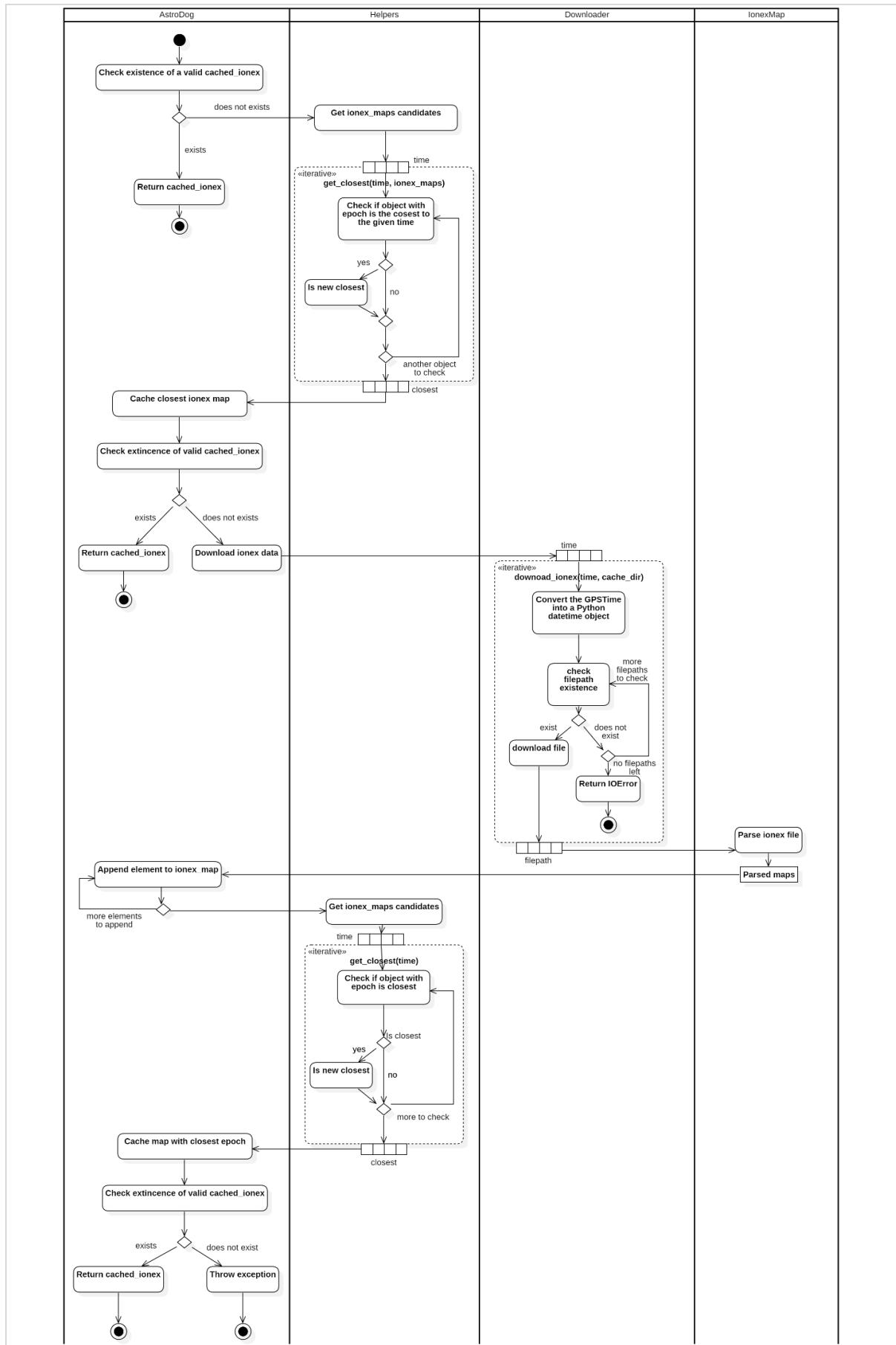


Figure 23 - `get_ionex()` activity diagram showing download, elaboration, and caching of IONEX files

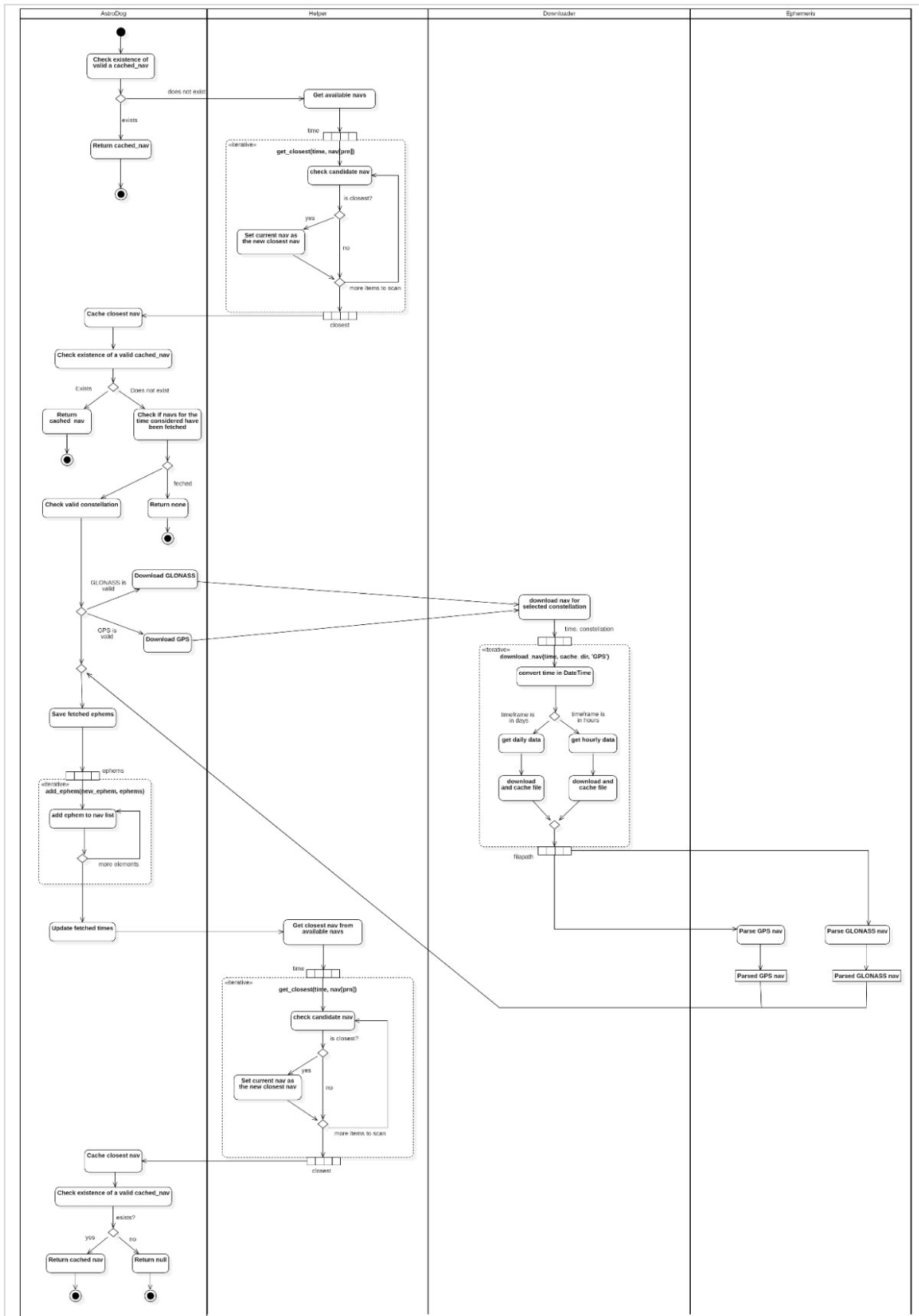


Figure 24 - `get_nav()` activity diagram showing download, elaboration, and caching of NAV files

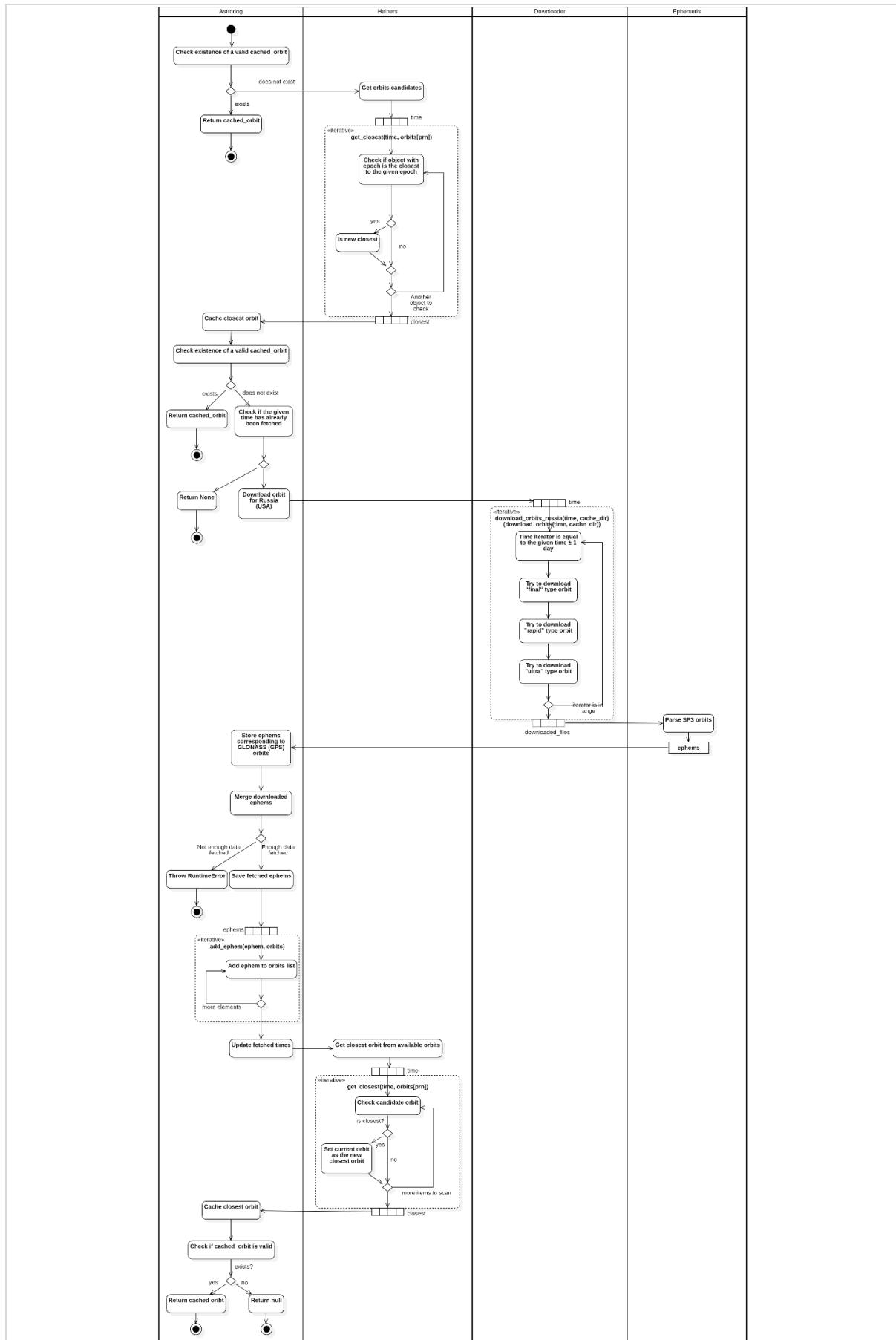


Figure 25 - `get_orbit()` activity diagram showing download, elaboration, and caching of ORBIT files

3.3.4 Testing

The functionalities of *laika* are tested through unit testing, using the *unittest* framework. The test cases focus on verifying the different aspects of the process of retrieving data from a satellite and checking their validity.

- **TestDOP:** get raw data from the satellite and verify their **dilution of precision** (DOP). DOP describes to what extent the errors in the measurement will affect the final state estimation and is defined as $GDOP = \Delta(\text{Output location})/\Delta(\text{Measured data})$ [25]. The tests verify the different measurements that compose the dilution of precision and check that the DOP stays close to a given threshold. The different types of DOP are:
 - HDOP – horizontal dilution of precision
 - VDOP – vertical dilution of precision
 - PDOP – position (3D) dilution of precision
 - TDOP – time dilution of precision
 - GDOP – geometric dilution of precision

TestDOP	
+	satellites
+	receiver
+	setUp()
+	test_GDOP()
+	test_HDOP()
+	test_VDOP()
+	test_PDOP()
+	test_TDOP()

Table 18 - TestDOP Test Case

- **TestAstroDog:** AstroDog is the main component of this library and these tests check that the different methodologies that can be used to retrieve satellites information (such as Ephemeris and Orbits) return close enough results.

TestAstroDog	
+	gps_times_list
+	svIds
+	gps_times
+	test_nav_vs_orbit_old()

Table 19 - TestAstroDog Test Case

- **TestFailCache:** this test tries to retrieve satellite data 1000 times and checks that it takes less than 10 seconds.

TestFailCache	
+	gps_times_list
+	svIds
+	gps_times
+	test_no_infinite_pulls()

Table 20 - TestFailCache Test Case

- **TestFetchSatInfo:** tests that AstroDog can retrieve information using all the different types of constellations and methodologies available. It does that by defining an array of all the possible combinations of the constellation and pull methodologies and requests for the satellites' data, expecting a not empty result.

TestFetchSatInfo
+ test_fetch_data_from_distant_future() + test_no_block_satellite_when_get_info_from_not_available_period() + test_get_all_sat_info_gps()

Table 21 - TestFetchSatInfo Test Case

- **TestPositioning:** it downloads and caches the satellite positioning information from the official website and computes its exact position, comparing actual data with the result of the computation of the same data using a WSL optimizer.

TestPositioning
+ test_station_position()

Table 22 - TestPositioning Test Case

- **TestConstellationPRN:** Check the validity of the unique identifier of the satellites for the different constellations (PRN). From the PRN should be possible to get all the information of the satellite, such as the constellation to which it belongs, and the tests in this test case try to retrieve this information for both valid and invalid PRNs.

TestConstellationPRN
+ test_constellation_from_valid_prn() + test_constellation_from_prn_with_invalid_identifier() + test_constellation_from_prn_outside_range() + test_prn_from_nmea_id_for_main_constellations() + test_prn_from_nmea_id_for_SBAS() + test_prn_from_invalid_nmea_id() + test_nmea_id_from_prn_for_main_constellations() + test_nmea_id_from_prn_for_SBAS() + test_nmea_id_from_invalid_prn() + test_nmea_ranges_are_valid()

Table 23 - TestConstellationPRN Test Case

- **TestTime:** performs different verifications on the synchronization of the satellites' time and on the methods needed to convert and compare the system time with the satellites' time. The checks performed try to verify that a given time is in a certain time range and that the methods used to manipulate the time data and the time ranges return the expected results, by comparing the result of their computation with the expected outcome of the computation.

TestTime
+ datetimes_strings
+ datetimes
+ gps_times_list
+ gps_times
+ test_gps_time_dt_conversion()
+ test_gps_time_week_rollover()
+ test_gps_time_subtraction_addition()
+ test_syncer()
+ test_utc_converter()

Table 24 - TestTime Test Case

- **TestTimeRangeHolder:** retrieves ranges of time from the satellite and verifies different properties, like the possibility to merge more ranges of time, to check if a given time is inside or outside of a given range and if a range is empty.

TestTimeRangeHolder
+ test_empty()
+ test_one_in_range()
+ test_one_outside_range()
+ test_merge_ranges()
+ test_extend_range_left()
+ test_extend_range_right()

Table 25 - TestTimeRangeHolder Test Case

To prove the efficacy of Laika, the library was tested using comma2k19, a large dataset ideal for the development and validation of tightly coupled GNSS algorithms and mapping algorithms that work with commodity sensors.

A lack of ground truth can make it difficult to judge GNSS algorithms since the true position of the receiver is never known. However, assuming the height of the road is constant within a small area, it can be estimated the altitude accuracy of a position fix by checking the variation of the estimated road height over small sections (5m x 5m) of road. This requires many passes through the same section of road to be reliable but the high-density data from comma2k19 is more than sufficient.

The conducted tests compare the measurement estimated using Laika algorithm and the ones computed using *Live u-blox baseline algorithm* and as it can be seen from Figure 26 the results show that overall, Laika can reduce the positioning error by 40%.

The measurement has been made with the antenna in two different positions, in one case on the roof of the vehicle and in the other case under the windshield, where the device is supposed to be when using Openpilot. The following bar charts show the error dispersion in the two cases when using Laika and when using u-blox live.

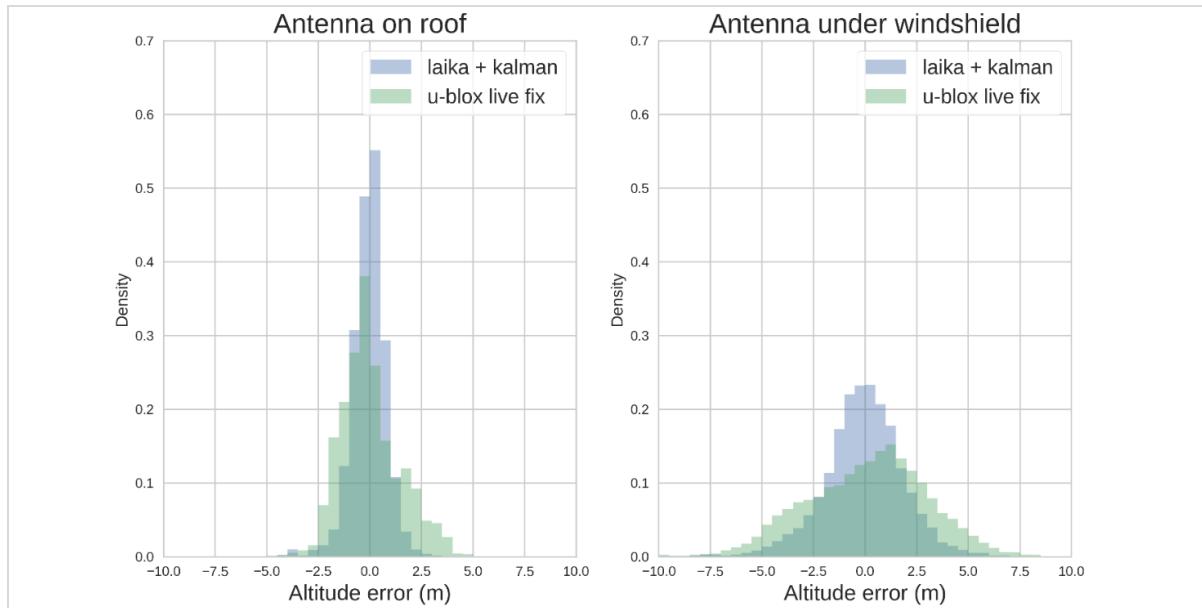


Figure 26 - Comparison of the measurements acquired with Laika and U-Blox. Laika shows a much higher concentration of data around 0 meters of altitude error

3.3.5 Development and community contribution

Laika is an open-source GNSS processing library developed with comma2k19. comma2k19 is a dataset of over 33 hours of commute in California's 280 highway. This means 2019 segments, 1 minute long each, on a 20km section of highway driving between California's San Jose and San Francisco.

comma2k19 is a fully reproducible and scalable dataset. The data was collected using comma EONs equipped with sensors similar to those of any modern smartphone, including a road-facing camera, phone GPS, thermometers, and 9-axis inertial measurement units (IMU). Additionally, the EON captures raw GNSS measurements, and all CAN data sent by the car with a comma grey panda.

The comma2k19 project was possible thanks to **Eddie Samuels, Nicholas McCoy, George Hotz, Greg Hogan, Viviane Ford, and Willem Melching**, who set up the hardware and infrastructure that enabled this research, while most of the work on the Laika library was made by **Harald Schafer**, the CTO of Comma.ai, who has made more than 100 commits to the repository.

Laika is a public repository on GitHub since the 20th of November 2018 and today counts almost 200 commits, more than half of which has been made by Harald Schafer. Of these commits, the ones that directly affect the code or the repository structure are around 80 commits, while the others are small fixes to the documentation, typos, and comments.

To better understand how the library was modified over time, we can distinguish between two classes of modifications: the ones that directly affected the functions of the library and the ones that improved the documentation and the test provided with the library.

For the second category of modifications, we will look at the commits that modified the files contained in the folders *laika_repo/examples* and *laika_repo/tests*, which contains respectively a series of Jupiter Notebooks which showcase the functioning of the library, also when combined with a Kalman filter, and a series of *unittests* to validate the library. At

the first publication of the repository the library was pretty simple, and so were the tests and the examples included.

Many tests were left commented, waiting for the implementation of the needed methods.

laika tests (29/11/2018) <Harald Schafer>
change test positions (29/11/2018) <Harald Schafer>

The following commits, instead, focused on the development of the Jupiter Notebooks which showcased the accuracy of Laika and compared the result of the computation of raw data using Laika and that using u-blox, together with some fixes on the import and the location of the tests files after the refactoring of the whole library structure.

add Kalman example (21/11/2018) <Harald Schafer>
run test using setup.py test (26/11/2018) <Willem Melching>
cleaned a little (27/11/2018) <Harald Schafer>
packaging fixes (29/11/2018) <Harald Schafer>
update notebooks (29/11/2018) <Harald Schafer>
add cache for speed (29/11/2018) <Harald Schafer>
final updates notebooks (29/11/2018) <Harald Schafer>
updated plots (29/11/2018) <Harald Schafer>

At this point, the repository includes three Jupiter Notebooks, *Compute_station_pos.ipynb*, *Kalman.ipynb*, and *Walkthrogh.ipynb*. They respectively download some observation data from a CORS station and confirm the position estimate to the known position of the station, show the difference between fixes computed with Laika from raw data of the u-blox receiver versus the fixes the u-blox receiver computes, and test the functioning of AstroDog, the main component of Laika. The Jupiter Notebooks were then further enriched by taking as input data the one collected in Comma.ai's database comma2k19.

use comma2k19 (12/12/2018) <Harald Schafer>
--

This required fixing some problems that the introduction of comma2k19 produced, and the fixes were published that same day.

fix last Kalman bugs (12/12/2018) <Harald Schafer>
another update..... (12/12/2018) <Harald Schafer>
pushed wrong... (12/12/2018) <Harald Schafer>

For more than 9 months the examples and the tests remained unchanged, until when Python3 was adopted as the standard for the project and some adjustments had to be made to be compliant with it. Most of the fixes regarded the syntax of the code. No other changes were made to the examples and tests in 2019.

python 3 stuff (27/09/2019) <Harald Schafer>
fix prints (27/09/2019) <Harald Schafer>

A new TestCase was introduced after the publication of the repository at the beginning of 2020. The TestPositioning test case has the same behavior as the Jupiter Notebook *Compute_station_pos.ipynb*, but in a form of a unittest.

Created test_positioning.py, a modification of Compute_station_pos.ipynb to test new code. Behavior should be the same, however differential code bias has not been
--

updated for new signals and returns 0 for new signals. Implemented warning to notify user. (18/01/2020) <Logan Maynard>
make unit test (23/01/2020) <Harald Schafer>
fix examples path (28/01/2020) <Willem Melching>

More tests and fixes were then added by a community member, **Slawomir Figiel**. In particular, the added unit test checks the return value of the methods which calculate the Dilution of Precision (DOP) parameters.

The pull request on GitHub was reviewed and merged with the master branch.

```
def test_GDOP():
    dop = get_DOP(self.receiver, self.satellites)
    self.assertAlmostEqual(dop, 2.352329857908973)

def test_HDOP():
    dop = get_HDOP(self.receiver, self.satellites)
    self.assertAlmostEqual(dop, 1.3418910470124197)

def test_VDOP():
    dop = get_VDOP(self.receiver, self.satellites)
    self.assertAlmostEqual(dop, 1.7378390525714509)

def test_PDOP():
    dop = get_PDOP(self.receiver, self.satellites)
    self.assertAlmostEqual(dop, 2.195622042769321)

def test_TDOP():
    dop = get_TDOP(self.receiver, self.satellites)
    self.assertAlmostEqual(dop, 0.8442153787485294)
```

Fix VDOP, add more DOP measurements (07/05/2020) <Slawomir Figiel>

A second pull request by the same user was made a few days later, and the changes made included important modifications to the library functionalities, that will be better discussed in the part regarding the evolution of the Laika library's functionalities and structure.

Remove hardcoded constellation size (09/05/2020) <Slawomir Figiel>
Add method for fetch all satellites from specific time (09/05/2020) <Slawomir Figiel>
Update Walkthrough (09/05/2020) <Slawomir Figiel>
Prevent to re-fetch data for prn not found, warnings instead of exceptions in helpers (16/05/2020) <Slawomir Figiel>

For what concerns the tests, both modified and newly added, in this pull request have been introduced three new unit tests: *test_prns.py*, *test_fetch_sat_info.py*, and *test_time_range_holder.py*. These modifications allowed to have a better division of the different types of tests, i.e., moving the tests which retrieved data from satellites, that were first included in the file *test_ephemeris.py*, to the dedicated file *test_fetch_sat_info.py*. Another test was then added to check that the AstroDog component stops trying to retrieve data after failing for 10 seconds. This simple test is important to verify that the execution of AstroDog doesn't stop unexpectedly.

don't keep retrying if it has already failed (12/05/2020) <Harald Schafer>

One last modification to the test fixed the time used for some satellites.

no old data on Russian servers (28/10/2020) <Harald Schafer>

For what concerns the modification to the library's structure and functionalities, we can see that in the latest years many users other than Harald Schafer contributed to the development of Laika, however until the end of 2019 he was the only active contributor.

The structure of Laika was different from the current one: the *Laika* folder was contained in the *gnss* folder (then eliminated), and the tests were included in the *Laika* folder and not in the main repository *laika_repo*.

Initial commit (20/11/2018) <Harald Schafer>

laika in submodule (20/11/2018) <Harald Schafer>

After some general fixes to the setup files and import files were made, and also caching has been improved, by making Laika download the needed files from the web in case of a cache miss. Caching was already implemented for other types of data, such as ionex and orbits, and it was then implemented also for DGPS. Also, the download method was fixed, allowing to download data from the correct repository corresponding to the specific type of data in case the cached files were not available or invalid.

cache naming for dgps (29/11/2018) <Harald Schafer>
--

pull from web if no cache (29/11/2018) <Harald Schafer>
--

downloading from cache wrongly (29/11/2018) <Harald Schafer>

Caching allows to improve the performances of Laika, but there were other problems with the library that had to be tackled. The first one was the fix of the dilution of precision parameters, which at the beginning was calculated as a unique parameter without making the distinction between the different types of DOP. The calculation of the dilution of precision was split into two components, the vertical and horizontal dilution of precision.

VDOP and HDOP (07/12/2018) <Harald Schafer>
--

Proper VDOP and HDOP (07/12/2018) <Harald Schafer>

Small optimizations were then also made to the download process, to consider the case in which while parsing a DGPS file no constellation among the ones required are in the constellation from which the raw data were acquired.

check constellations in pre-processed measurements (08/12/2018) <Harald Schafer>

Other unexpected behaviors were experienced when retrieving u-blox data, and this was due to a missing control on the *sigId* parameter in an *if* statement, which caused the method *read_raw_ublow()* to read also messages containing information about the subset of signals marked as raw data. This particular behavior of the *sigId* parameter is described in a note in the official u-blox documentation [26].

guess this was always wrong... (12/02/2019) <Harald Schafer>

Up to this moment, orbit data were retrieved by using the data of the GLONASS constellation, and in case of a failure while downloading the data then used the GPS constellation. This approach worked but it has been demonstrated how combining data coming from different constellations improves the availability of signals, gives operators more access, and increases accuracy.

Therefore, the method `get_orbit_data()` was modified to download both GLONASS and GPS orbit data at the same time, rather than downloading only when the download of GLONASS orbit data failed.

always pull both (06/08/2019) <Harald Schafer>

In September 2019, Laika was then migrated to Python3, and this migration was performed by George Hotz and Harald Schafer.

apply 2to3 (25/09/2019) <Andy Haden>
ignore swp files (25/09/2019) <George Hotz>
integer divide (25/09/2019) <George Hotz>
w -> wb (25/09/2019) <George Hotz>
python 3 stuff (27/09/2019) <Harald Schafer>

An important contribution, made by a community member, was made to support the new GNSS codes introduced with the deployment of the new civil signals. This was made by adding the frequencies of the new civil signals to the list of constants used by the AstroDog component and adding the cases corresponding to the new signal types to the methods which retrieve those frequencies given a specific signal type.

<pre>if get_constellation(prn) == 'GPS': if signal[1] == '1': return constants.GPS_L1 elif signal[1] == '2': return constants.GPS_L2 elif signal[1] == '5': return constants.GPS_L5 elif signal[1] == '6': return constants.GALILEO_E6 elif signal[1] == '7': return constants.GALILEO_E5B elif signal[1] == '8': return constants.GALILEO_E5AB</pre>

After the verification by the Comma.ai staff, the commits were merged with the master branch.

Modified download_cors_station function to clarify error if file on server not found Modified RINEXfile class to handle exception where file is not in local cache, ie, not downloaded from server (17/01/2020) <Logan Maynard>
--

Modified program to handle new navigation signals and satellites Created test_positioning.py, a modification of Compute_station_pos.ipynb to test new code. Behavior should be the same, however differential code bias has not been updated for new signals and returns 0 for new signals. Implemented warning to notify user. (18/01/2020) <Logan Maynard>

Before making other big changes, were made small fixes to the download process (the connection now is closed right after the data have been read).

close connection (23/01/2020) <Harald Schafer>

As already anticipated, the contributor **Slawomir Figiel** managed to add the methods needed to calculate the vertical and horizontal Dilution of Precision.

Fix VDOP, add more DOP measurements (07/05/2020) <Slawomir Figiel>

He also discussed with Harald Shafer the dynamics to retrieve the PRN from satellites to avoid problems that could arise after the introduction of new satellites in the different constellations. Eventually, a solution was found by establishing a new convention for the unsupported satellites of which PRN is not available. [27]

Remove hardcoded constellation size (09/05/2020) <Slawomir Figiel>

Add method for fetch all satellites from specific time (09/05/2020) <Slawomir Figiel>

Support cache during read list of items from AstroDog (10/05/2020) <Slawomir Figiel>

Prevent to re-fetch data for prn not found, warnings instead of exceptions in helpers (16/05/2020) <Slawomir Figiel>

Other major contributions were made by **Tyler**, who fixed problems related to the download of RINEX files and the management of exceptions in the case of errors.

minor fixups/changes (14/05/2020) <Tyler>

fix up url from debugging (14/05/2020) <Tyler>

The same user also pointed out other problems in the download process: in particular, he also managed to drastically reduce the download time for CORS coordinates and update the used standard for CORS station position.

download faster + more robust (15/05/2020) <Tyler>

use ITRF2014 position (15/05/2020) <Tyler>

make sure to pass back file paths (15/05/2020) <Tyler>

To better support the contribution of external users like Tyler and Slawomir Figiel, it has been introduced a static analysis at commit time, which is executed every time that a user commits. Like in the case of the other packages, this was done through the implementation of **pre-commit.ci**.

add pre-commit ci (29/05/2020) <Willem Melching>

Install python differently (29/05/2020) <Willem Melching>

fix config location (29/05/2020) <Willem Melching>

try with laika as argument (29/05/2020) <Willem Melching>

run pre-commit after install (29/05/2020) <Willem Melching>

A case that was not considered when retrieving information from a GLONASS satellite was that of a satellite with an unknown channel, so that was fixed by returning None in the case of an unknown channel and returning None as a value for a delay or frequency of a satellite with an unknown channel.

```
def get_glonass_channel(self, prn, time):
    nav = self.get_nav(prn, time)
    if nav:
        return nav.channel
    else:
        return None
```

deal with unknown channel for glonass (02/06/2020) <Harald Schafer>

Another fix for what concerns RINEX files was made again by Tyler, and it was made to deal with RINEX files having a data rate too fast. This allows saving memory and speed by ignoring stations with a data rate below a certain threshold.

```
if self.rate and (hdr[0].microsecond or hdr[0].second % self.rate != 0):
    self._skip_obs(f, len(hdr[2]))
    continue
```

allow decimating rinex files if their rates are too fast (13/07/2020) <Tyler>

The geodesy data are downloaded from online repositories, which may not be always available due to ordinary or extraordinary interventions that could be made throughout the years. For this reason, different URL bases are available to download the geodesy files, and multiple times they have been changed and updated in Laika to ensure that there was always at least one operative service from which these data could be downloaded.

bye nasa (03/11/2020) <Harald Schafer>
missing / (03/11/2020) <Harald Schafer>
at least run that code.. (03/11/2020) <Harald Schafer>
no longer exists (13/11/2020) <Harald Schafer>
catch (05/01/2021) <Harald Schafer>
replace server (13/01/2021) <Harald Schafer>
esa is deleting stuff (04/06/2021) <Harald Schafer>

To ensure high availability of these data, GNSS and GLONASS data have been mirrored in a dedicated GitHub repository, where data coming from the official sources are uploaded periodically.

Added GitHub mirrors for gnss and glonass data (#50) (30/06/2021) <Mitchell Goff>

The download process itself has been modified to support more functionalities, such as the download from both FTP and HTTPS URLs and the secret login when downloading files from the NASA repositories and improved the existing ones. The caching functionalities were updated to also support files downloaded from FTP and HTTP URLs and to throw an exception in case of a missing known preamble in the download link.

The decompression process was optimized as well, introducing a new library (Hatanaka) to automatically decompress the downloaded files rather than doing it manually.

support API (12/11/2020) <Harald Schafer>
add curl (12/11/2020) <Harald Schafer>
more robust (07/01/2021) <Harald Schafer>
some issue with ssl (12/11/2020) <Harald Schafer>
cache_failures (#40) (26/02/2021) <Harald Schafer>
secret login (#41) (11/03/2021) <Adeeb Shihadeh>
allow .netrc usage (#42) (11/03/2021) <Harald Schafer>
update nav data compression format (#43) (08/04/2021) <Martin Valgur>
Use Hatanaka library for RINEX decompression (#45) (19/04/2021) <Martin Valgur>
Added 10s ftp connection timeout (#51) (14/07/2021) <Mitchell Goff>
Catch socket.timeout error (15/07/2021) <mitchellgoffpc>
Only use netrc file for nasa.gov downloads (#53) (24/07/2021) <Mitchell Goff>

3.4 OpenDBC

OpenDBC is a repository containing all the reverse-engineered signals corresponding to the supported car. The signals coming from the cars are reversed engineered through **Cabana**, accessible at <https://my.comma.ai/cabana/>, which allows visualizing the messages exchanged on the CAN bus and creating a DBC file specific for the car. In the master branch are also provided tools that allow generating a DCB file.

3.4.1 Package structure

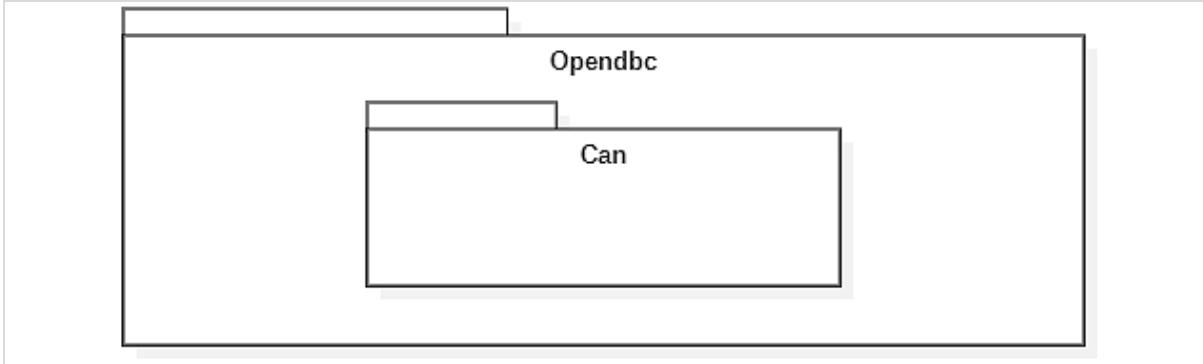


Figure 27 - Opendbc package diagram

OpenDBC is a C++ library that is wrapped using Cython and can be accessed like a normal Python library. The C++ implementation grants high performances, while the callable Python class and methods make the library much more usable.

Language	Files	Blank	Comment	Code
C++	5	102	27	622
Python	6	81	42	293
Cython	2	55	9	200
C/C++ Header	2	31	1	139
SUM:	15	269	79	1.254

Table 26 - Packages included in opendbc main directory

3.4.2 Implementation

A DBC file is a proprietary file format that describes the data that travels over a CAN bus. Each DBC file can contain the definition of multiple DCB files. The starting point of each file is indicated by the field **CM_**.

```
CM_ "Imported file _commadbc starts here";
```

For each file are defined the specifications of the messages exchanged on the bus. A message must have one identifier and must contain one or more signals associated. A simple message has the following structure [28]:

```
BO_ MSG_ID MSG_NAME: DLC SENDER  
SG_ SG_NAME : STARTING_BIT | BIT_SIZE@TYPE (SCALE,OFFSET) [MIN|MAX]  
"UNIT" DST_NODE
```

- the **MSG_ID** field univocally identifies the message and is an unsigned integer.
- the **MSG_NAME** field is the name given to the message to make it human comprehensive and is a string.
- the **DLC** (Data Length Code) field corresponds to the data field length in the requested data frame. [29]
- the **SENDER** field is the component sending the message on the CAN bus.
- the **SG_NAME** field is the name given to the signal to identify it more easily.
- the **STARTING_BIT** field indicates the bit position at which the signal data frame starts.
- the **BIT_SIZE** field indicates the size, in number of bits, of the signal data frame.
- the **TYPE** field indicates the formatting at the data level of the data frame. It can assume values:
 - @0-: signed big-endian
 - @0+: unsigned big-endian
 - @1-: signed little-endian
 - @1+: unsigned little-endian
- the couple (**SCALE**,**OFFSET**) identifies the range and the precision required by the variable to be sent. For instance, if the SCALE is 0.1 and TYPE is unsigned, the variable value can range from 0.0 to 25.5.
- the couple [**MIN**|**MAX**] identifies if there is a minimum and a maximum value that the variable sent can assume.
- the **UNIT** field indicates the unity of measure of the variable (it could be, for example, “miles” or “centimeters”). It can also be an empty value.
- the **DST_NODE** field indicates the destination node, that is the receiver of the signal.

The following fragment of code is a typical message specification that is present in the DBC files contained in the *opendbc* repository. In particular, the following message is the GAS_PEDAL message specification of the Acura ILX 2016.

```
BO_ 513 GAS_SENSOR: 6 INTERCEPTOR
SG_ INTERCEPTOR_GAS : 7|16@0+ (0.253984064, -83.3) [0|1] "" EON
SG_ INTERCEPTOR_GAS2 : 23|16@0+ (0.126992032, -83.3) [0|1] "" EON
SG_ STATE : 39|4@0+ (1,0) [0|15] "" EON
SG_ COUNTER_PEDAL : 35|4@0+ (1,0) [0|15] "" EON
SG_ CHECKSUM_PEDAL : 47|8@0+ (1,0) [0|255] "" EON
```

In the specific case, the message GAS_SENSOR is sent by the component INTERCEPTOR, and is received by the EON, that is the device on which is installed Openpilot. It includes 5 different signals: INTERCEPTOR_GAS, INTERCEPTOR_GAS2, STATE, COUNTER_PEDAL, and CHECKSUM_PEDAL.

DBC files also allow defining the enumeration type, used to visualize names instead of numbers. It is defined as follows:

```
VAL_ MSG_ID SG_NAME VAL1 "VAL1_NAME" VAL2 "VAL2_NAME" [...] ;
```

A VAL_ field can be defined for each SG_NAME, and it allows to define a customize value corresponding to the numeric value (VAL1, VAL2, ...) that the signal can assume. If we consider again the DCB file of the Acura ILX 2016, we can see that for the signal STATE is defined an enumeration type for the values that can assume:

```
VAL_ 513 STATE 5 "FAULT_TIMEOUT" 4 "FAULT_STARTUP" 3 "FAULT_SCE" 2
"FAULT_SEND" 1 "FAULT_BAD_CHECKSUM" 0 "NO_FAULT" ;
```

In *opendbc*, A DBC file can be represented using the struct DBC. Its messages and values can be accessed by knowing the pointers to their first bit and the number of messages and values defined in the file. Signal, Messages, and Values have their corresponding structure in *opendbc* [Figure 28].

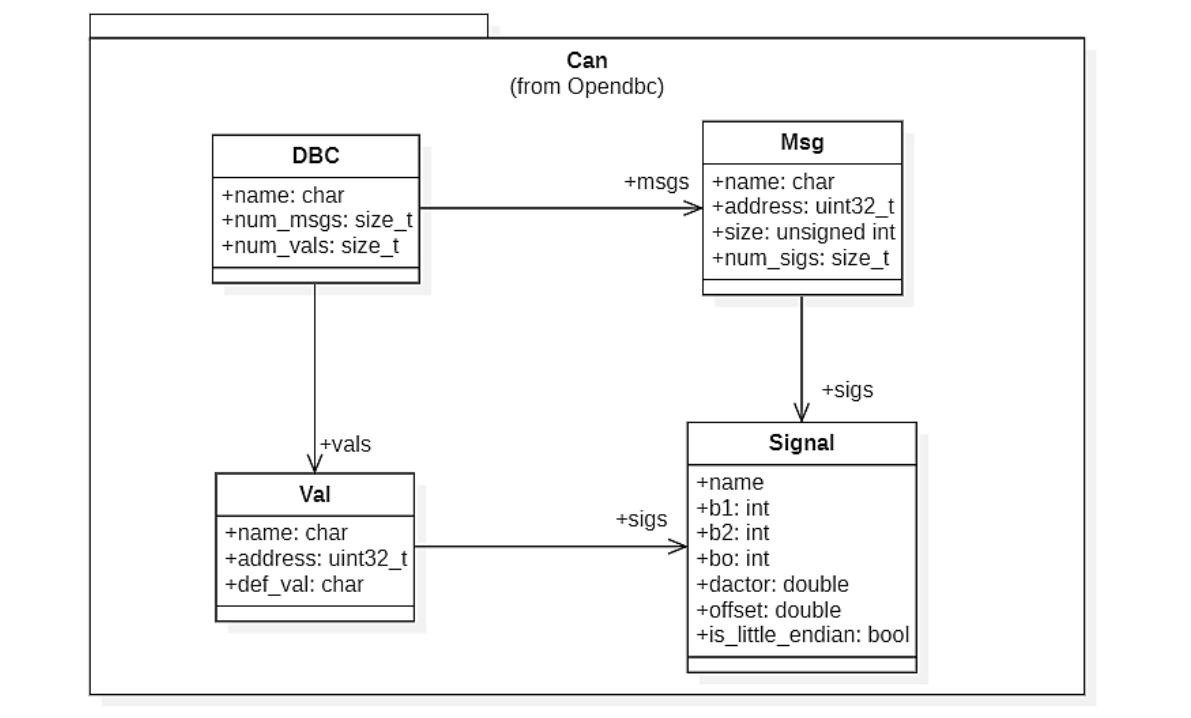


Figure 28 - DBC class diagram. Each part of a DBC file has its implementation (Msg, Signal, Val).

The C++ representation corresponding to the DBC file of a car is generated by the process *process_dbc.py*. The method *process(in_fn, out_fn)* generate a file starting from a template, defined in the file *dbc_template.cc*, and to perform this operation it uses the library **Jinja2**.

```
template_fn = os.path.join(os.path.dirname(__file__), "dbc_template.cc")
with open(template_fn, "r") as template_f:
    template = jinja2.Template(template_f.read(),
                               trim_blocks = True,
                               lstrip_blocks = True)
```

After defining what is the template to use, a *dbc* object (a Python representation of the .dbc file) is created by parsing the .dbc file. When a *dbc* object is initialized, passing the path of the .dbc file, each line of the text file is stored in a list of strings, with each element of the list corresponding to a line of text.

```
def __init__(self, fn):
    with open(fn, encoding="ascii") as f:
        self.txt = f.readlines()
```

The definition of some regular expressions will then help to parse each line of the file, since applying the regular expression to the line will split it into different groups that can then be added to the corresponding array of elements.

```
bo_regex = re.compile(r"^\BO_\_ (\w+) (\w+) *: (\w+) (\w+)")
sg_regex = re.compile(r"^\SG_\_ (\w+) : (\d+) | (\d+)@(\d+) ([\+|\-]) \
    (([0-9.+|-eE]+), ([0-9.+|-eE]+)) \[([0-9.+|-eE]+) \
    | ([0-9.+|-eE]+)\] \"(.*)\" (.*)")
sgm_regex = re.compile(r"^\SG_\_ (\w+) (\w+) *: (\d+) | (\d+)@(\d+) ([\+|\-]) \
    (([0-9.+|-eE]+), ([0-9.+|-eE]+)) \[([0-9.+|-eE]+) \
    | ([0-9.+|-eE]+)\] \"(.*)\" (.*)")
val_regex = re.compile(r"^\VAL_\_ (\w+) (\w+) (\s*[-+]? \
    [0-9]+\s+\\".+\?"[^;]*")")
```

For instance, applying the regular expression `bo_regex` to a line that starts with “`BO_`” will generate 3 groups, corresponding to the three components of a DBC message: `MSG_ID`, `MSG_NAME`, and `DLC`.

```
if l.startswith("BO_ "):
    dat = bo_regex.match(l)
    if dat is None:
        print("bad BO {0}".format(l))
    name = dat.group(2)          # MSG_NAME
    size = int(dat.group(3))    # DLC
    ids = int(dat.group(1), 0)  # MSG_ID
    if ids in self.msgs:
        sys.exit("Duplicate address detected %d %s" % (ids, self.name))
    self.msgs[ids] = ((name, size), [])
```

According to the car manufacturer, the checksum parameters are initialized in the proper way.

```
if can_dbc.name.startswith(("honda_", "acura_")):
    checksum_type = "honda"
    checksum_size = 4
    counter_size = 2
    checksum_start_bit = 3
    counter_start_bit = 5
    little_endian = False
elif can_dbc.name.startswith(("Toyota_", "lexus_")):
```

After some sanity checks on the COUNTER and CHECKSUM rules, the template is then rendered, meaning different Signal, Msg, and Val objects are generated and stored.

```
parser_code = template.render(dbc = can_dbc,
                               checksum_type = checksum_type,
                               msgs = msgs,
                               def_vals = def_vals,
                               len = len)
```

The destination of the file is indicated as `out_fn`.

```

with open(out_fn, "a+") as out_f:
    out_f.seek(0)
    if out_f.read() != parser_code:
        out_f.seek(0)
        out_f.truncate()
        out_f.write(parser_code)

```

The two main components through which Openpilot can manipulate data traveling on the CAN bus are CANPacker and CANParser.

CANPacker provides the methods to generate and lookup for CAN messages on the CAN bus. [Figure 29]

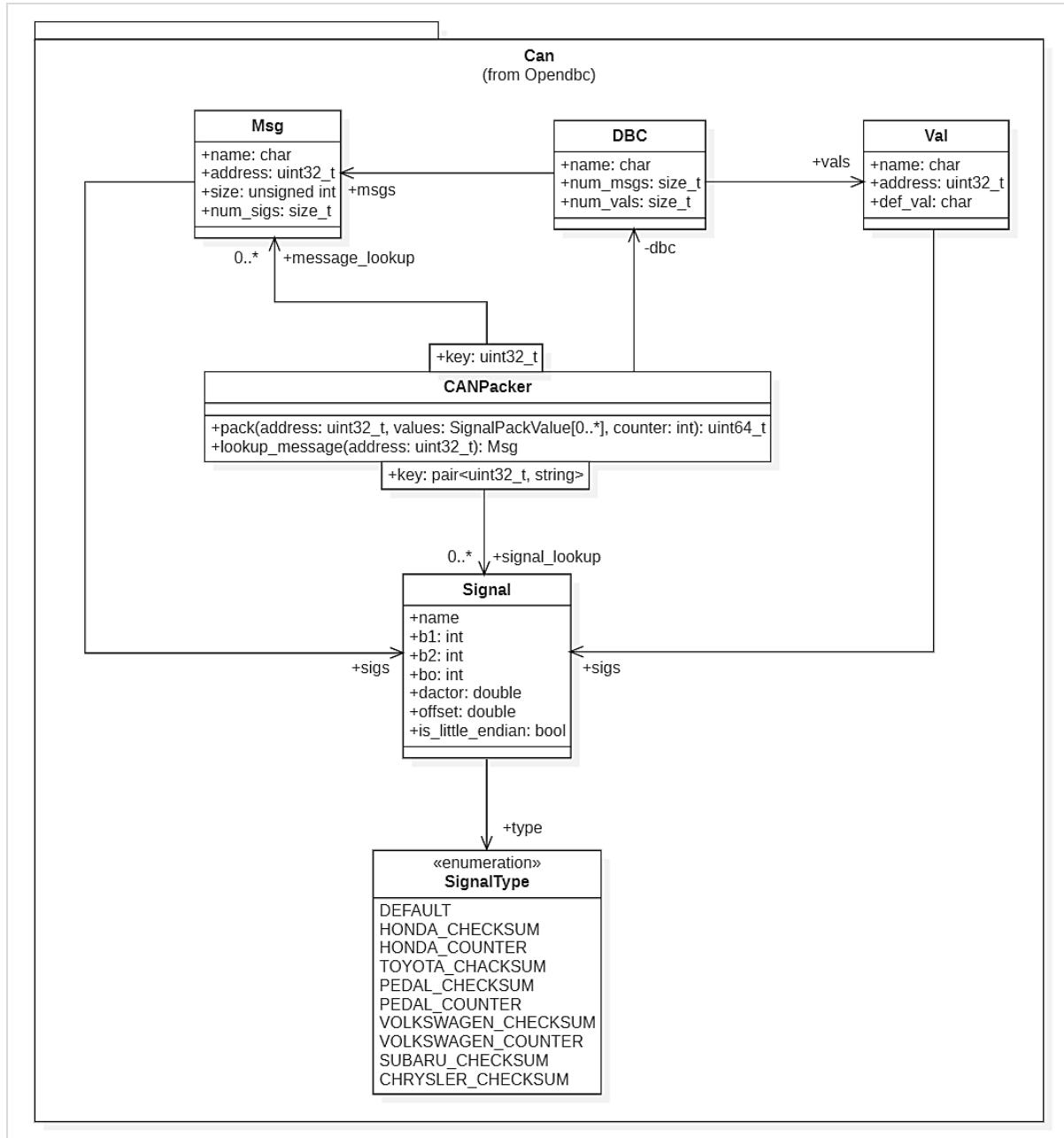


Figure 29 - CANPacker class diagram. The component construct a message starting from a series of values.

When the CANPacker is initialized, it is first associated with a DBC file, and the file representation is stored in the variable *dbc*.

For each message of the DBC file, a pointer to that is added to the vector *message_lookup*, so that it will be possible to easily access each message when needed.

The vector signal lookup is also initialized, by finding all the different signals that are present for each DBC message in the given DCB file. This vector will contain all the signals that are specified in the DBC file.

```
CANPacker::CANPacker(const string& dbc_name) {
    dbc = dbc_lookup(dbc_name);
    assert(dbc);
    for (int i = 0; i < dbc->num_msgs; i++) {
        const Msg* msg = &dbc->msgs[i];
        message_lookup[msg->address] = *msg;
        for (int j=0; j<msg->num_sigs; j++) {
            const Signal* sig = &msg->sigs[j];
            signal_lookup[make_pair(msg->address, string(sig->name))] = *sig;
        }
    }
    init_crc_lookup_tables();
}
```

In the end, is also set up a static CRC table, which allows performing a consistency check on the data. CRC stands for Cyclic Redundancy Check and is an algorithm that calculates a checksum to attach to data that helps to identify the bit errors that could occur during data transmission.

The method *pack()* creates a DCB message by combining all the signals that have to be included in the message identified by the value *address*, which can indicate both the name and the ID of the message.

```
for (const auto& sigval : signals) {
    string name = string(sigval.name);
    double value = sigval.value;
    auto sig_it = signal_lookup.find(make_pair(address, name));
    if (sig_it == signal_lookup.end()) {
        WARN("undefined signal %s - %d\n", name.c_str(), address);
        continue;
    }
    const auto& sig = sig_it->second;
    int64_t ival = (int64_t)(round((value - sig.offset) / sig.factor));
    if (ival < 0) {
        ival = (1ULL << sig.b2) + ival;
    }
    ret = set_value(ret, sig, ival);
}
```

For each signal in the vector of signals passed to the function *pack()*, the name and the value of each element are retrieved and is checked if the signal is valid or is undefined for that specific message, identified by checking the pair (MSG_NAME, SG_NAME), where MSG_NAME is equal to the address.

If it is undefined, it means that the signal is not part of that message, and the next operations are not performed, and the control continues on the next elements of the vector.

If the signal is valid, its value is calculated by performing the operation $(value - sig.offset)/sig.factor$. If the value is less than 0, then the *two's complement* is calculated, by taking the MAX value allowed by the signal, shifting its value of 1 bit to the left, and adding the negative value of the signal.

The value is then converted into a binary value through the function *set_value()*. What this function does is to apply a mask to the value to keep only the relevant bits.

```
static uint64_t set_value(uint64_t ret, const Signal& sig, int64_t ival)
{
    int shift = sig.is_little_endian? sig.b1 : sig.bo;
    uint64_t mask = ((ULL << sig.b2)-1) << shift;
    uint64_t dat = (ival & ((ULL << sig.b2)-1)) << shift;
    if (sig.is_little_endian) {
        dat = ReverseBytes(dat);
        mask = ReverseBytes(mask);
    }
    ret &= ~mask;
    ret |= dat;
    return ret;
}
```

After that, is checked if the message also has to include a *counter* signal, mainly needed for synchronization purposes, and is verified if the couple (address, “COUNTER”) is valid, and if it is, then the signal is added to the pack of signals that will be sent to the receiver, also in this case after applying bit masking.

```
if (counter >= 0) {
    auto sig_it = signal_lookup.find(make_pair(address, "COUNTER"));
    if (sig_it == signal_lookup.end()) {
        WARN("COUNTER not defined\n");
        return ret;
    }
    const auto& sig = sig_it->second;
    if ((sig.type != SignalType::HONDA_COUNTER) &&
(sig.type != SignalType::VOLKSWAGEN_COUNTER)) {
        WARN("COUNTER signal type not valid\n");
    }
    ret = set_value(ret, sig, counter);
}
```

Finally, it is checked if a checksum signal has to be included in the message, by verifying that the pair (address, “CHECKSUM”) is in the *signal_lookup* vector, and if it is the checksum is then retrieved and added to the pack of signals.

```
if (sig.type == SignalType::HONDA_CHECKSUM) {
    unsigned int chksm = honda_checksum(address, ret,
                                         message_lookup[address].size);
    ret = set_value(ret, sig, chksm);
} else if (...) { // check other checksums
    ...
}
```

In the end, the pack of signals stored in the variable *ret* is returned by the function. The CANParser component provides the methods to parse the messages that are sent over the CAN bus.

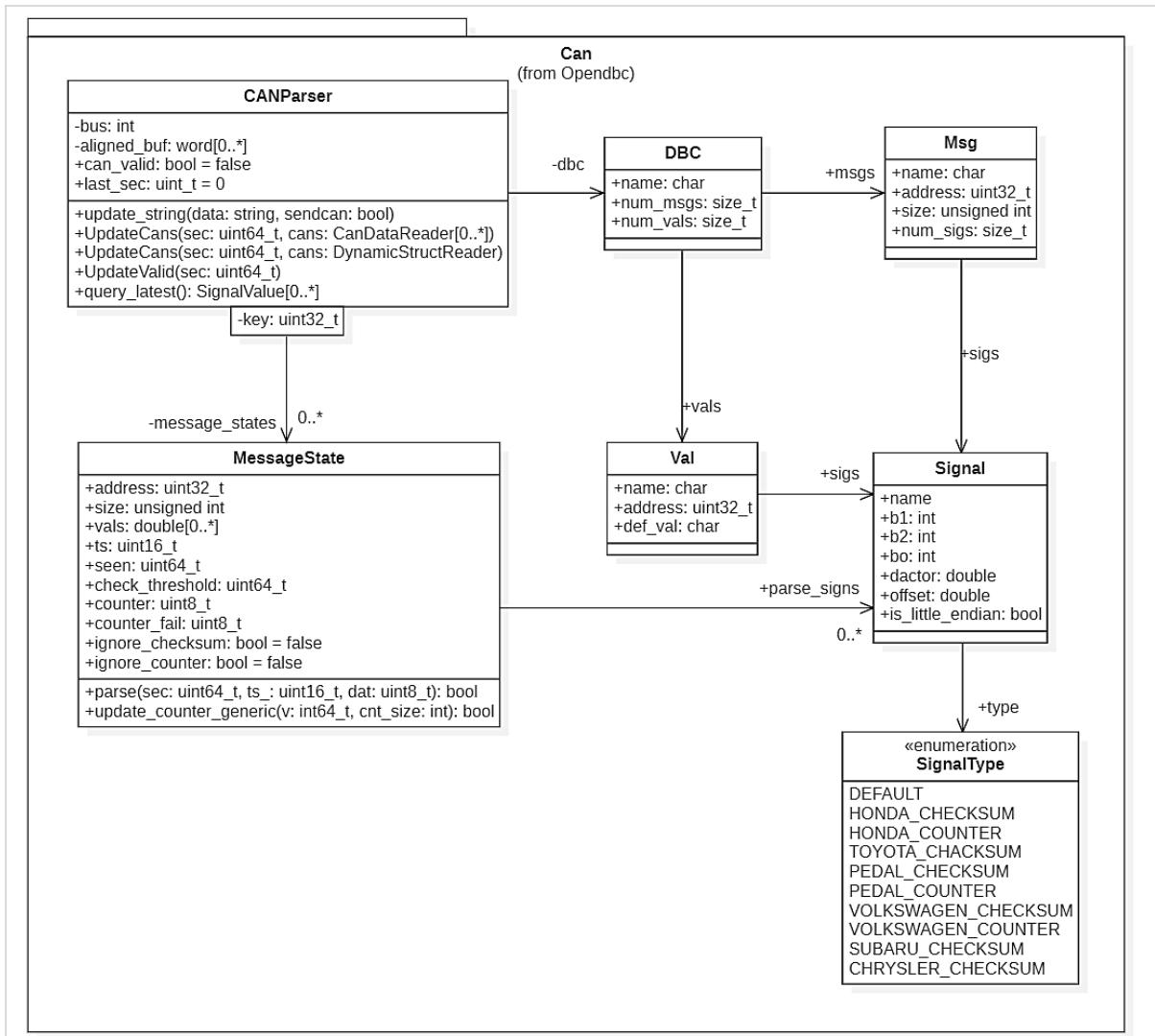


Figure 30 - CANParser class diagram. The component parses the received message through the MessageState component

The method `UpdateCans` parses the list of `CanData` received on the specified CAN bus. The `CanData` type is a struct, defined in the file `log.capnp` in the *cereal* library, that contains four parameters.

```
struct CanData {
    address @0 :UInt32;
    busTime @1 :UInt16;
    dat      @2 :Data;
    src      @3 :UInt8;
}
```

For each `CanData` parameter, its *source* field is compared with the *bus* field of the `CANParser` to check that both match: if not, the current `CanData` element is skipped. For the `CanData` elements that pass the control, the state is retrieved. The state of a `CanData` is expressed through the `MessageState` class, which holds all the relevant information for a message, including the signals and values that compose it, its size, and address.

The `MessageState` class provides the functionalities to parse the signals of which is composed a message through the method `parse(uint64_t sec, uint16_t ts_, uint8_t * dat)`. This method iterates over the vector of signals `parse_sig`, and for each of them, it performs a series of checks.

The first control is on the byte ordering of the signal: it performs a right shift of a number of bits equals to the minimum value allowed, in the case of a little-endian signal, or equals to the starting bit in the case of a big-endian signal. In both cases, a bit mask is applied to select only the relevant number, that will be equal to the number of bits of the biggest value allowed.

```
if (sig.is_little_endian)
    tmp = (dat_le >> sig.b1) & ((1ULL << sig.b2)-1);
else
    tmp = (dat_be >> sig.bo) & ((1ULL << sig.b2)-1);
```

If the signal is signed, then the two's complement is calculated.

```
if (sig.is_signed)
    tmp -= (tmp >> (sig.b2-1)) ? (1ULL << sig.b2) : 0; //signed
```

Then, if the signal requires to verify the checksum and the current signal is of type checksum, the value of the signal is compared with the result of a calculation of the checksum, that is specific for each car manufacturer.

```
if (!ignore_checksum) {
    if (sig.type == SignalType::HONDA_CHECKSUM) {
        if (honda_checksum(address, dat_be, size) != tmp) {
            return false; // checksum fail
        }
    } else if (...) { // the other checksums are checked
        [...]
    }
}
```

A similar check is made for the signals of type *counter*. If the signal is a *counter* and it fails the check 5 times, then the execution of the methods ends.

```
if (!ignore_counter) {
    if (sig.type == SignalType::HONDA_COUNTER) {
        if (!update_counter_generic(tmp, sig.b2)) {
            return false;
        }
    } else if (sig.type == SignalType::VOLKSWAGEN_COUNTER) {
        if (!update_counter_generic(tmp, sig.b2)) {
            return false;
        }
    } else if (sig.type == SignalType::PEDAL_COUNTER) {
        if (!update_counter_generic(tmp, sig.b2)) {
            return false;
        }
    }
}
```

If all the checks pass, the value is computed by multiplying it by the signal factor and adding the signal offset and is added to the vector of values on the MessageState object.

```
vals[i] = tmp * sig.factor + sig.offset;
```

The method parse is called for each can message over which the method UpdateCans iterates.

```
void CANParser::UpdateCans(uint64_t sec, const List<CanData>::Reader& cans) {
    int msg_count = cans.size();

    DEBUG("got %d messages\n", msg_count);

    for (int i = 0; i < msg_count; i++) {
        auto cmsg = cans[i];
        // parse the messages
        if (cmsg.getSrc() != bus) {
            // DEBUG("skip %d: wrong bus\n", cmsg.getAddress());
            continue;
        }
        auto state_it = message_states.find(cmsg.getAddress());
        if (state_it == message_states.end()) {
            // DEBUG("skip %d: not specified\n", cmsg.getAddress());
            continue;
        }

        if (cmsg.getDat().size() > 8) continue; //shouldn't ever happen
        uint8_t dat[8] = {0};
        memcpy(dat, cmsg.getDat().begin(), cmsg.getDat().size());

        state_it->second.parse(sec, cmsg.getBusTime(), dat);
    }
}
```

The same method has two different implementations: the former is used if multiple CAN messages are passed to the function, while if the second implementation is for parsing a single CAN message. The parsing process is the same for both the implementations.

```
void CANParser::UpdateCans(uint64_t sec, const
capnp::DynamicStruct::Reader& cmsg) {
    // assume message struct is `cereal::CanData` and parse
    assert(cmsg.has("address") &&
           cmsg.has("src") &&
           cmsg.has("dat") &&
           cmsg.has("busTime"));

    if (cmsg.get("src").as<uint8_t>() != bus) {
        DEBUG("skip %d: wrong bus\n", cmsg.get("address").as<uint32_t>());
        return;
    }

    auto state_it =
message_states.find(cmsg.get("address").as<uint32_t>());
    if (state_it == message_states.end()) {
        DEBUG("skip %d: not specified\n",
cmsg.get("address").as<uint32_t>());
        return;
    }

    auto dat = cmsg.get("dat").as<capnp::Data>();
    if (dat.size() > 8) return; //shouldn't ever happen
    uint8_t data[8] = {0};
    memcpy(data, dat.begin(), dat.size());
    state_it->second.parse(sec, cmsg.get("busTime").as<uint16_t>(), data);
}
```

The validity of a CAN message is checked through the method `UpdateValid()`. For each message state, the method verifies if the time at which the message is seen respect the allowed threshold. The parameter `check_threshold` indicates the maximum number of seconds that can pass from when the message is sent to when it is seen. If one message does not respect the threshold, the whole CAN pack of messages is invalidated.

```
if (state.check_threshold > 0 &&
    (sec - state.seen) > state.check_threshold) {
    if (state.seen > 0) {
        DEBUG("0x%X TIMEOUT\n", state.address);
    } else {
        DEBUG("0x%X MISSING\n", state.address);
    }
    can_valid = false;
}
```

3.4.3 Usage

The OpenDBC repository and its functionalities are leveraged by the packages of selfdrive which let the car of the different manufacturers interface with Openpilot.

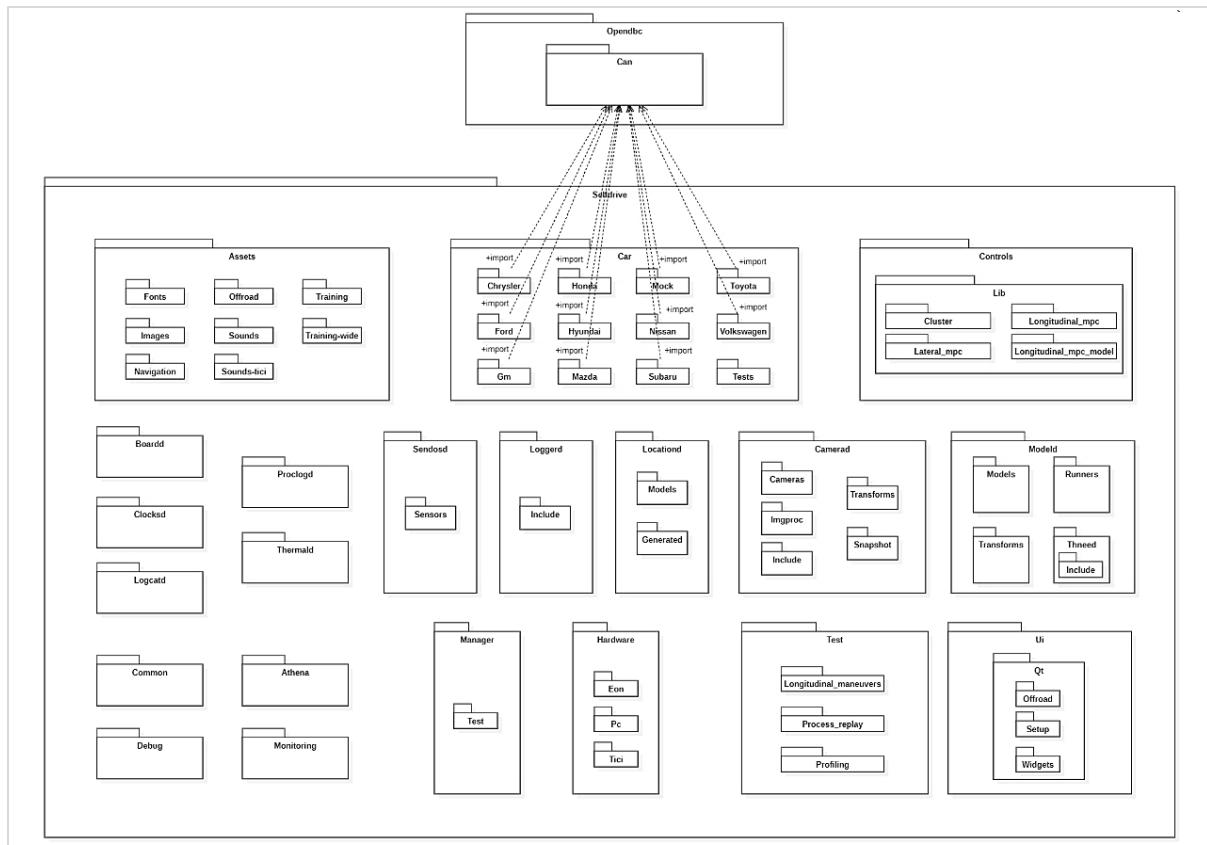


Figure 31 - Import dependencies between selfdrive and opendbc

The C++ implementation of the library is wrapped using Cython and can be accessed by importing the files `parser.py` and `packer.py`, which expose the class `CANParser` and `CANPacker`, respectively. Openpilot has to deal with the different car manufacturers differently, since each of them adopts different standards and approaches to send data over the CAN bus. In general, for each car manufacturer, there is a specific interface and a set of signal signals allowed.

The car state is managed through the class `CarState`, which allows intercepting the signals traveling on the bus and parsing them. The parse happens thanks to the **CANParser** provided by the OpenDBC functionalities.

We can distinguish two types of signals, that are also managed by two different parsers:

- the signals coming directly from the car, that include gear change signals, door opening signals, etc.
 - signals coming from the cameras.

In the first case, the CANParser is created by the method `get_can_parser()`.

Inside of the method is defined the list of all the possible signals that can travel on the bus, by specifying the signal name and the signal address, that corresponds to the message name in the dbc file format. It also defines a *check* array, which contains the couples composed by the signal address and their working frequency, expressed in Hertz.

These signals are hardcoded and change from car manufacturer to car manufacturer.

```
def get_can_parser(CP):
    signals = [
        ("PRNDL", "GEAR", 0),      # (sig_name, sig_address, default)
        ("DOOR_OPEN_FL", "DOORS", 0),
        # [...]
    ]
    checks = [
        ("BRAKE_2", 50),          # (sig_address, frequency)
        ("EPS_STATUS", 100),
        # [...]
    ]
    if CP.enableBsm:
        signals += [
            ("BLIND_SPOT_RIGHT", "BLIND_SPOT_WARNINGS", 0),
            ("BLIND_SPOT_LEFT", "BLIND_SPOT_WARNINGS", 0),
        ]
        checks += [("BLIND_SPOT_WARNINGS", 2)]
    return CANParser(DBC[CP.carFingerprint]["pt"], signals, checks, 0)
```

Similarly, the parser managing the camera signals is created by the method `get_cam_can_parser()`.

```
def get_cam_can_parser(CP):
    signals = [
        # sig_name, sig_address, default
        ("COUNTER", "LKAS_COMMAND", -1),
        ("CAR_MODEL", "LKAS_HUD", -1),
        ("LKAS_STATUS_OK", "LKAS_HEARTBIT", -1)
    ]
    checks = [
        ("LKAS_COMMAND", 100),
        ("LKAS_HEARTBIT", 10),
        ("LKAS_HUD", 4),
    ]
    return CANParser(DBC[CP.carFingerprint]["pt"], signals, checks, 2)
```

By using this specific case, relative to the Chrysler CanState, we can see how the different signals are also specified in the .dbc file.

```
BO_ 658 LKAS_COMMAND: 6 XXX
SG_ COUNTER : 39|4@0+ (1,0) [0|15] "" XXX
SG_ CHECKSUM : 47|8@0+ (1,0) [0|255] "" XXX
SG_ LKAS_STEERING_TORQUE : 2|11@0+ (1,-1024) [0|1] "" XXX
SG_ LKAS_HIGH_TORQUE : 4|1@0+ (1,0) [0|1] "" XXX

BO_ 678 LKAS_HUD: 8 XXX
SG_ LKAS_ICON_COLOR : 1|2@0+ (1,0) [0|3] "" XXX
SG_ LKAS_LANE_LINES : 19|4@0+ (1,0) [0|1] "" XXX
SG_ LKAS_ALERTS : 27|4@0+ (1,0) [0|1] "" XXX
SG_ CAR_MODEL : 15|8@0+ (1,0) [0|255] "" XXX
```

A CANParser is also created to parse the signals relative to the radar sensors of the car. These include information relative to the distance of the car from the obstacles in the long and short-range and the relative speed of the car.

```
def _create_radar_can_parser(car_fingerprint):
    msg_n = len(RADAR_MSGS_C)
    signals = list(zip(
        ['LONG_DIST'] * msg_n +
        ['LAT_DIST'] * msg_n +
        ['REL_SPEED'] * msg_n,
        RADAR_MSGS_C * 2 +      # LONG_DIST, LAT_DIST
        RADAR_MSGS_D,           # REL_SPEED
        [0] * msg_n +           # LONG_DIST
        [-1000] * msg_n +       # LAT_DIST
        [-146.278] * msg_n))   # REL_SPEED set to 0
    checks = list(zip(RADAR_MSGS_C +
                       RADAR_MSGS_D,
                       [20]*msg_n +          # 20Hz (0.05s)
                       [20]*msg_n))          # 20Hz (0.05s)
    return CANParser(DBC[car_fingerprint]['radar'], signals, checks, 1)
```

The second main component of OpenDBC, the **CANPacker**, is used by the car controller component (*carcontroller.py*) to generate the CAN messages. Each CarController class has a CanPacker, which is initialized at the beginning of the execution. The packer object is utilized to create new messages that can be elaborated by Openpilot.

```
new_msg3 = create_lkas_command(self.packer,
                               int(apply_steer),
                               self.gone_fast_yet,
                               frame)
```

Each of the method used to initialize the different messages behaves in the same way: the different signals are associated with the parameters passed to the function and then the method and the packer generates the new message.

```
def create_lkas_command(packer, apply_steer, moving_fast, frame):
    values = {
        "LKAS_STEERING_TORQUE": apply_steer,
        "LKAS_HIGH_TORQUE": int(moving_fast),
        "COUNTER": frame % 0x10,
    }
    return packer.make_can_msg("LKAS_COMMAND", 0, values)
```

The method `make_can_msg()`, defined in the Cython wrapper of the C++ implementation of the CANPacker component, after computing the size of the message and its address, it computes the binary representation of the message.

The representation is created through the method `pack()` provided by the CANPacker component, already described in the previous pages.

```
cpdef make_can_msg(self, name_or_addr, bus, values, counter=-1):
    cdef int addr, size
    if type(name_or_addr) == int:
        addr = name_or_addr
        size = self.address_to_size[name_or_addr]
    else:
        addr, size =
    self.name_to_address_and_size[name_or_addr.encode('utf8')]
    cdef uint64_t val = self.pack(addr, values, counter)
    val = self.ReverseBytes(val)
    return [addr, 0, (<char *>&val)[:size], bus]
```

3.4.4 Testing

The OpenDBC functionalities are tested through the `unittest` framework. The test folder includes three Test Cases and a total of four tests, which look into the main component of the library.

- **TestCanParserPackerExceptions:** it verifies that the CANPacker and CANParser are able to throw an exception in the case in which an invalid file is given to the two components.

TestCanParserPackerExceptions
+ test_civic_exceptions()

Table 27 - `TestCanParserPackerExceptions` test case

- **TestCANDefine:** It tries to generate a CANDefine object of a Honda Civic Touring 2016 starting from the corresponding DBC file and checks that the generated values are the expected ones.

TestCANDefine
+ test_civic()

Table 28 - `TestCANDefine` test case

- **TestCanParserPacker:** it instantiates both a CANParser and a CANPacker and generates a message, that then tries to parse and verifies that the values given to the CANPacker are the same as the ones that the CANParser retrieves for the same message.

TestCanParserPacker
+ test_civic()
+ test_subaru()

Table 29 - `TestCanParserPacker` test case

The unit tests are executed with GitHub Actions, which executed the tests whenever a new commit is made, to ensure that each change doesn't modify the behavior of the software and work as expected.

```
Run docker run opendbc bash -c "python -m unittest discover opendbc"
  docker run opendbc bash -c "python -m unittest discover opendbc"
    shell: /usr/bin/bash -e {0}
    env:
      RUN: docker run --shm-size 1G --rm opendbc /bin/sh -c
      BUILD: docker pull $(grep -ioP '(?=<^from)\s+\$+' Dockerfile) ||
true
      docker pull docker.io/commaai/opendbc:latest || true
      docker build --cache-from docker.io/commaai/opendbc:latest -t opendbc
-f Dockerfile
-----
Warning, using python time.time() instead of faster sec_since_boot
-----
Ran 4 tests in 0.061s

OK
```

As stated in the introduction of this paragraph, Cabana is a tool that helps to identify and classify the different signals that travel on the CAN bus. The tool can also be used to test the DBC files and to identify new signals, by connecting the car directly to Cabana through Panda.

3.4.5 Development and community contributions

Unlike other packages in the repository, OpenDBC has many contributions coming from the community members. The reason for the high level of contribution is probably that generating a .dbc file for a new car does not require any particular skill or experience in coding: in fact, tools such as *Cabana* and *CANdevStudio* allow to read the traffic on the CAN bus and help the contributors to identify the different messages that travel on it.

General modifications and improvement of the library functionalities, however, are done almost exclusively by the Comma.ai team, with some exceptions. OpenDBC directory was made a public repository on May 31st, 2017. When it was created, it only supported five cars.

Import the DBCs from Openpilot (31/05/2017) <George Hotz>

make opendbc import work, and ignore junk (05/06/2017) <George Hotz>

Back in 2017, the data needed to train the predictive model was collected by means of an Android application, **CHFFR**. The phone, mounted on the car, would act as a dashcam, and record the trip. The recorded information could be used to train the self-driving car model [30]. To better support the data recorded through the app, ad-hoc signals and messages were added to the dbc specification to map standard messages and signals to custom metrics, which could be displayed in the CHFFR application. [31]

Add chffr metrics for cars (10/09/2017) <Andy Haden>

As already anticipated, Cabana was used to help users to reverse engineer the signals traveling on their car's network and create a specific DBC file. This was the case of a user,

Jeankalud, who thanks to Cabana reverse-engineered his Tesla and updated the relative DBC file, correcting many signals and messages parameters. [32]

Lots of correction, thanks to cabana! (03/11/2017) <jeankalud>

To optimize the generation of the DBC files for the different cars, it was introduced a preprocessor which had the role of creating a DBC file by combining a brand-specific DBC file with a model-specific DBC file.

The DBC file relative to the model will contain an “*IMPORT*” field, which is not part of the specifications of the DBC standards but is needed by the generator to include in the generated file the brand-specific fields.

In a model-specific DBC file will be present the following instruction (in the specific case, for a Honda car).

CM_ "IMPORT _hondadbc"

On the other hand, the generator finds the files starting with that specific pattern and include the right brand specific DBC file.

```
dbc_file = open(os.path.join(dir_name, filename)).read()
include = re.search(r'CM_ "IMPORT (.*)"', dbc_file)

if include is not None:
    dbc_file = dbc_file.replace(include.group(0),
                                '\nCM_ "%s starts here"' % filename)
    include_path = os.path.join(dir_name, include.group(1))
    # Load included file
    include_file = open(include_path).read()
    include_file = 'CM_ "Imported file %s starts here"\n'%include.group(1)
                + include_file
    dbc_file = include_file + dbc_file
```

The generator is only included in the version of Openpilot on the master branch, since is meant to be a developer tool that aims to facilitate the generation of new DBCs for cars that are not supported and test new values for messages and signals creating customs DBCs.

dbc file preprocessor (27/01/2018) <Willem Melching>
whitespace consistency (27/01/2018) <Willem Melching>
fix comments and values (27/01/2018) <Willem Melching>
fix PCM_SPEED factor (27/01/2018) <Willem Melching>
cleanup (27/01/2018) <Willem Melching>
move generated files to root folder (27/01/2018) <Willem Melching>
regenerate new steer torque eps factor (27/01/2018) <Willem Melching>
readme explanation of preprocessor (27/01/2018) <Willem Melching>
Updated README with a recommended overview (19/05/2018) <Riccardo Biasini>

Minor fixes were then made to the DBC files structure, such as changing the end line sequence from CRLF (*0x0D* at the end of each line) to LF (*0x0A* at the end of each line) and changing the values that signals without a unit of measure could have, making them act like boolean variables.

convert all line endings to unix style (13/02/2018) <Willem Melching>

set scaling to 1 for brake and gas which have no real unit (21/02/2018) <Willem Melching>

In March 2018 was introduced Comma Pedal, a solution to virtually pressing the gas pedal on Honda and Acura cars to enable stop and go functionalities on the supported models. To communicate with the device, it was required the definition of new messages, corresponding to the messages sent by the Comma Pedal on the CAN bus.

Comma Pedal: sending 2 tracks on 0x200 (03/03/2018) <Riccardo Biasini>

Comma Pedal: added state byte and enable bit (03/03/2018) <Riccardo Biasini>

Comma Pedal: made GAS_COMMAND 6 bytes (03/03/2018) <Riccardo Biasini>

With the introduction of the Comma Pedal, it was also convenient to move the messages, signals, and values related to it in a file like the DBC file for a brand of a car, to be imported in all the model-specific DBC files. The generator had to be modified to be able to parse multiple headers imported (the comma DBC file and the car brand DBC file).

```
CM_ "IMPORT _honda_2017dbc"  
CM_ "IMPORT _commadbc"
```

```
dbc_file = open(os.path.join(dir_name, filename)).read()  
dbc_file = '\nCM_ "%s starts here"\n' % filename + dbc_file  
  
includes = re.finditer(r'CM_ "IMPORT (.*)"', dbc_file)  
for include in includes:  
    dbc_file = dbc_file.replace(include.group(0), '')  
    include_path = os.path.join(dir_name, include.group(1))  
    include_file = open(include_path).read()  
    include_file = '\n\nCM_ "Imported file %s starts here"\n' %  
                 include.group(1) + include_file  
    dbc_file = include_file + dbc_file
```

update generator script to allow for multiple imports (09/03/2018) <Willem Melching>

Pedal Interceptor: fault state VAL moved to _comma (13/03/2018) <Riccardo Biasini>

In a pull request by an external contributor, **dekerr** on GitHub, he updated the generator to make it build the included files in the correct order and to improve the readability of the generated files.

Update generator.py (17/06/2018) <dekerr>

Update generator.py (17/06/2018) <dekerr>

The changes, however, introduced some problems when generating the files, but the bugs were soon pointed out by the Comma.ai staff and fixed.

Update generator.py (17/06/2018) <dekerr>

Update generator.py (17/06/2018) <dekerr>

Update generator.py (17/06/2018) <dekerr>

small generator cleanup (17/06/2018) <Willem Melching>

pass dirname explicitly to generator helperfunctions and whitespace (17/06/2018)
<Willem Melching>

Another member of the community instead managed to fix a bad encoding problem that afflicted some files, and in particular some units of measure like the Celsius degrees (°C).

```
- SG_EngineCoolantTemp : 23|8@0+ (1,-40) [0|0] "♦C" NEO  
+ SG_EngineCoolantTemp : 23|8@0+ (1,-40) [0|0] "°C" NEO
```

Syntax and encoding fixes (#111) (17/09/2018 07:14) <Oscar Söderlund>

Many modern cars have the Eco and Sport mode, and a user managed to decode those signals traveling on the CAN bus and adding them to the DBC file of Toyota and Lexus cars.

```
SG_SPORT_ON : 2|1@0+ (1,0) [0|1] "" XXX  
SG_ECON_ON : 40|1@0+ (1,0) [0|1] "" XXX  
  
VAL_ 956 SPORT_ON 0 "off" 1 "on";  
VAL_ 956 ECON_ON 0 "off" 1 "on";
```

Update Gear Packet with Eco and Sport button (#135) (29/01/2019) <arne182>

More adjustments were then made to Comma Pedal, especially to the checksum and counter signals.

Pedal: same checksum and counter (#143) (03/03/2019) <Riccardo Biasini>

Pedal: back again at 6 bytes (03/03/2019) <Riccardo Biasini>

To enable the support of more cars, it was needed to support their hardware first and decode how the basic component for different car manufacturers and models communicate with the car. In particular, many cars from Volvo used a specific radar, the Delphi ESR Radar, which was not supported until that moment. Also on this occasion, the pull request was made by a community member, **AdasCoder** on GitHub, which decoded the 64 messages and the relative signals that the ESR Radar uses to communicate with the car. [33]

Add files via upload (#147) (14/03/2019) <AdasCoder>

Apart from the addition of new messages and signals for the already supported cars the addition of new cars, most of which were added thanks to the community members, the main repository functionalities were not modified, but only small fixes to the file formatting were made.

Fix manually created dbc files (#154) (04/04/2019) <Maksim Salau>

Fix Spelling (#180) (03/09/2019) <Arne Schwarck>

Remove non ascii characters (21/09/2019) <Willem Melching>

After the migration to Python 3, also the CANParser and CANPacker were moved from Openpilot's repository to OpenDBC. This allowed to achieve a higher level of abstraction and have a more coherent separation of the code and its functionalities. This migration required more than two weeks of work and the collaboration of many members of the Comma.ai team.

move generator to python3 (27/09/2019) <George Hotz>

Can migration (#199) (23/11/2019) <Riccardo Biasini>
fix gitignore (27/11/2019) <Comma Device>
OpenDBC needs cereal (02/12/2019) <George Hotz>
consistent naming (03/12/2019) <Willem Melching>
unify can packer and parser (03/12/2019) <Willem Melching>
add test for can define (03/12/2019) <Willem Melching>
move CANDefine to parser code (03/12/2019) <Willem Melching>
no more python version of libdbc, everything through Cython (03/12/2019) <Willem Melching>
packer depends on libdbc (03/12/2019) <Willem Melching>
Azure pipelines ci (#202) (03/12/2019) <Willem Melching>
deterministic dependency order (03/12/2019) <Willem Melching>

A simple test for the generator was also added. The test verified that the generator is executed for each change made to the repository. If after running the generator there are new untracked files it means that the generator was not run after the last change made to the repository, and the test fails.

added generator test (#207) (16/12/2019) <Riccardo Biasini>

The continuous work on the OpenDBC repository for more than two years led to the creation of a complex and complete database, which was also used by people external to Comma.ai to develop other tools and applications, such as CANdevStudio. Since the tool could be useful also to developers of Openpilot, the reference to CANdevStudio was added to the README.md file in the OpenDBC repository. [34]

Add reference to CANdevStudio in README file (16/01/2020) <Remigiusz Kołataj>

With the introduction of the support to GitHub CI, the tests were moved from the Azure pipeline to GitHub Actions, allowing to execute them before every commit and ensure a higher level of quality. GitHub CI also allowed to perform a static analysis of the code and enforce the linter rules defined by *flake8*.

GitHub actions (#217) (17/02/2020) <Nelson Chen>
Library cleanup (#261) (13/05/2020) <Willem Melching>
run pre commit in ci (#268) (29/05/2020) <Willem Melching>

More fixes were then made to Cython files, improving the reliability of the setup and build processes and fixing some missing dependencies.

Build cython extensions in common setup.py (#281) (08/07/2020) <Adeeb Shihadeh>
rebuild cython extensions when python/cython/distutils change (16/07/2020) <Adeeb Shihadeh>
add packer.cc and parser.cc dependencies on their cython extensions (02/08/2020) <Adeeb Shihadeh>
parallel cythonize extension build (02/08/2020) <Adeeb Shihadeh>
cleanup parser_pyx (02/08/2020) <Adeeb Shihadeh>
fix mac build (09/09/2020) <Willem Melching>

A major fix instead was made by an external contributor, **Gregor Kikelj**, who closed an open issue (issue #222, opened by Willem Melching), about a problem with the CANDefine component not hanging when a not existing DBC file was passed as an argument.

To test the solution, the user also added a test to verify the capability of CANDefine to throw an exception in this particular case.

Fix 222 (#296) (10/09/2020) <Gregor Kikelj>

The libraries and the adopted technologies were then updated to their latest version available, to leverage the improvement in performance and reliability offered by their new versions.

C++17 (#305) (17/10/2020) <Willem Melching>

fix pycapnp (29/10/2020) <Adeeb Shihadeh>

ubuntu 16.04 -> 20.04 (#309) (29/10/2020) <Adeeb Shihadeh>

Scons cython builder (#316) (27/11/2020) <Gregor Kikelj>

Another piece of hardware that is possible to mount of the car, like the Comma Pedal, is the Zorro Steering Sensor (ZSS), developed by **Ross Fisher** (**zorrorbyte** on GitHub).

This sensor is much more precise than the standard one mounted in cars, and to make it communicate with Openpilot it was required to add a new signal.

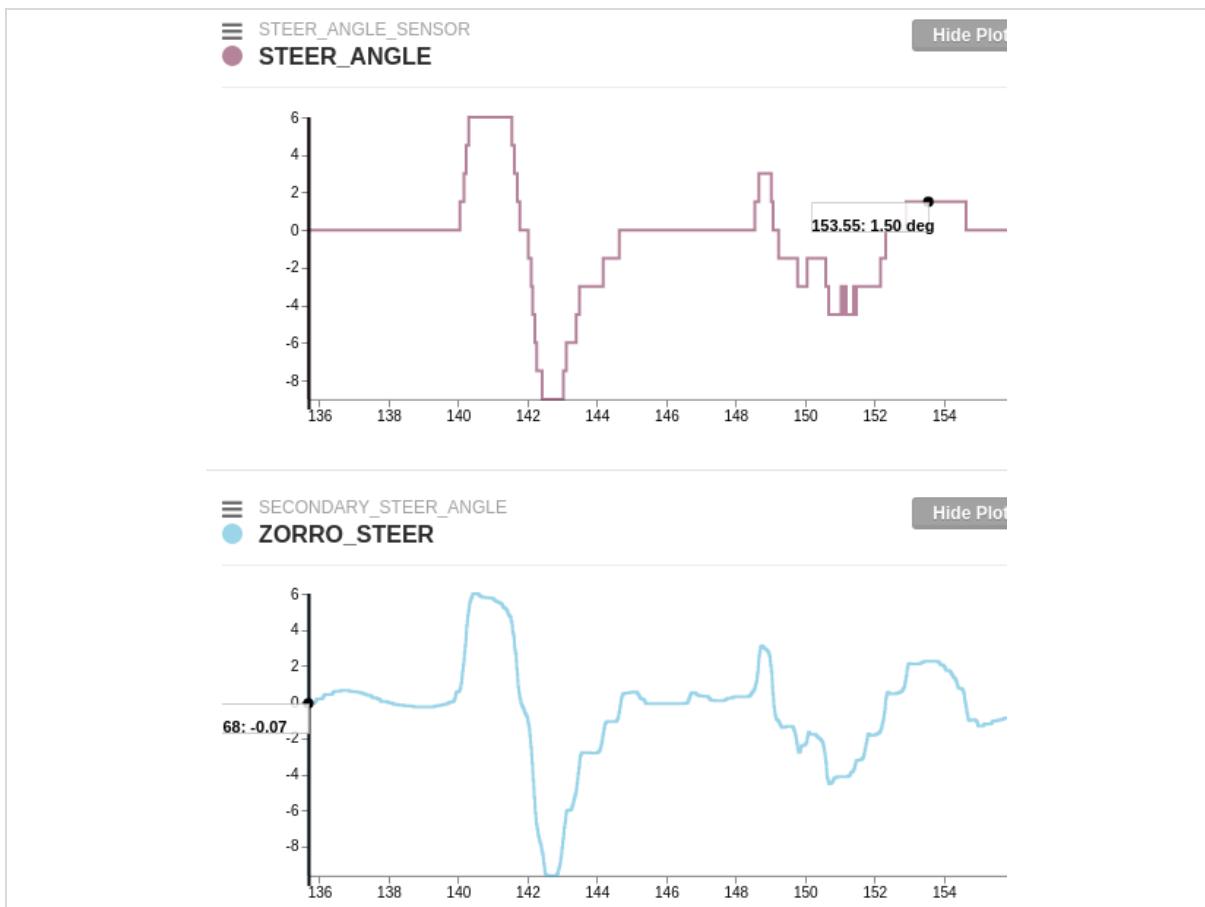


Figure 32 - Comparison between the precision of ZSS and that of the standard sensor

Add ZSS signal (#322) (03/12/2020) <Shane Smiskol>

Other optimizations also regarded the C++ implementation of the CAN parser, where many parameters were changed to static and constants to improve the performances and reliability of the component. Many security adjustments were also made to the same component.

CANParser::update_string :use const std::string& as parameter (31/01/2021) <Dean Lee>

packer.cc: const Signal& (#354) (08/03/2021) <Dean Lee>
update_string: use cached buffer (#356) (09/03/2021 16:10) <Dean Lee>
init message_states map without making copies (#355) (09/03/2021 16:11) <Dean Lee>
Modified can parser to run it from PlotJuggler (#352) (09/03/2021 16:33) <Joost Wooning>
CANParser: add option to enforce message checks (25/04/2021 08:09) <Adeeb Shihadeh>
log missing addrs that are validated (#343) (25/04/2021 11:45) <Greg Hogan>

The last change before the release of Openpilot 0.8.7 was the addition of the MIT License

add license (30/04/2021) <Adeeb Shihadeh>

Most of the community contributions focused on decoding the DBC messages of their car and contribute to the creation to the DBC database that is today available.

- **Acura:** The support to Acura cars was added for the first time on July 27th, 2017, and they are maintained almost exclusively by the Comma.ai team. [*Table 73 - Contributions to the DBC files of Acura cars*]
- **BMW:** The DBC files for the BMW models from 2008 to 2013 were added by a community member. No more modifications were made to them. [*Table 74 - Contributions to the DBC files of BMW cars*]
- **Chrysler:** Chrysler additions were made for the first time by Drew Hintz and also most of the next contributions to Chrysler cars were made by the same user. [*Table 75 - Contributions to the DBC files of Chrysler cars*]
- **Ford:** [*Table 76 - Contributions to the DBC files of Ford cars*]
- **General Motors, Chevrolet, and Cadillac:** GM, Chevrolet, and Cadillac cars share a similar platform and the messages and signals that travel on the CAN bus often are the same for these car manufacturers. The DBC files relative to General Motors' cars [*Table 77 - Contributions to the DBC files of GM cars*]. Chevrolet cars support was added thanks to the support of the community members, in particular of **Vasily Tarasov** who first added the support to Chevrolet Volt back in 2017 [*Table 78 - Contributions to the DBC files of Chevrolet cars*]. Cadillac DBC files were generated starting from the Chevrolet DBC files, since many signals are common for the two car manufacturers. Cadillac cars are maintained by the Comma.ai team, and in particular by Riccardo Biasini when he was still part of the organization [*Table 79 - Contributions to the DBC files of Cadillac cars*].
- **Honda:** The cars from Honda were the first to be supported and also today are the ones to which more changes and improvements are made, both from the community and the Comma.ai team. OpenDBC counts seventeen cars in the master branch of the repository and almost all of them are also supported by the release version of Openpilot. [*Table 80 - Contributions to the DBC files of Honda cars*]
- **Hyundai:** The support to Hyundai cars was added very early in the development of OpenDBC and was maintained almost exclusively by the Comma.ai team, but also many community members contributed to the identification and tuning of different signals. [*Table 81 - Contributions to the DBC files of Hyundai cars*]
- **Luxgen:** The Luxgen cars did not reach the release version of Openpilot, nevertheless a user added the DBC file of a Luxgen car to the master branch for anyone wanting to test it. [*Table 82 - Contributions to the DBC files of Luxgen cars*]

- **Mazda:** The support to Mazda car was introduced by an external contributor, **Jafar Al-Ghraibeh**, and also today is the only one who updates and improves the DBC files related to the cars from the manufacturer. The model that was also included in the final release is the Mazda CX-5 from 2017. [*Table 83 - Contributions to the DBC files of Mazda cars*]
- **Mercedes:** No Mercedes car is officially supported by Openpilot and included in the final release, nevertheless a user, **quillford** on GitHub, decoded many messages and signals for the Mercedes e350 of 2010. [*Table 84 - Contributions to the DBC files of Mercedes cars*]
- **Nissan:** Nissan is another example of a car manufacturer of which cars were supported thanks to the community. The two cars were added thanks to **Bugsy** and **Andre Volmensky**. [*Table 85 - Contributions to the DBC files of Nissan cars*]
- **Subaru:** The DBC of a Subaru car was included for the first time by **Jeff Palmer** and his work was resumed after a few years by **Riccardo Biasini**. After that he left the organization, more changes were then made by the community. Today, 4 Subaru cars of the five included in the master branch are also available in the release 0.8.7. [*Table 86 - Contributions to the DBC files of Subaru cars*]
- **Tesla:** The support for Tesla cars was introduced with the creation of the OpenDBC repository. Most of the corrections were then made by external contributors, however, the focus remained on cars that did not already provide an autonomous drive system and not many changes during the years were made to Tesla's DBCs. [*Table 87 - Contributions to the DBC files of Tesla cars*]
- **Toyota and Lexus:** Toyota and Lexus were the first car manufacturers to be supported by an official release of Openpilot. The two share a common platform, therefore the messages and signals that are sent on the CAN bus are often the same. These two car manufacturers count the largest number of cars supported by Openpilot, but also the majority of commits from both the Comma.ai team and external contributors [*Table 88 - Contributions to the DBC files of Toyota cars*]. Lexus cars can leverage the fact that many messages and signals are congruent to that defined in the DBC file of the Toyota manufacturer, in fact, the same file is used to generate the model-specific DBC files for the different Lexus car models. [*Table 89 - Contributions to the DBC files of Lexus cars*]
- **Volvo:** The Volvo V40 was reverse-engineered by a community member, **danielzmod** on GitHub, and required 131 commits in total, then merged with the master branch of OpenDBC. [*Table 90 - Contributions to the DBC files of Volvo cars*]
- **Volkswagen:** Volkswagen cars were added by **jessrussell** for the first time and then updated and fixed by **Jason Young**, with almost no contribution from the Comma.ai team apart from smaller fixes to the comments and indentation of the DBC files. [*Table 91 - Contributions to the DBC files of Volkswagen car*]

3.5 Panda

Panda is a universal car interface developed by Comma.ai. It connects to the ODB-II port [35] and the camera of the car, supporting the majority of communication busses adopted by many car manufacturers. In combination with OpenDBC, it allows to read and interpret all the signals traveling on the car network.

Its first version was the White Panda [*Figure 33*], it supported 3 CAN buses, 2 LIN buses, and 1 GMLAN bus. It could interface with the car through a second device, the Giraffe, which had to be connected between the car camera sensor connector and the White Panda. This interface was specific for each different car and had manual switches to change modality (engage Openpilot or normal driving). This interface was hard to build, replace and act on it, therefore was soon eliminated.



Figure 33 - The White Panda

The Grey Panda [*Figure 34*] offers the same capabilities as the White Panda, but also provides an additional GPS antenna to increase the precision of the localization-based services.



Figure 34 - The Grey Panda with its external GPS antenna

Its next version, the Black Panda, brought many technical innovations. The form factor was much more compact, and it also included the GPS antenna in the Panda enclosure. It also added the support for a fourth CAN bus, coming directly from the OBD-II port of the car and allowing a much faster and more effective fingerprinting of the car, and it replaced the OBD-II port with a USB A to connect to the Comma device and a USB C to connect to the car harness.

The Giraffe was also replaced by a much simpler system to interface with the car: the interface is now composed of a relay, capable of automatically switching the driving configuration, and the car harness, which became the only part that has to be changed from car to car (rather than replacing the whole Giraffe).



Figure 35 - The Black Panda

In its last release, the Red Panda [Figure 36], it was made a big leap in performance, thanks to a new CPU, and it was adopted the new CAN FD protocol on all the CAN bus. The CAN FD protocol allows the ECU of the car to dynamically switch to different data-rate and with larger or smaller message sizes [36]. This version is not supported by Openpilot 0.8.7, but the Comma.ai staff is already testing it on newer releases.



Figure 36 - The Red Panda

In the newest Comma devices, the Panda is already integrated into the device's enclosure, nevertheless is a separate device. Is possible to buy the Panda alone in case it is needed for testing purposes or to mount it with older devices such as the EON, but it is not recommended since the support for those devices is limited.

3.5.1 Package structure

The Panda repository provides both the source code that is executed on the device and the library to interact with it.

The software executed on the device, the drivers, firmware, safety constraints, and processes executed, the source code can be found in the `panda` directory, structured as shown in Figure 37.

The software run on the board can be found in the `/board`, while the Python library is located in folder `/python`.

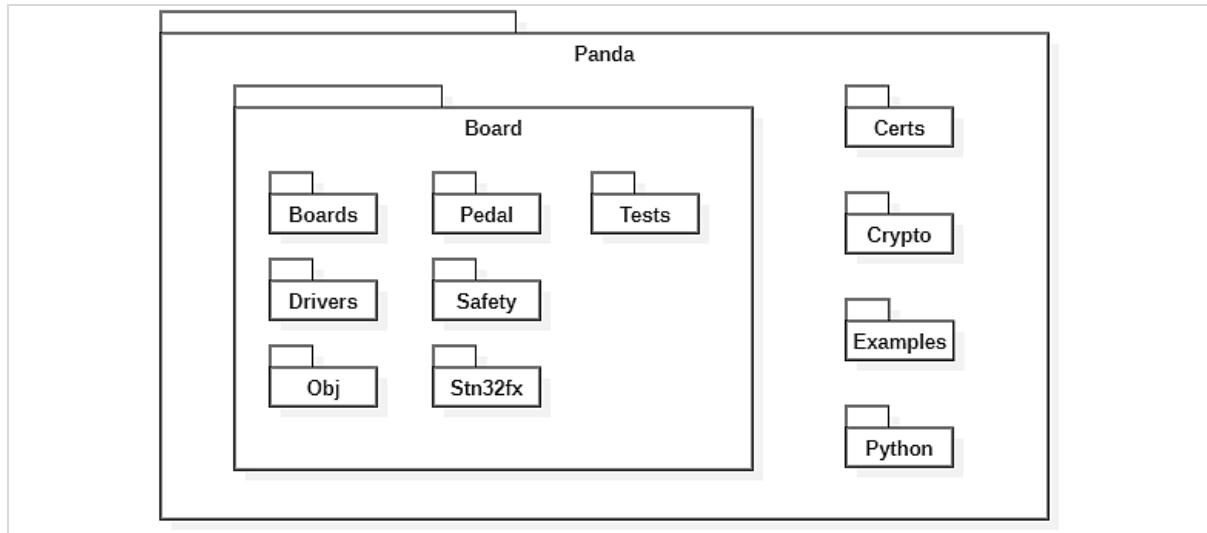


Figure 37 - Panda package diagram

The source code executed on the board, as well as the drivers for the STM32 platform, are written in C. This allows to have a high level of performance and to be executed directly on the Panda board. The Panda library, instead, is written in Python and can be used by developers to easily test the device's functionalities.

Language	Files	Blank	Comment	Code
C/C++ Header	82	4.783	6.120	31.164
Python	10	309	139	1.610
C	6	219	310	1.254
Assembly	2	220	178	696
Bourne Shell	6	9	1	35
Markdown	1	12	0	25
SUM:	107	5.552	6.748	34.784

Table 30 - Packages included in Panda main directory

3.5.2 Implementation

The `board` directory contains the source code that runs on the STM32, which is the hardware platform on which Panda is based. Its `main()` first initializes the hardware and retrieves the parameters needed, and after that executes an endless for-loop at the frequency of 8Hz.

```

int main(void) {
    init_interrupts(true); // Init interrupt table
  
```

The first operation initializes the *handler* of each of the possible interrupts that could be invoked.

Registering interrupt handlers provides the system with a way to associate an interrupt handler with an interrupt specification. The interrupt handler is called when the device might have been responsible for the interrupt. The handler has the responsibility of determining whether it should handle the interrupt and, if so, of claiming that interrupt. [37]

The timer of the interrupts is then set to 1 second.

```
void init_interrupts(bool check_rate_limit) {
    check_interrupt_rate = check_rate_limit;

    for(uint16_t i = 0U; i < NUM_INTERRUPTS; i++) {
        interrupts[i].handler = unused_interrupt_handler;
    }

    interrupt_timer_init(); // Init interrupt timer for a 1s interval
}
```

Through the APIs provided by STM32 it is possible to interact with the hardware abstraction layer (HAL) and communicate with the different peripherals. When an interrupt event occurs (or like in the previous example is initialized) the STM32 HAL APIs are used to reconfigure the low-level hardware.

If no errors occur when initializing all the interrupts, the interrupt requests (IRQ) are then disabled, by setting the IRQ mask bit of the Current Program Status Register (CPSR) to 0;

```
// shouldn't have interrupts here, but just in case
disable_interrupts();
```

disable_interrupts() disables all the interrupts by setting the I-bit of the CPSR by calling a low-level function written in C-asm, that allows inserting Assembly code in a high-level language such as C or C++, much faster than a high-level function performing the same operation.

```
_STATIC_FORCEINLINE void __disable_irq(void)
{
    __ASM volatile ("cpsid i" : : : "memory");
    // "cpsid i"-> CPS register, I-bit, disable
}
```

The clocks and the peripherals are then initialized. The Reset and Clock Controller (RCC) is in charge of the generation of all the clocks and of the system and peripherals resets, thus all its registers have to be initialized to the correct values and synchronized. The registers that compose the RCC and that the method **clock_init()** initialize are the Clock Control Register (CR), Clock Configuration Register (CFGR), Phase Lock Loop (PLL) Configuration Register (PLLCFGR). It also initializes the flash memory instruction and data cache and wait state.

Similarly, the **peripherals_init()** method initializes the clock register and reset register of all the connected peripherals.

A check for an external serial debugging device is made. The device is detected through the General Purpose I/O (GPIO) interface, by setting the GPIO mode to the pull-down configuration (which means that the load is placed between the GPIO and the ground) and checking the input on the third pin of the GPIO.

The method `detect_board_type()` then initializes the configuration that is specific for the different types of board. The board can be any of Panda hardware (from White Panda to Black Panda), but also a Comma Pedal, a device specifically developed to improve the precision of the accelerator pedal for some models of cars.

The last component to be initialized is the Analog to Digital Converter (ADC), through the method `adc_init()`.

```
// init early devices
clock_init();
peripherals_init();
detect_external_debug_serial();
detect_board_type();
adc_init();
```

If the hardware type, retrieved by the method `detect_board_type()`, is unknown, then the program will wait endlessly, until the device is turned off. Otherwise, the board details are logged.

```
// check for non-supported board types
if(hw_type == HW_TYPE_UNKNOWN) {
    puts("Unsupported board type\n");
    while (1) { /* hang */ }
}

puts("Config:\n");
puts(" Board type: "); puts(current_board->board_type); puts("\n");
puts(has_external_debug_serial ? " Real serial\n" : " USB serial\n");
```

The selected board is then initialized. The initialization method is different for each of the boards, since each board may have different peripherals connected to different pins.

```
// init board
current_board->init();
```

Panda devices provide a Floating-Point Unit (FPU) that can execute operations on floating-point numbers. It is enabled by setting the coprocessor access mode to full access, and that is done by acting on the coprocessor registers CP10 and CP11.

```
// panda has an FPU, let's use it!
enable_fpu();
```

If an external debugger is connected to the Universal Asynchronous Receiver/Transmitter (UART) of the board, it is initialized by registering and enabling the interrupts corresponding to the configuration selected. All the different hardware platform versions on which the Panda devices can be built have the possibility to connect an external device through the

UART. SMT32 boards have various UART channels, which include USART1, USART2, and USART3, which allows synchronous and asynchronous data transfer, and UART5 for only asynchronous data transfer.

All the channels can be accessed by two specific pins, identified with TX and RX. Data are transmitted at the specified rate, defined as the *baud rate*. [38]

```
// enable main uart if it's connected
if (has_external_debug_serial) {
    uart_init(&uart_ring_debug, 115200);
}
```

If a GPS is connected to the board's UART is also initialized by registering and enabling the interrupts of the GPS antenna and setting the correct gauge rate.

```
if (current_board->has_gps) {
    uart_init(&uart_ring_gps, 9600);
} else {
    // enable ESP uart
    uart_init(&uart_ring_gps, 115200);
}
```

If the car on which the Panda device is installed adopts a LIN bus, the LIN mode is first enabled by initializing the Control Buffer and their interrupts have to be also registered and enabled.

```
if(current_board->has_lin){
    // enable LIN
    uart_init(&uart_ring_lin1, 10400);
    USART5->CR2 |= USART_CR2_LINEN;
    uart_init(&uart_ring_lin2, 10400);
    USART3->CR2 |= USART_CR2_LINEN;
}
```

The timer for the control register, the event generation register, and the prescaler are initialized and set to the enabled state.

```
microsecond_timer_init();
```

The safety mode is then initialized to the silent configuration: the replay is set to the passthrough mode, disabling the replay functionalities by default, and the CAN mode is set to normal.

```
// init to SILENT and can silent
set_safety_mode(SECURITY_SILENT, 0);
```

The component which enables the physical link between the bus is the *transceiver*. This component enables to transmit and receive data, converting data coming from the CAN controller into electrical signals and transmitting them on the bus [39]. It is enabled and configured specifically for the board with which the transceiver has to communicate.

```
// enable CAN TXs
current_board->enable_can_transceivers(true);
```

The ticker timer is then initialized. A ticker is a timer that is used to recurrently call a function at a predefined rate, in the specific case at 8Hz. The registered interrupt will be in charge of calling the function tick_handler() at the rate defined by TICK_TIMER_IRQ. tick_handler() is in charge of managing the sirens, the led state, and the fan spin.

```
// 8Hz timer
REGISTER_INTERRUPT(TICK_TIMER_IRQ,
                   tick_handler,
                   10U,
                   FAULT_INTERRUPT_RATE_TICK)
tick_timer_init();
```

Like for the other peripherals, the interrupt for the USB port has to be registered. The corresponding handler, OTG_FS_IRQHandler(), is called whenever an event coming from the USB port occurs. It first disables the Nested Vector Interrupt Control (NVIC) register, the purpose of which is to prioritize the interrupts, to allow to trigger directly the interrupt coming from the USB. The handler then verifies what event triggered the interrupt and acts accordingly.

```
// enable USB (right before interrupts or enum can fail!)
usb_init();
```

The registered interrupts are then enabled, and this is done by clearing the I bit of the CPSR. This function performs the exact opposite operation of the disable_interrupts() method.

```
puts("**** INTERRUPTS ON ****\n");
enable_interrupts();
```

After that all the parameters, interrupts, and peripherals are enabled, the program starts the execution of an endless loop, with a counter *cnt* that keeps track of the number of executions.

At each loop there is a check that verifies if the power saving mode is enabled.

```
uint64_t cnt = 0;
for (cnt=0;;cnt++) {
    if (power_save_status == POWER_SAVE_STATUS_DISABLED) {
        #ifdef DEBUG_FAULTS
        if(fault_status == FAULT_STATUS_NONE) {
        #endif
```

If it is not, the device's led is turned on and off with a fading effect, to indicate that there are no problems during the execution.

```
uint32_t div_mode = ((usb_power_mode == USB_POWER_DCP) ? 4U : 1U);

// useful for debugging, fade breaks = panda is overloaded
for(uint32_t fade = 0U; fade < MAX_LED_FADE; fade += div_mode){
    current_board->set_led(LED_RED, true);
    delay(fade >> 4);
    current_board->set_led(LED_RED, false);
    delay((MAX_LED_FADE - fade) >> 4);
}

for(uint32_t fade = MAX_LED_FADE; fade > 0U; fade -= div_mode){
    current_board->set_led(LED_RED, true);
    delay(fade >> 4);
    current_board->set_led(LED_RED, false);
    delay((MAX_LED_FADE - fade) >> 4);
}
```

If problems occur during the debug, the device's led will turn on and off without any fading effect.

```
#ifdef DEBUG_FAULTS
} else {
    current_board->set_led(LED_RED, 1);
    delay(512000U);
    current_board->set_led(LED_RED, 0);
    delay(512000U);
}
#endif
```

If the device is in power saving mode, it is put in a sleeping state called Wake from Interrupt (WFI), this means that it will stay in the sleeping state until an interrupt that has already been registered and enabled occurs.

```
} else {
    __WFI();
}
return 0;
}
```

In the *panda/python* directory is located the Python implementation of the Panda library, through which is possible with the device. In particular, the class Panda [*Figure 38*] provides the methods to connect and send data to the Panda device. Through a Panda object is possible to communicate to Openpilot the command that the user input through the car controls.

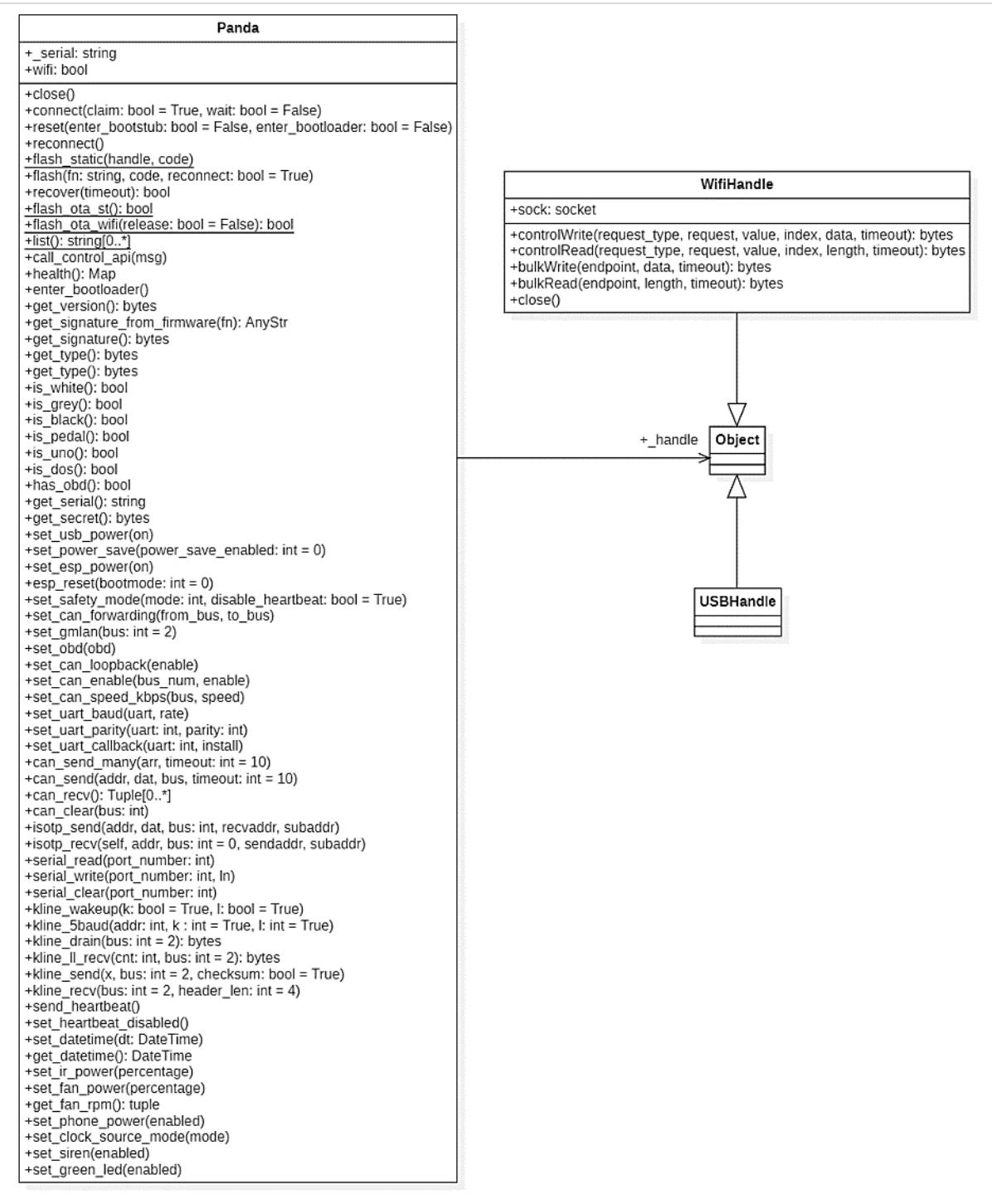


Figure 38 - Panda class diagram. The handle attribute can be either USB handle (standard from usblib) or a custom Wi-Fi handle, according to the connection method.

Openpilot can connect to Panda through the method **connect()**. It first retrieves a list of the connected devices through the method **list()**. USB devices are assigned a logical handle by operating systems when they are first plugged in. This process is known as enumeration. Once a USB device has been enumerated, it is ready for use by the host computer software. For the host application software to communicate with the USB device, it must first obtain the handle assigned to the USB device during the enumeration process. The handle can be obtained using an open function along with some specific information about the USB device. Information that can be used to obtain a handle to a USB device includes, serial number,

product ID, or vendor ID [40]. By checking the vendor identifier and product identifier of the device, the Panda device can be discovered.

```
if device.getVendorID() == 0xbbaa and device.getProductID()
    in [0xddcc, 0xddee]
```

Its handle is then saved into the `_handle` variable and a Panda object can use it to read and write information to and from the Panda device.

```
if self._serial is None or this_serial == self._serial:
    self._serial = this_serial
    print("opening device", self._serial, hex(device.getProductID()))
    self.bootstub = device.getProductID() == 0xddee
    self._handle = device.open()
```

The methods `close()` and `reconnect()` use the functionalities provided by the device handle to respectively close and restore (by closing and connecting again the handle) the connection to the handle. The method `reset()` sends a reset request to the panda device and its default parameters are restored.

A series of methods allow flashing a firmware for the Panda device. Different possibilities are available: `flash_ota_st()` and `flash_ota_wifi()` allow to make OTA update, without the need of flashing the whole firmware from scratch; `flash_static()` and `flash()` both install the newest version of the firmware; `recover()` resets the device, it puts it into DFU (device firmware update or recovery mode) and flashes again the firmware.

To check global status statistics of the device is possible to call the method `health()`, which reads a series of parameters about the status of the device and returns them in form of a Map.

```
dat = self._handle.controlRead(Panda.REQUEST_IN, 0xd2, 0, 0, 44)
a = struct.unpack("<IIIIIIIBBBBBBBHBBB", dat)

return {
    "uptime": a[0],
    "voltage": a[1],
    "current": a[2],
    "can_rx_errs": a[3],
    "can_send_errs": a[4],
    "can_fwd_errs": a[5],
    "gmlan_send_errs": a[6],
    "faults": a[7],
    "ignition_line": a[8],
    "ignition_can": a[9],
    "controls_allowed": a[10],
    "gas_interceptor_detected": a[11],
    "car_harness_status": a[12],
    "usb_power_mode": a[13],
    "safety_mode": a[14],
    "safety_param": a[15],
    "fault_status": a[16],
    "power_save_enabled": a[17],
    "heartbeat_lost": a[18],
}
```

Different methods are available to get the device's basic information. They allow retrieving the Panda version, signature, or hardware used.

`get_version()`, `get_signature_from_firmware()`, `get_signature()`, `get_serial()`, `get_secret()`, and `has_obd()` return the devices' basic information, such as the version of the firmware or the serial number. Other getters are also available to retrieve other kind of details such as the fan rpms (`get_fan_rpm()`) and the system datetime (`get_datetime()`).

The hardware version of Panda can be retrieved using the methods `is_white()`, `is_grey()`, `is_black()`, `is_pedal()`, `is_ultimo()`, and `is_dos()`, which return a boolean indicating if the hardware is the one for what is being checked for.

The library also allows setting different parameters of the Panda device itself, such as the speed of the different modalities, the transfer speed of data, or even resetting the device.

- `set_usb_power()`: change the USB power mode, which can be USB disabled, standard USB charging port, or Charging Downstream Port (CDP) and Dedicated Charging Port (DCP), which can supply a higher current.
- `set_fan_power()`: specify the percentage to which the device fan should spin.
- `set_power_save()`: enable or disable the power-saving mode.
- `set_esp_power()`: enable or disable the ESP of the car.
- `esp_reset()`: reset the ESP configuration.
- `set_siren()`: enable or disable the device's siren.
- `set_safety_mode()`: select a safety model for a specific car manufacturer.
- `set_can_forwarding()`: forward a CAN message to the specified bus.
- `set_gmlan()`: leverage the CAN2 and CAN3 modules to convert CAN data, and in particular GMLAN (General Motors Local Area Network) data, into analog outputs and send them over the CAN2 bus.
- `set_obd()`: send data over the OBD port.
- `set_can_loopback()`: enable or disable the CAN loopback mode, which sends the CAN messages back to the source to test the network reliability.
- `set_can_enable()`: set the CAN transceiver enable pin. Since the board support bidirectional communication, each input pin has also to be associated with an enable pin.
- `set_can_speed_kbps()`: set the bandwidth of the bus, in kbps.
- `set_uart_baud()`: set the baud rate of the UART.
- `set_uart_parity()`: UART can have an optional parity bit, through this method it can be defined if use an even or odd parity bite or no parity bit.
- `set_uart_callback()`: define a callback function for the specified UART peripheral.

The main functionalities regard sending and receiving data over the CAN bus. `can_send()` permits to send data in hexadecimal format to the specified bus, while `can_recv()` decodes the data arriving from the Panda device. `can_clear()` drains all the messages still in queue.

Panda also supports ISO-TP, which is a transport protocol defined in the ISO-Standard ISO15765-2. This transport protocol extends the limited payload data size for classical CAN (8 bytes) and CAN-FD (64 bytes) to theoretically 4 GB. ISO-TP segments the data packets into small fragments depending on the payload size of the CAN frame. The method `isotp_recv()` reads the data from the receive FIFO of the context, while `isotp_send()` sends data to a peer that listens to the specified address. [41]

The library also allows leveraging the board UART to send and receive data synchronously by using the methods `serial_read()` and `serial_write()`.

Also in this case `serial_clear()` allows to drain off the messages buffer.

Other than CAN protocol, in the past, it was widely used the K-Line protocol, and even if it does not play a substantial role as it did in the past, it is still widely used by many car manufacturers, that yet implement their ECUs using K-Line technology.

The K-Line is suitable for both on-board and off-board diagnostics, and it offers two special initialization patterns: `kline_wakeup()` is based on a 10,400 baud standard, and it sends a wake-up pattern. There is also what is known as the 5-Baud Init pattern, implemented by the method `kline_5baud()`, in which the system sends an address byte at five baud, and the receiver detects this slow transmission rate. Data are sent through `kline_send()` and received through `kline_recv()` at a standard transmission rate of 10.400 baud and speeds up to 115.200 baud for such purposes as programming of flash memories [42]. The K-Line buffer can be drained through the method `kline_drain()`.

Panda devices send periodical signals to indicate their normal operation: these signals, commonly named *heartbeat*, are sent through the method `send_heartbeat()`. The Panda heartbeat can be disabled through the method `set_heartbeat_disabled()`: in this case, the checks for the device's heartbeat are automatically disabled and can be re-enabled by sending a new heartbeat signal.

3.5.3 Usage

The Python implementation of the Panda library is used by *selfdrive* to perform operations at start time or that don't require a high computation time, while a C++ version of the Panda library is available in *selfdrive*, which allows achieving a higher level of performance.

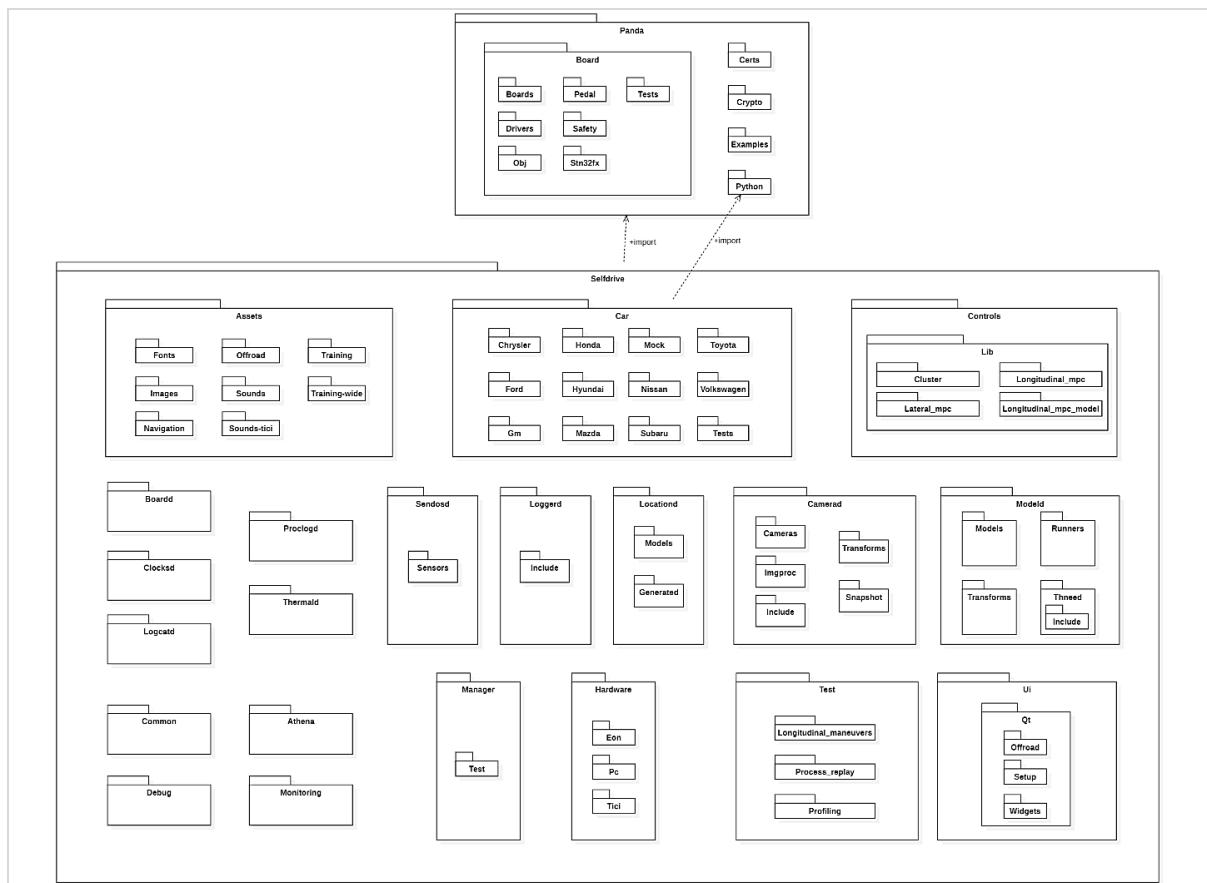


Figure 39 - Import dependencies between selfdrive and panda

One example of the usages of the Python library is to have an easy way to retrieve the signature of the firmware installed on the device.

```
def get_expected_signature() -> bytes:
    try:
        return Panda.get_signature_from_firmware(PANDA_FW_FN)
    except Exception:
        cloudlog.exception("Error computing expected signature")
        return b""
```

It is also used to instantiate a Panda object, or a PandaDFU object in the case in which the device is in recovery mode.

```
while True:
    # break on normal mode Panda
    panda_list = Panda.list()
    if len(panda_list) > 0:
        cloudlog.info("Panda found, connecting")
        panda = Panda(panda_list[0])
        break

    # flash on DFU mode Panda
    panda_dfu = PandaDFU.list()
    if len(panda_dfu) > 0:
        cloudlog.info("Panda in DFU mode found, flashing recovery")
        panda_dfu = PandaDFU(panda_dfu[0])
        panda_dfu.recover()
```

The Panda instance can then be used to check that the firmware is updated and that the heartbeat is always sound.

```
health = panda.health()
if health["heartbeat_lost"]:
    cloudlog.event("heartbeat lost", deviceState=health)
```

3.5.4 Testing

The library and its functionalities were first tested through CircleCI and then the tests were migrated to GitHub Actions. In particular, the jobs defined allow to perform a static analysis of the source code and check that the safety logic constraints needed to be compliant with the Federal Motor Vehicle Safety Standards (FMVSS) are respected. Panda, as well as the other safety-relevant components of Openpilot, has to observe the ISO26262 guidelines, as well as strict coding guidelines that include *MISRA C:2012* for C source code and *flake8* and *pylint* linters for Python source code.

The workflows defined in GitHub Actions allow automating the check of both the drivers and the safety constraints, respectively in the workflow *drivers* and *tests*.

The workflow **drivers**, after setting up the environment and solving all the dependencies, try to carry out the installation of **SocketCAN**, which is the driver for the Linux kernel that provides a socket interface to user space applications and builds upon the Linux network layer.

```
name: Build socketcan driver
run: |
  cd drivers/linux
  make link
  make all
  make install
```

SocketCAN allows using Panda with tools such as *can-utils*, which contains userspace utilities for the Linux SockenCAN subsystem. The utilities include basic tools to display, record, generate and replay CAN traffic. If the job is carried out till the end and completes the installation of panda after that of the driver it means that there are no issues with SocketCAN.

The workflow **tests** run the static analysis of the source code and verify that the safety constraints are respected.

```
safety:
steps:
- name: Run safety tests
  run: |
    $RUN "cd /tmp/Openpilot && \
          scons -c && \
          scons -j$(nproc) -i opendbc/ cereal/ && \
          cd panda/tests/safety && \
          ./test.sh"
```

Safety tests are executed for each of the supported platforms and for each car manufacturer and make sure that the test fails in the case of unknown hardware.

```
for hw_type in 0 1 2 3 4 5 6
do
  echo "Testing HW_TYPE: $hw_type"
  HW_TYPE=$hw_type python -m unittest discover .
done
```

The *discovery* functionality provided by the *unittest* framework allows executing all the Test Case in the same directory of the script. All the tests inherit the properties and behaviors of PandaSafetyTestBase, which is an instance of *unittest.TestCase*, is inherited by the common safety test cases *PandaSafetyTest*, *TorqueSteeringSafetyTest*, and *InterceptorSafetyTest*. These test cases define all the standard tests that are shared for all the safety models.

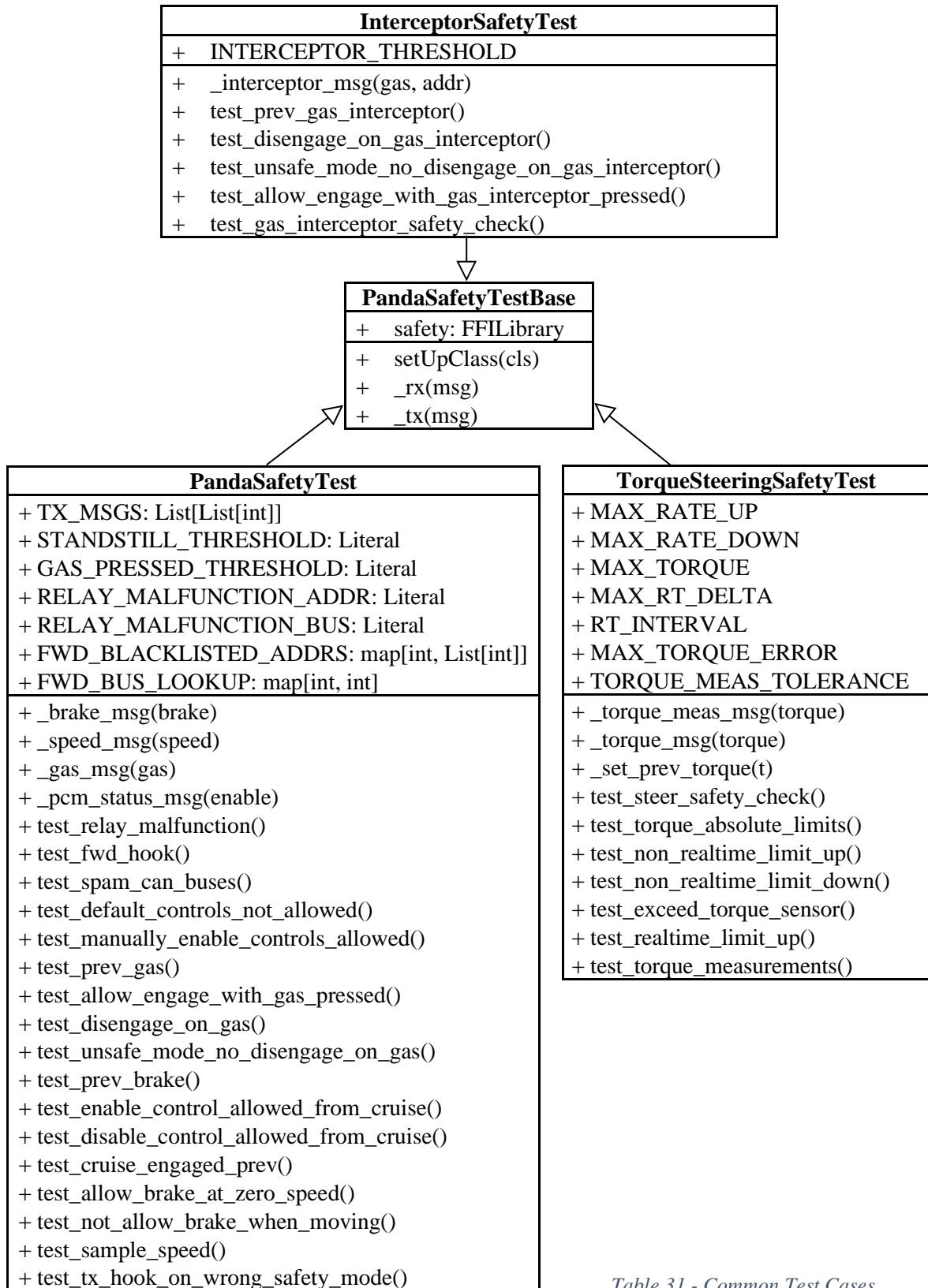


Table 31 - Common Test Cases

Each car manufacturer-specific test case also includes the test that inherits from its parent common classes, and also defines tests that are specific for the different types of cars.

For instance, the test case that verifies the safety constraints for Chrysler cars [Table 32] defines, besides PandaSafetyTest and TorqueSteeringSafetyTest common tests, three more tests that check the ability of Panda to engage or disengage only when needed.

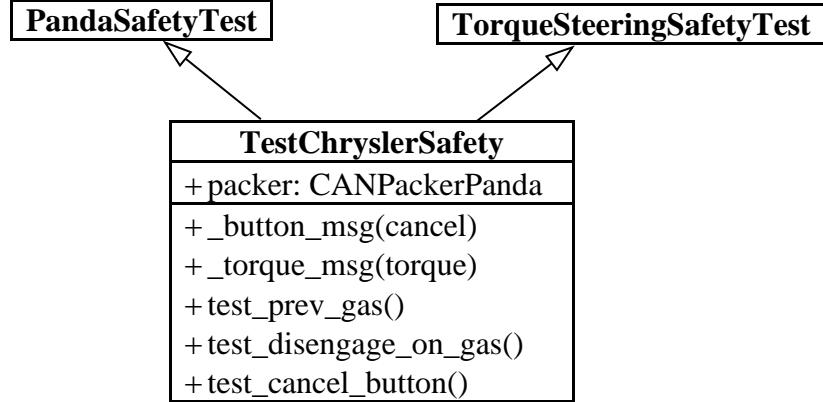


Table 32 - TestChryslerSafety

Similarly, GM safety tests [Table 33] ensure that values such as the torque constraints are never exceeded.

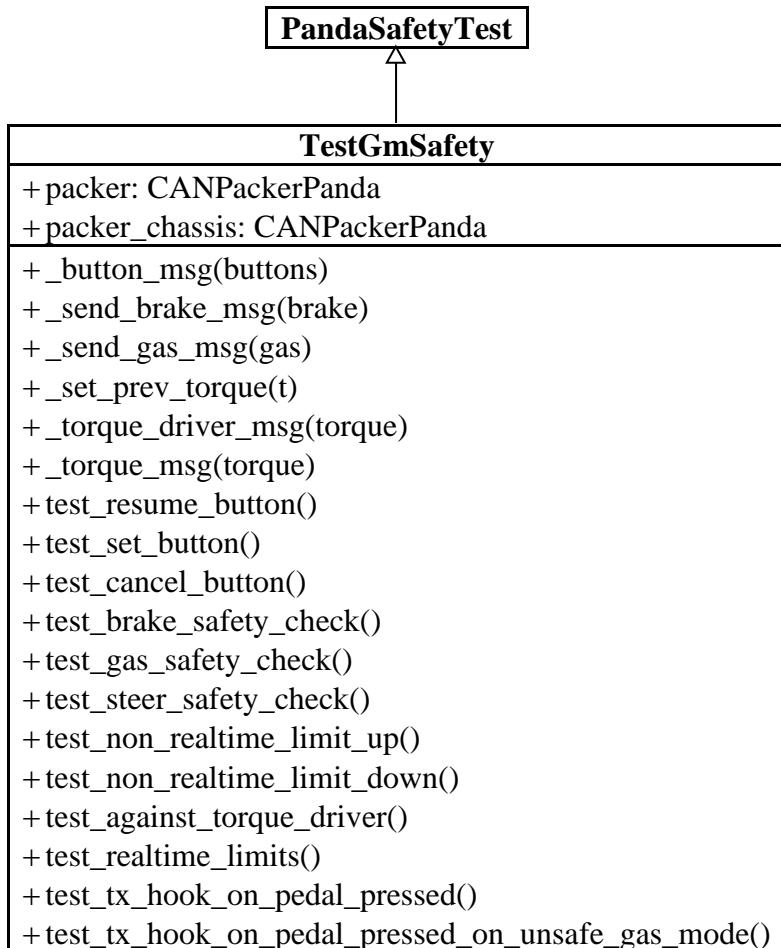


Table 33 - TestGmSafety Test Case

The test cases regarding Toyota cars [Table 34] consider all the versions supported, which means that there are different tests for the Toyota cars that have a Nidec camera and for those that have a Bosch camera. Also, there is the distinction between Panda devices connected through harness and Giraffe, since each configuration uses different values for the parameters used by Panda and has to be initialized in different ways.

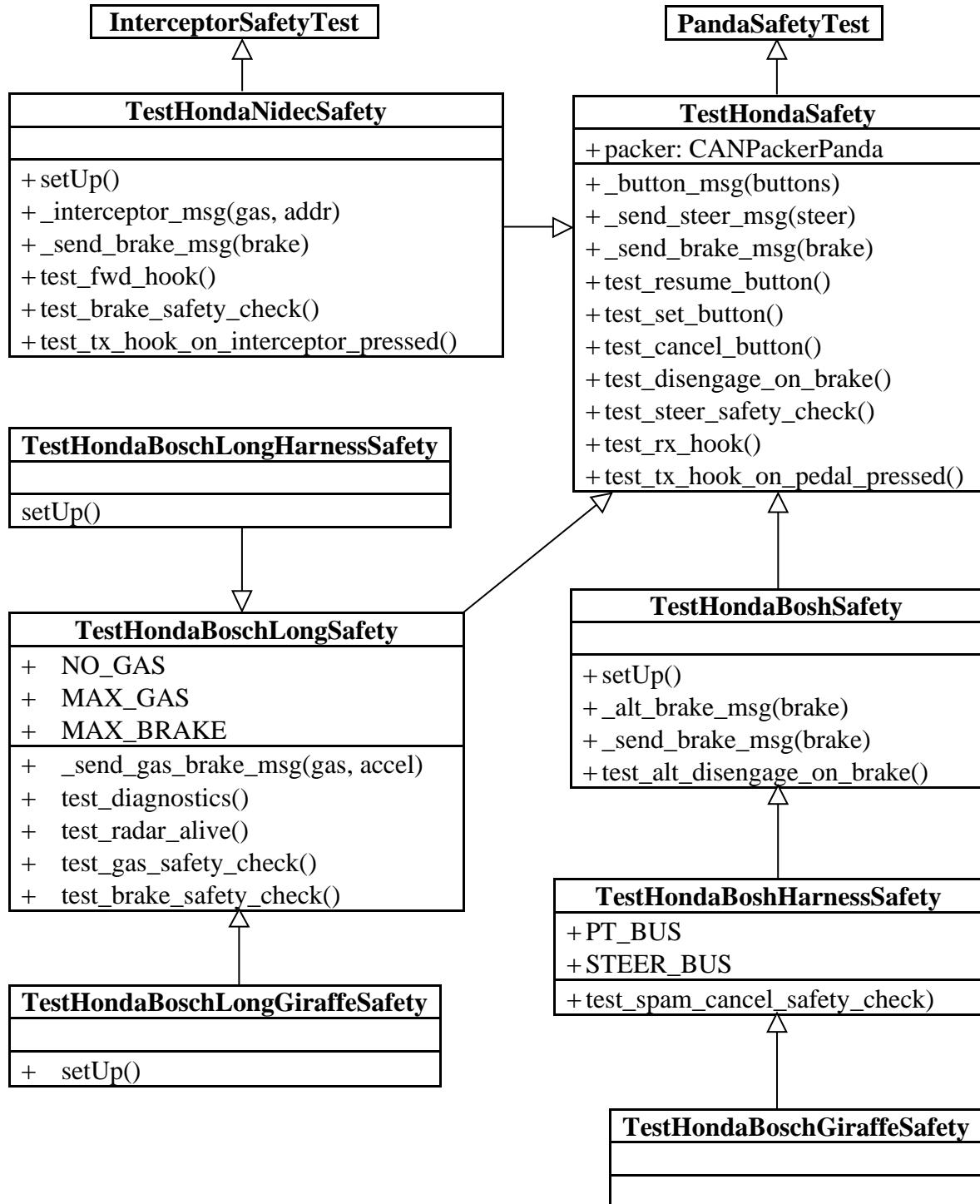


Table 34 - Honda Test Cases

Hyundai test cases [*Table 35*], like for Chrysler ones, verify that the torque constraints are respected. Moreover, also the longitudinal tuning functionalities are tested, and these include the calculation of the acceleration that the car can have, checking also by means of the radar that is safe to accelerate or decelerate.

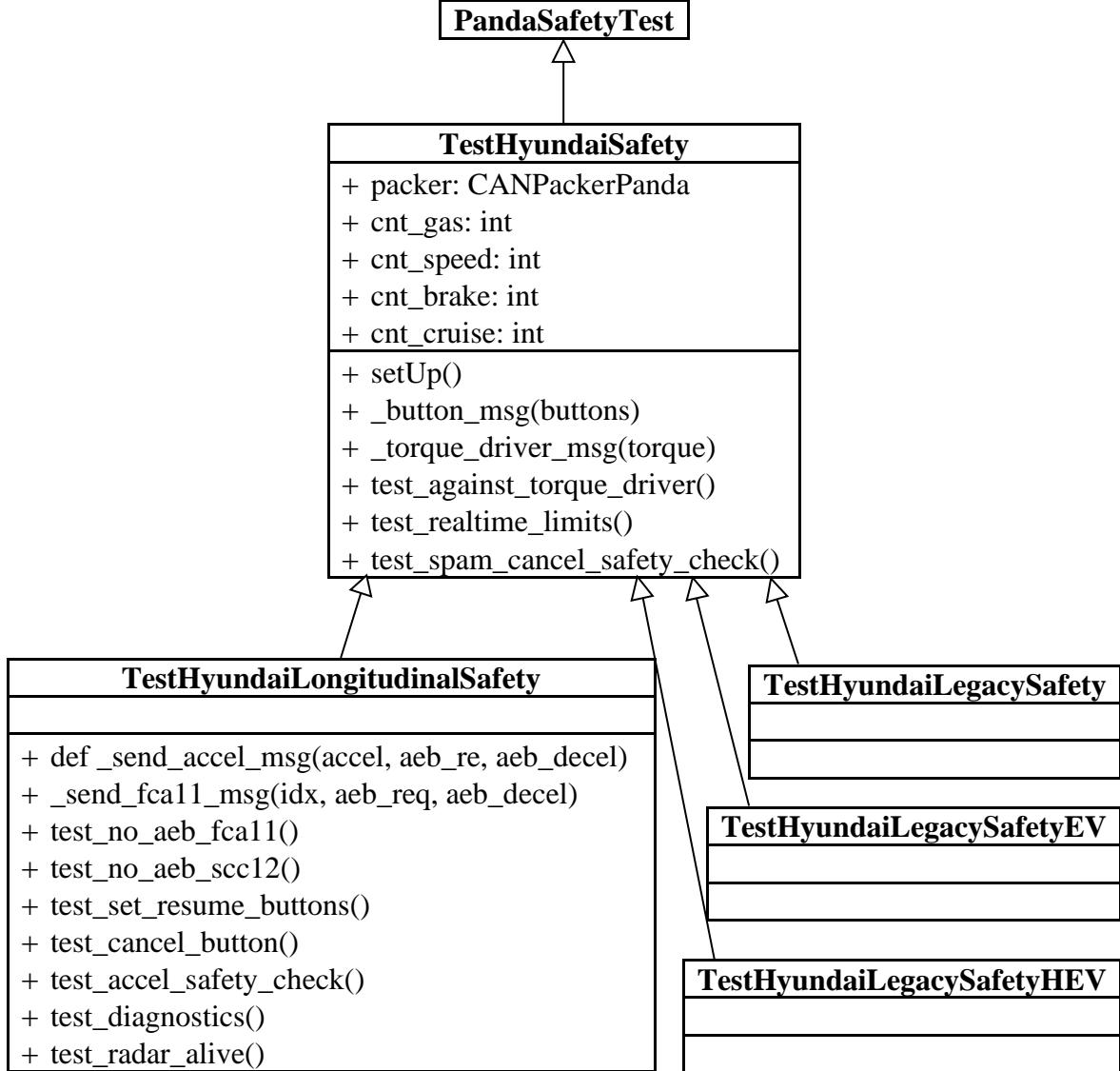


Table 35 - Hyundai Test Cases

Mazda test case [*Table 36*] does not differ from the others and, above the common safety tests, also checks the torque constraints.

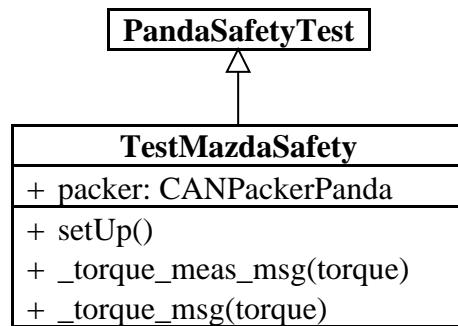


Table 36 - TestMazdaSafety Test Case

The Nissan safety test [Table 37] also includes safety checks about the Lane Keeping Assist System (LKAS) and Adaptive Cruise Control (ACC), ensuring that when the two systems are enabled the angle command rate limit is always enforced.

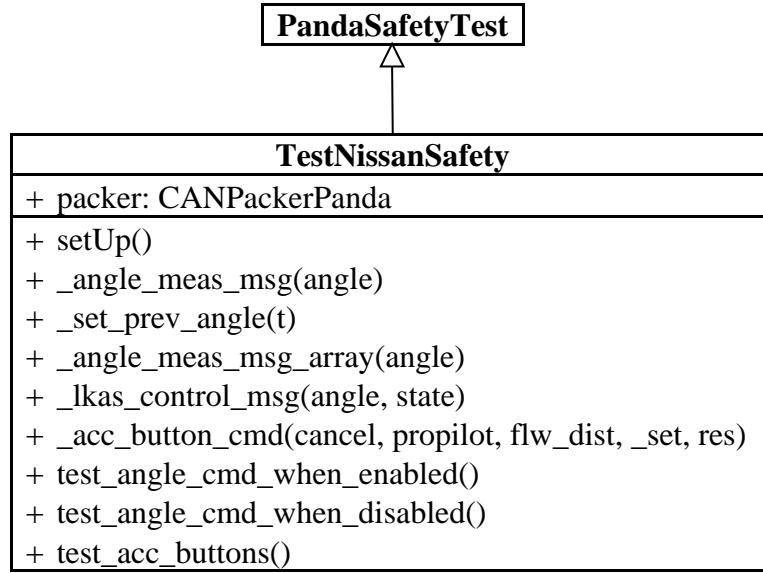


Table 37 - *TestNissanSafety Test Case*

TestSubaruLegacySafety test case [Table 38] verifies the compliance with the safety constraints for old Subaru cars that are still supported by Openpilot. In particular, the torque values limits are checked, ensuring that the CAN message is delivered only if the torque values fall within the limits.

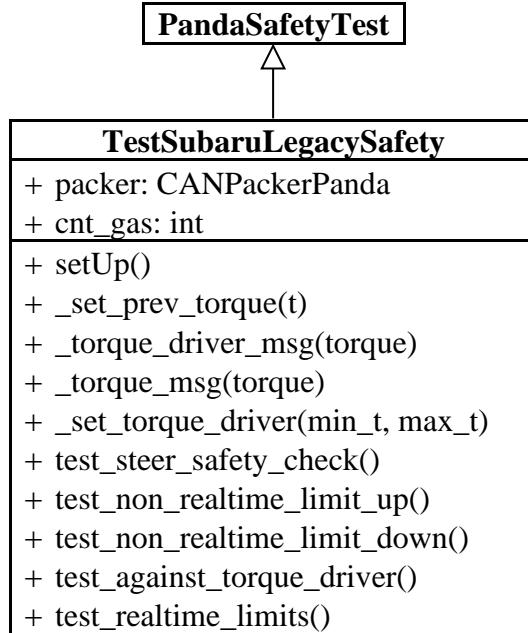


Table 38 - *TestSubaruLegacySafety Test Case*

The compliance with the safety constraints of the newer models of Subaru (manufactured after 2017) is tested by the PandaSafetyTest test case [Table 39]. Even if the differences are the same for both the safety and the newer models of Subaru cars two test cases are needed because the parameters have to be initialized differently.

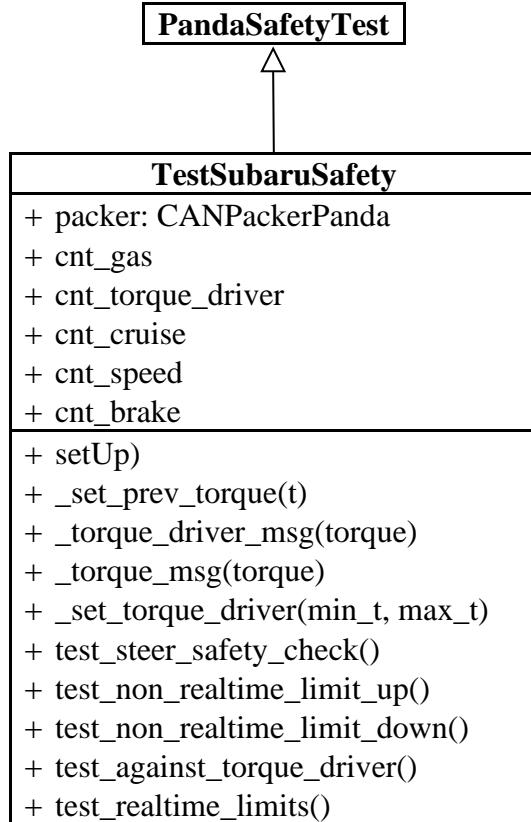


Table 39 - TestSubaruSafety Test Case

The safety tests for Tesla check that the messages regarding the different components, including Tesla's Autopilot, lead to the expected status.

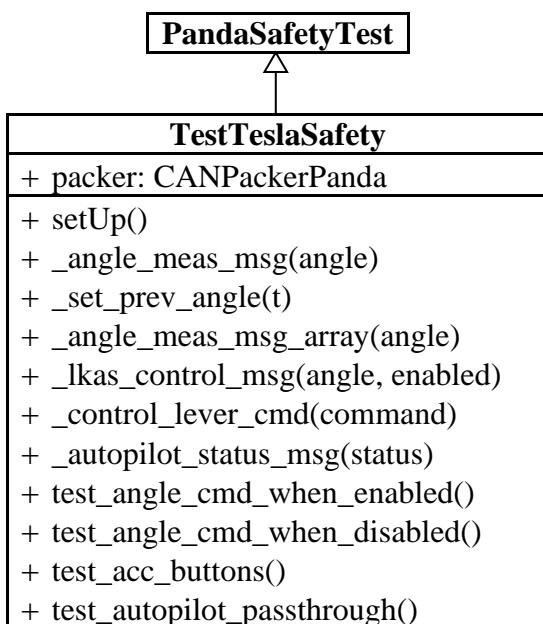


Table 40 - TestTeslaSafety Test Case

Toyota test case comprehends all the common tests defined, together with more validation of the acceleration and steering parameters that can be sent to Panda.

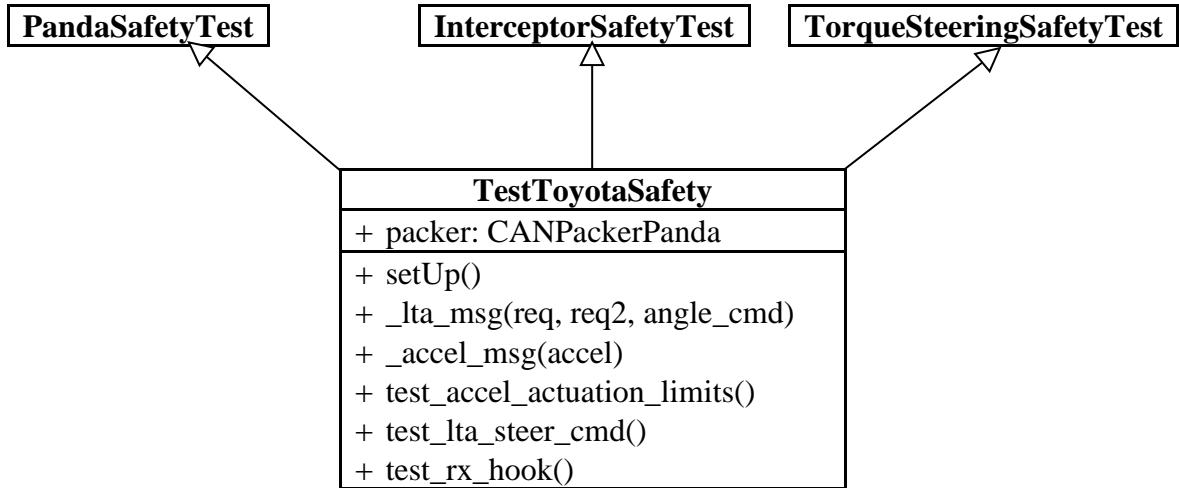


Table 41 - TestToyotaSafety Test Case

Volkswagen cars can be categorized into cars that adopt the MQB (Modularer Querbaukasten) platform, and those that adopt the PQ platform (also known as New Small Family platform (NSF)).

MQB is a system that consists of a main core that is the base for many other platforms of different cars. The test case for this car platform [*Table 42*] does not differ from the test case for the other platform of Volkswagen, but it has to initialize the destination bus to which send the messages differently.

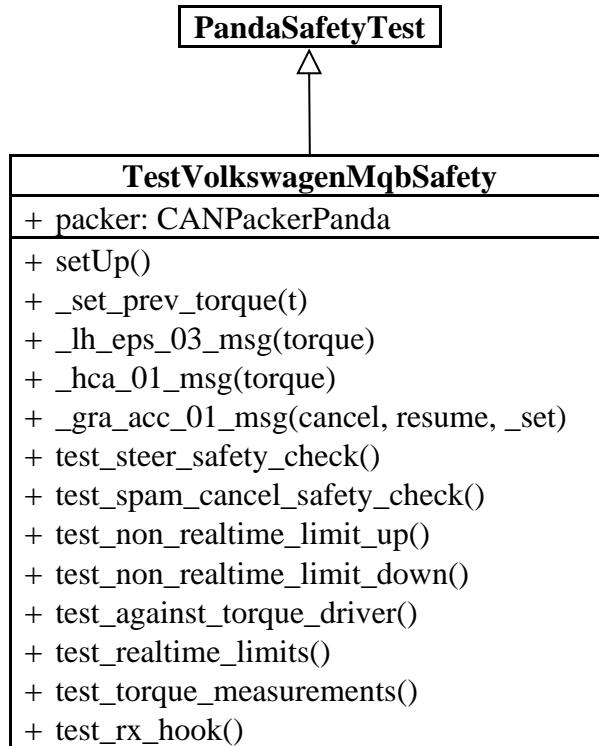


Table 42 - TestVolkswagenMqbSafety Test Case

The PQ platform is destined for a range of ultra-compact city cars and is substantially different from the MQB platform. Its test case, however, is specular to that of the other platform, with only the variables representing the destination pins for the messages initialized differently.

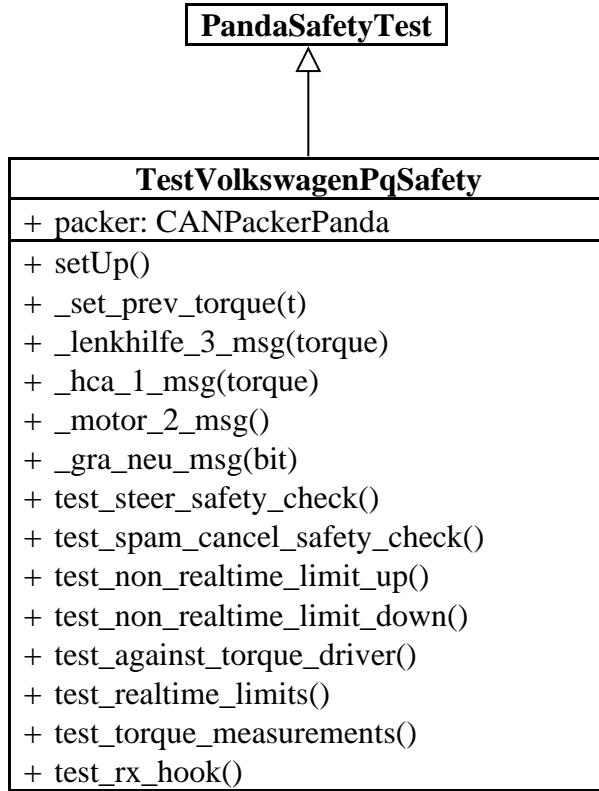


Table 43 - *TestVolkswagenPqSafety* Test Case

All these test cases and the relative tests defined within are executed for each version of the Panda hardware. Overall, the 501 tests defined are executed for the seven hardware types available.

```

Testing HW_TYPE: 4
scons: Entering directory '/tmp/Openpilot/panda'
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
scons: 'tests/safety' is up to date.
scons: done building targets.
[...]
Ran 501 tests in 3.349s

Testing HW_TYPE: 5
scons: Entering directory '/tmp/Openpilot/panda'
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
scons: 'tests/safety' is up to date.
scons: done building targets.
[...]
Ran 501 tests in 3.527s

```

The other type of tests executed by GitHub Actions are the *safety replay* tests. The different route logs, acquired by various cars, are replayed by resending all the CAN messages of the drive (through TX bus) and recording the outcome of each message (arriving on RX bus).

```
replaying 2425568437959f9d|2019-12-22--16-24-37.bz2 with safety mode 1  
and param 0  
  
RX  
total rx msgs: 1271442  
invalid rx msgs: 0  
invalid addrs: set()  
https://github.com/commaai/panda/runs/3648263901?check\_suite\_focus=true - step:4:26  
TX  
total Openpilot msgs: 89954  
total msgs with controls allowed: 3046  
blocked msgs: 45  
blocked with controls allowed: 0  
blocked addrs: {506}
```

The other jobs executed through GitHub Actions make a static analysis of the source code. C code is checked against the Misra-C:2012 regulations. MISRA-C:2012 contains 143 rules and 16 "directives", each of which is classified as mandatory, required, or advisory. They are separately classified as either Single Translation Unit or System. Additionally, the rules are classified as Decidable or Undecidable [43]. The output of these verifications shows which rules are violated.

```
PANDA F4 CODE  
Checking ../../board/main.c ...  
Checking ../../board/main.c:  
PANDA=1;STM32F4=1;CAN3=1;UID_BASE=1;STM32F423xx...  
Checking ../../board/main.c.dump...  
Checking ../../board/main.c.dump, config  
PANDA=1;STM32F4=1;CAN3=1;UID_BASE=1;STM32F423xx...  
  
MISRA rules violations found:  
Undefined: 466  
  
MISRA rules violated:  
misra-c2012-9.3 (-): 1  
misra-c2012-11.4 (-): 10  
misra-c2012-11.5 (-): 11  
misra-c2012-19.2 (-): 8  
misra-c2012-20.1 (-): 1  
misra-c2012-20.7 (-): 11  
misra-c2012-20.10 (-): 10  
misra-c2012-21.1 (-): 414
```

Python source code is checked by using different linters, including *flake8* and *pylint*.

Check python ast.....	Passed
Check Yaml.....	Passed
Check for merge conflicts.....	Passed
Check for broken symlinks.....	(no files to check) Skipped
mypy.....	Passed
flake8.....	Passed
pylint.....	Passed

3.5.5 Development and community contribution

The Panda repository was one of the first to be published when Openpilot was first released.

initial commit (07/04/2017) <George Hotz>

The Python library was developed only in a second moment, in the beginning, only the code that had to be run on the boards was included.

Created python package and implemented industry best practices. Supports Python 2 and 3 (to the best of my testing ability at the time) (14/06/2017) <Jessy Diamond Exum>

When GitHub CI was not available yet, tests were manually executed. The available test included a loopback test, a standalone test, and a throughput test.

Fix some tests to not explode on python3. (28/06/2017) <Jessy Diamond Exum>

Repairing panda tests. (28/06/2017) <Jessy Diamond Exum>

loopback test works with new CAN bus ids. (12/07/2017) <Jessy Diamond Exum>

The safety policies implemented were refactored and made modular, to allow to manage them more easily and add new ones without acting on the singular components. The simple checks that verified if a control sent over the TX bus and RX bus was enabled were substituted with the actual safety policies for the different cars, at the beginning only available for Honda cars.

Refactor of safety to support more modular additions of safety policies. (12/07/2017) <Jessy Diamond Exum>

Change all output safety mode identifier to prevent user mistakes. (12/07/2017) <Jessy Diamond Exum>

Modularize safety modes to encourage 3rd party safety code contribution. (17/07/2017) <Jessy Diamond Exum>

separate out controls allowed and safety mode (17/07/2017) <Firmware Batman>

refactor safety and enable tests (18/07/2017) <Firmware Batman>

An important fix was made to correctly send the request to and from the USB bus. Thanks to the distinction between the ingoing and outgoing request destination, it was made easier to direct the data flow.

```
- REQUEST_TYPE = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE
+ REQUEST_IN = usb1.ENDPOINT_IN | usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE
+ REQUEST_OUT = usb1.ENDPOINT_OUT | usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE
```

Panda library now correctly sends USB direction bit. (18/07/2017) <Firmware Batman>
oops, I mean that (18/07/2017) <Firmware Batman>

A further addition was the introduction of a method that allowed to change the bit rate at which data were sent over the CAN bus.

support can speed, and test it (18/07/2017) <Firmware Batman>

The introduction of this way to change the bitrate brought some problems that made the tests fail and the fix required different tries and commits.

hmm, reliability test fails... (18/07/2017) <Firmware Batman>
clean up can interrupts, still failing (18/07/2017) <Firmware Batman>
serial echo test is failing (18/07/2017) <Firmware Batman>
add debug mode, more locks, fix test (18/07/2017) <Firmware Batman>

Also, thanks to the new tests defined and automated through a shell script, more fixes were possible. In particular, it was easier to figure out which buses were used to transmit specific information, for instance, GMLAN was moved over the CAN3 bus, instead of the CAN2 bus where was originally assigned.

cleanly put GMLAN on bus 3, love tests (19/07/2017) <Firmware Batman>

The firmware installation was also optimized, by including during the build process a flasher, in charge of downloading and installing the newest firmware.

write soft flasher (25/07/2017) <Firmware Batman>
make new flasher the default for make (30/07/2017) <Firmware Batman>

To have a clearer separation of the tests, which could allow also more precise testing of the different functionalities, the tests were refactored, and also new ones were added. The new tests added aimed to verify the reliability of the build process and the flashing of the firmware over Wi-Fi.

break tests into two files (30/07/2017) <Firmware Batman>
add wifi tests (30/07/2017) <Firmware Batman>
fix legacy build issue, add build test, fix warnings (01/08/2017) <Firmware Batman>

Also, thanks to the tests were detected some problems when using UDP transfer protocol to download the firmware, problems that were then addressed and solved by introducing a delay.

st wifi flash is failing (30/07/2017) <Firmware Batman>
begin to address UDP reliability issues (13/08/2017) <Firmware Batman>
adding delay made it much better (13/08/2017) <Firmware Batman>

An important addition was the inclusion of the support to ELM327, a command protocol for interfacing with cars using an OBD-II port to read standard vehicle diagnostic codes. ELM is mainly used to diagnose vehicles and reset fault codes in the car's computer after fixing an issue.

ELM327: Car simulator now simulates LIN ISO 14230-4 (KWP FAST) (19/08/2017) <Jessy Diamond Exum>

ELM327: incoming messages cleared before commands to prevent congestion. (19/08/2017) <Jessy Diamond Exum>
some computers are slow to enumerate (21/08/2017) <Firmware Batman>

The introduction of the support of the DSU mode in the Python library allowed to boot the device directly in a recovery mode to perform the erase and recover of the device's firmware, and also automatically trigger the boot of the recovery mode in case of a reset failure.

support DFU mode in python library (22/08/2017) <Firmware Batman>
factor out DFU code (23/08/2017) <George Hotz>
list can fail (13/09/2017) <George Hotz>
ugh, pass (13/09/2017) <George Hotz>
add to dfu as well (13/09/2017) <George Hotz>
try a DFU recover if reset failed (19/09/2017) <George Hotz>
try DFU every time (01/10/2017) <George Hotz>

The Panda Python library folder was then renamed from *panda/panda* to *panda/python*, to avoid misapprehensions.

rename panda to python because of git ambiguity (07/12/2017) <George Hotz>

It was then added the support to more Panda functionalities and hardware, including the Panda Serial and Panda Pigeon, Panda Debug functionalities, Grey Panda hardware check, and K-Line debugging features.

add PandaSerial and location panda (aka pigeon) test (07/01/2018) <George Hotz>
fix panda serial write (19/01/2018) <George Hotz>
add pandadebug support (24/01/2018) <George Hotz>
add is_grey (16/02/2018) <George Hotz>
add kline debug support (01/03/2018) <Jennifer Strange>
kline checksum algo was broken... (01/03/2018) <Jennifer Strange>

Another addition was the support of the IsoTP transfer protocol and a way to send and receive messages through it.

despite it being bad code, move isotp (10/03/2018) <George Hotz>
add way to call isotp (10/03/2018) <George Hotz>
forgot the selfs (10/03/2018) <George Hotz>
long isotp msgs (22/06/2018) <George Hotz>

More safety hooks were also added to be compliant with the guidelines and regulations that are mandatory for level 2 self-driving cars. Up to this moment in time, the safety hooks available regarded cars manufactured by Chrysler, GM, Honda, and Toyota.

added bosch safety hooks and forwarding (06/03/2018) <gregjhogan>
Chrysler safety controls (#130) (06/11/2018) <Drew Hintz>

A community member, **Chris Vickery**, implemented the WebUSB and WinUSB 2.0 specifications, and also added the support to USB 2.1 specifications. [44]

Implement WebUSB and upgrade WinUSB to 2.0 (#107) (11/04/2018) <Chris Vickery>

Another community member, Nigel Armstrong, made a series of improvements to optimize the power consumption of the device and also added the specification for a hardware-in-the-loop Jenkins test.

Capture make failure so it can be logged to sentry (21/02/2019) <Nigel Armstrong>
Power Saving (#169) (14/03/2019) <Nigel Armstrong>
Additional Power saving (#170) (02/04/2019) <Nigel Armstrong>
Jenkins (#179) (09/04/2019) <Nigel Armstrong>

The next big change was made with the release of the new version of the Panda, the Black Panda. The support to the new hardware was added to the Python library and the new specifications were added to the board source code. This was also the occasion to split the code-related different boards into different files, to better manage and maintain them.

Black (#254) (24/07/2019) <rbiasini>
Black panda Jenkins (#256) (28/08/2019) <robbederks>

After updating the Python version used by the library, many changes were needed to adapt to the new standard and to replace functions and methods that were no longer available in the new version.

env python -> env python3 (25/09/2019) <Riccardo>
xrange is gone (25/09/2019) <Riccardo>
Fix all the prints with 2to3, some need to be undo (25/09/2019) <Riccardo>
undo unnecessary brackets for print (25/09/2019) <Riccardo>
2to3 applied (25/09/2019) <Riccardo>
read file as byte and no tab before sleep (25/09/2019) <Riccardo>
Fixed some python3 bugs in the test scripts and PandaSerial (28/09/2019) <Robbe Derks>
python2 -> 3 fixes to pedal flasher (#292) (09/10/2019) <rbiasini>
More Python 3 fixes, attempting to fix Jenkins wifi regression test (#295) (10/10/2019) <rbiasini>

More additions to the safety specification were made to also include that for Volkswagen, Audi, SEAT, and Skoda. Additional improvements were also made by aligning the enum in cereal with those in the safety specifications of the Panda library.

match safety enum in cereal (#285) (03/10/2019) <rbiasini>
Panda safety code for Volkswagen, Audi, SEAT, and Škoda (#293) (09/10/2019) <Jason Young>

To simplify the process of flashing Electronic Control Units (ECUs) over CAN using a Panda and perform general scanning of ECUs for interesting data it was developed and included in Panda the Unified Diagnostic Services (UDS) library. UDS is a diagnostic communication protocol used in ECUs within automotive electronics, which is specified in the ISO 14229-1. [45]

The addition of the UDS Panda library was entirely conducted by **Greg Hogan**, who collaborated with the Comma.ai team to optimize it and fix the issues that arise during the different tests.

uds lib (15/10/2019) <Greg Hogan>
more UDS message type implementation (15/10/2019) <Greg Hogan>
SERVICE_TYPE enum (15/10/2019) <Greg Hogan>

uds can communication (15/10/2019) <Greg Hogan>
zero pad messages before sending (15/10/2019) <Greg Hogan>
fix remaining size calculation (15/10/2019) <Greg Hogan>
clear rx buffer and numeric error ids (15/10/2019) <Greg Hogan>
multi-frame tx (15/10/2019) <Greg Hogan>
bug fixes (15/10/2019) <Greg Hogan>
flow control delay (15/10/2019) <Greg Hogan>
fix flow control delay scale (15/10/2019) <Greg Hogan>
fix separation time parsing (15/10/2019) <Greg Hogan>
handle separation time in microseconds (15/10/2019) <Greg Hogan>
convert uds lib to class (15/10/2019) <Greg Hogan>
fix rx message filtering bug (15/10/2019) <Greg Hogan>
bug fixes (15/10/2019) <Greg Hogan>
add timeout param (15/10/2019) <Greg Hogan>
support tx flow control for chunked messages (15/10/2019) <Greg Hogan>
updates for python3 (15/10/2019) <Greg Hogan>
more python3 (15/10/2019) <Greg Hogan>
custom errors from thread (15/10/2019) <Greg Hogan>
bytes() > chr().encode() (15/10/2019) <Greg Hogan>
fix WARNING_INDICATOR_REQUESTED name (15/10/2019) <Greg Hogan>
fix more encoding and some bytes cleanup (#300) (15/10/2019) <rbiasini>
better CAN comm abstraction (15/10/2019) <Greg Hogan>
more uds debug (15/10/2019) <Greg Hogan>
timeout is float (16/10/2019) <Greg Hogan>
proper python3 exception inheritance (16/10/2019) <Greg Hogan>
uds drain before send and use has_obd() (06/11/2019) <Greg Hogan>
improve uds message processing (06/11/2019) <Greg Hogan>
uds: no need for threads if you always drain rx (13/11/2019) <Greg Hogan>
uds: better debug prints (13/11/2019) <Greg Hogan>
single addr was better (14/11/2019) <Greg Hogan>
uds zero second timeout (15/11/2019) <Greg Hogan>
uds: handle function addrs and fw version query example (17/11/2019) <Greg Hogan>
functional addr handling (17/11/2019) <Greg Hogan>
UDS: handle remote addressing (17/12/2019) <Willem Melching>
uds: rx message buffering (06/02/2020) <Greg Hogan>
uds: clear rx buffer on drain (06/02/2020) <Greg Hogan>

Another refactor of the power saving was necessary because of two main problems:

- The CAN-based ignition would not trigger the interrupt on the ignition line that changed the power save mode and set the USB power mode to CDP.
- easy to trigger race conditions where EON changed the safety mode and re-enabled some of the CAN lines after interrupt corresponding to the ignition off was triggered.

The solution proposed by **Riccardo Biasini** [46] included several fixes:

- Move power save mode logic to boardd (Openpilot). ON when the car is ON, OFF when the car is ON.

- Only set_power_save_state() call in Panda is when EON is disconnected: enter both SILENT safety mode and power saving mode to ensure that if EON is disconnected while the car is ON, Panda won't remain in high consumption mode.
- Enter CDP mode when check_started() is True.
- It was added a USB command to change the power save state.
- It was added a power save state to the health packet.
- The ignition line interrupt was disabled

Power saving refactor (#389) (21/11/2019) <rbiasini>

Other improvements were made to the functionalities allowing to control the health status of the device. In particular, it was added a global timer instead of the local one that was included before, and the structure of the health packet was also modified, including new parameters indicating the faults and errors that occurred while running the process.

Add uptime counter to the health packet (#391) (22/11/2019) <Robbe Derks>
Fixed health struct size. We should really get an automated test for this (23/11/2019) <Robbe Derks>
Added faults integer to health packet (27/11/2019) <Robbe Derks>
send can_rx_errs in health (20/12/2019) <Riccardo>

As more cars were being supported by Openpilot, also new safety codes and specifications for the new car manufacturers had to be added and the ones for the existing manufacturers were also corrected and improved to support a wider range of cars.

better differentiation of honda safety modes (21/12/2019) <Riccardo>
Volkswagen safety updates: Phase 1 (#444) (19/02/2020) <Jason Young>
Added Nissan safety (#244) (26/02/2020) <Andre Volmensky>
separating subaru legacy safety mode from global (#452) (28/02/2020) <rbiasini>
remove toyota ipas safety code and tests (#460) (04/03/2020) <rbiasini>
Safety model for Volkswagen PQ35/PQ46/NMS (#474) (01/04/2020) <Jason Young>
remove cadillac (#496) (13/04/2020) <Adeeb Shihadeh>

An important improvement made to increase productivity was made thanks to the introduction of GitHub Actions. Thanks to the CI capabilities offered, it was no longer needed the integration with CircleCI, which was replaced. GitHub Actions also allowed the enforcement of static code constraints for both C and Python source code.

GitHub Actions (#535) (18/05/2020) <Adeeb Shihadeh>
only push to dockerhub from master (18/05/2020) <Adeeb Shihadeh>
pull base image (18/05/2020) <Adeeb Shihadeh>
fix docker file path (18/05/2020) <Adeeb Shihadeh>
Fast CI (#539) (19/05/2020) <Adeeb Shihadeh>
Add pre commit checks + CI (#545) (29/05/2020) <Willem Melching>
Speed up misra test in CI (#552) (08/06/2020) <Adeeb Shihadeh>
only push to docker registry from master (08/06/2020) <Adeeb Shihadeh>
update dockerhub token (12/06/2020) <Adeeb Shihadeh>
Build socketcan driver in CI (#588) (09/08/2020) <Adeeb Shihadeh>

The last big update regard modifications aimed to support the new Red Panda, which compared to its predecessors is based on a new board model (STM32H7).

Support for STM32H7 and Red Panda (#694) (03/08/2021) <Igor Biletskyy>

3.6 Rednose

Rednose is a Kalman filter library that can be used for a wide range of optimization problems. In particular, it is used for problems in the field of visual odometry and sensor fusion localization (SLAM). It is designed to provide very accurate results, work online or offline, and be computationally efficient.

The library applies the **Rauch-Tung-Striebel (RTS) Smoother algorithm**, which is composed of two passes: the forward pass consists of a standard Extended Kalman Filter (EKF), while the backward pass is introduced to reduce the inherent bias in the EKF estimates.

In estimation theory, EKF is the nonlinear version of the Kalman filter which linearizes about an estimate of the current mean and covariance. The EKF can be considered as the de facto standard in the theory of nonlinear state estimation, navigation systems, and GPS.

3.6.1 Package structure

The library offers a series of helper functions and classes that allow performing the filtering actions, as well as the templates that help the function to generate the result of its computation. [Figure 40]

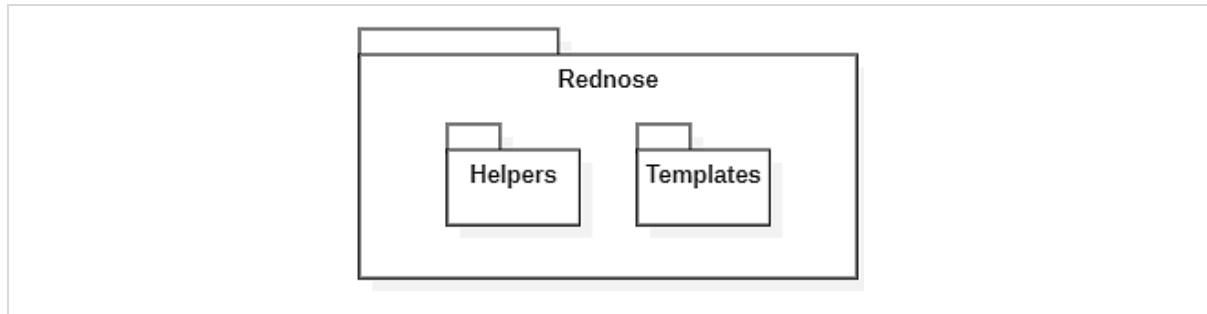


Figure 40 - Rednose package diagram

It provides both a Python and a C++ implementation for its classes and methods. It also comes with a Cython wrapper for the main EKF component, which allows generating C code and calls it in Python libraries.

Language	Files	Blank	Comment	Code
C++	5	158	93	2.741
Python	7	248	134	911
C/C++ Header	5	34	4	201
C	3	34	21	176
Cython	1	35	6	149
SUM:	21	509	258	4.178

Table 44 - Lines of code of Rednose, by programming language

3.6.2 Implementation

The main component exposing the Extended Kalman Filter functionalities is the EKFSym class.

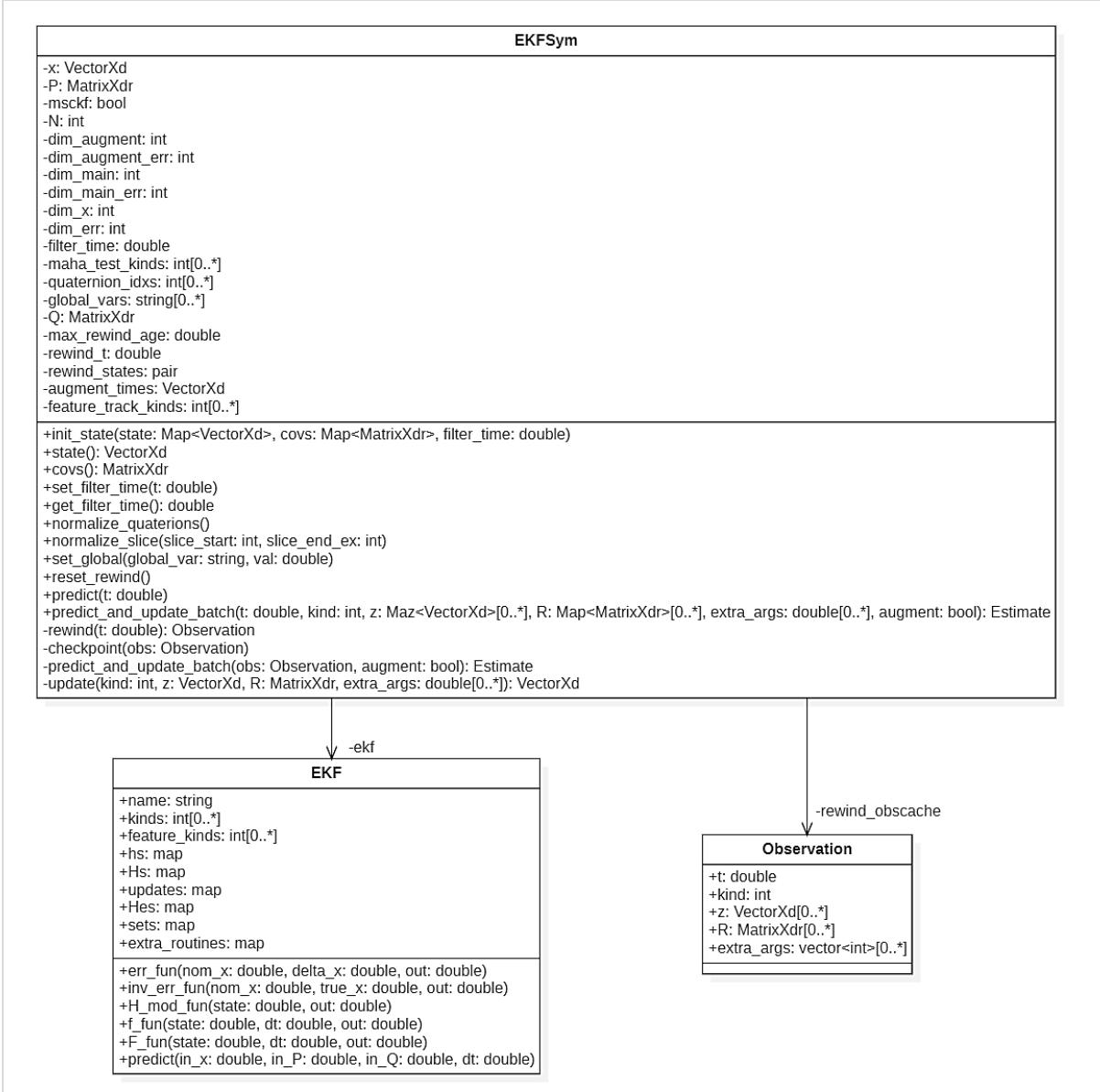


Figure 41 - EKFSym class diagram. EKFSym leverages the Extended Kalman Filter EKF that is generated according to the selected measurement model and provides the prediction and update methods.

When an EKFSys object is instantiated it first initializes the process functions and all the observation functions of the Kalman filter.

Compared to a simple Kalman Filter, an EKF uses a method called **First Order Taylor Expansion** to linearize the distribution of a nonlinear function, like can be a radar or GPS measurement. If a real-valued function $f(x)$ is differentiable at the point $x = a$, then it has a linear approximation near this point, and this relation can be described by using (3.13).

$$f(x) \approx f(a) + f'(a)(x - a) \quad (3.13)$$

A nonlinear system can be generally defined by equations (3.14) and (3.15), where x identifies the state vector and z the measurements vector and w_k and v_k are the process and measurement noises.

$$x_k = f(x_{k-1}, u_k) + w_k \quad (3.14)$$

$$z_k = h(x_k) + v_k \quad (3.15)$$

The initial optimal state and error covariance are initialized with the mean (3.16) and covariance (3.17) of x , as those are the only data available when the algorithm starts.

$$x_0^a = E[x_0] \quad (3.16)$$

$$P_0 = Cov[x_0] \quad (3.17)$$

```
void EKFSym::init_state(Map<VectorXd> state,
                        Map<MatrixXd> covs,
                        double filter_time) {
    this->x = state;
    this->P = covs;
    this->filter_time = filter_time;
    this->augment_times = VectorXd::Zero(this->N);
    this->reset_rewind();
}
```

By expanding the nonlinear function describing the state using the first-order Taylor expansion we obtain (3.18)

$$f(x_{k-1}) = f(x_{k-1}^a) + F \cdot e_{k-1} \quad (3.18)$$

where F is the Jacobian of the main state and $e_{k-1} \equiv x_{k-1} - x_{k-1}^a$. The Jacobian is defined as:

$$F = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \dots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \quad (3.19)$$

By conditioning $f(x_{k-1})$ by the vector of the measurements z we can identify the expected value with the (3.20), where $E[e_{k-1}|Z_{k-1}] = 0$, therefore the forecasted value will be expressed by the (3.21):

$$E[f(x_{k-1})|Z_{k-1}] \approx f(x_{k-1}^a) + F \cdot E[e_{k-1}|Z_{k-1}] \quad (3.20)$$

$$x_k^f \approx f(x_{k-1}^a) \quad (3.21)$$

By substituting (3.18) in the forecast error equation we obtain:

$$\begin{aligned} e_k^f \equiv x_k - x_k^f &\approx F \cdot e_{k-1} \\ &\quad + w_{k-1} \end{aligned} \quad (3.22)$$

The forecast error covariance will be given by:

$$\begin{aligned}
P_k^f &\equiv E \left[e_k^f (e_k^f)^T \right] \\
&= F \cdot E[e_{k-1} \cdot e_{k-1}^T] \cdot F^T + E[w_{k-1} \cdot w_{k-1}^T] \\
&= F \cdot P_{k-1} \cdot F^T + Q_{k-1}
\end{aligned} \tag{3.23}$$

The predictions of the state and state covariance are calculated by the method `predict()`. The EKF component of a EKFSym can be generated starting from different measurement models, and the prediction algorithm, as well as the update algorithm, will change accordingly to model from which the EKF component is generated.

```
// functions generated from sympy
f_fun(in_x, dt, nx);
F_fun(in_x, dt, in_F);

EEM F(in_F);
EEM P(in_P);
EEM Q(in_Q);

RRM F_main = F.topLeftCorner(MEDIM, MEDIM);

P.topLeftCorner(MEDIM, MEDIM) = (F_main*P.topLeftCorner(MEDIM, MEDIM)) *
    F_main.transpose();
P.topRightCorner(MEDIM, EDIM - MEDIM) = F_main *
    P.topRightCorner(MEDIM, EDIM - MEDIM);
P.bottomLeftCorner(EDIM - MEDIM, MEDIM) =
    P.bottomLeftCorner(EDIM - MEDIM, MEDIM) *
    F_main.transpose();

P = P + dt*Q; // prediction step
```

The update step starts with the computation of the difference between the measured value and the actual value. The measured value is computed by function `h(x)`, which specifies how the speed and the position of x are mapped to polar coordinates.

$$y = z - h(x) \tag{3.24}$$

The total error (or residual covariance) and the (near-optimal) Kalman Gain can then be computed by applying (3.25) and (3.26), respectively.

$$S = H \cdot P \cdot H^T + R \tag{3.25}$$

$$K = P \cdot H^T \cdot S^{-1} \tag{3.26}$$

H is the Jacobian matrix of the observations and R is the measurement noise.

The updated state estimate and covariance estimate can be computed by applying:

$$x_k^a = x_k^f + K \cdot y \tag{3.27}$$

$$P_k = (I - K \cdot H)P_k^f \tag{3.28}$$

```

// update state by injecting dx
Eigen::Matrix<double, EDIM, 1> dx(delta_x);
dx = (KT.transpose() * y);
memcpy(delta_x, dx.data(), EDIM * sizeof(double));
err_fun(in_x, delta_x, x_new);
Eigen::Matrix<double, DIM, 1> x(x_new);

// update cov
P = ((I_KH * P) * I_KH.transpose()) + ((KT.transpose() * R) * KT);

```

In the case in which a Multi-State Constraint Kalman Filter is used, it is needed to run a **Mahalanobis Distance test**. The Mahalanobis Distance is an effective multivariate distance metric that measures the distance between a point and a distribution. This measure is useful to identify multivariate outliers, which could negatively influence the outcome of the computation.

```

if (MAHA_TEST) {
    XXM a = (H_err * P * H_err.transpose() + R).inverse();
    double maha_dist = y.transpose() * a * y;
    if (maha_dist > MAHA_THRESHOLD) {
        R = 1.0e16 * R;
    }
}

```

The method *predict_and_update_batch()* is the one providing the main Kalman filter functionalities. It leverages the C functions included in the library to perform prediction and update operations in a faster and more efficient way.

Like in the first case, the prediction function computes the Jacobian matrix F and it calculates the estimated covariance.

In the update step, after the computation of the Jacobian of the measurements, it is calculated the loss between the measured value and the actual value of the observation. Like in the previous case, the formulas derived from the resolution of the equations are used to compute the updated state estimate and covariance estimate.

The library provides a function that allows generating the source code describing all the parameters needed by the Extended Kalman Filter to perform the prediction and the update step starting from a given measurement model. The input parameters include all the state variables of the models, while the output consists of 2 files, a C header, and the C source code file, containing the definition of the linearization functions using the Jacobian, the error functions, the state propagation functions, and the observation functions.

The backward recursion pass of the Rauch-Tung-Striebel Smoother algorithm is performed by the method *rts_smooth()*. The smoothed estimates generated during the backward are more precise and reliable than the estimation of EKF only.

The Python library allows to use only the EKF or combine it with RTS smoother. Even if the second method usually gives better results, in some cases it may be preferable to use only the EKF to reduce the computation time and still have a good level of precision. [47]

3.6.3 Usage

The main component of Openpilot that makes usage of the Rednose library is the process *locationnd*. This process takes care of the localization of the car by combining data coming from vision sensors and GPS and smoothing the result using the Extended Kalman Filter.

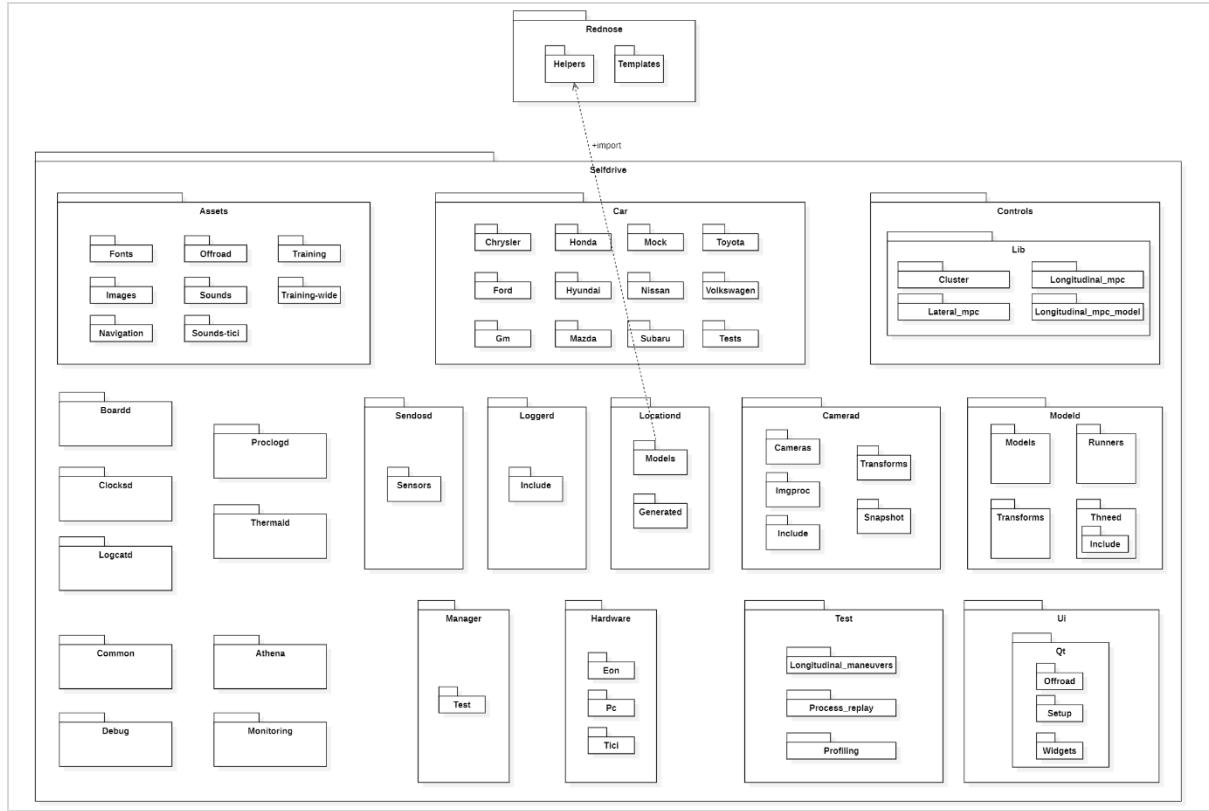


Figure 42 - Import dependencies between selfdrive and rednose

It is possible to choose among various measurement models for the EKF implementation. A particular model is selected based on many factors, one being the limitations of the available measurements. One of the main models is Laika, which provides much more accurate GNSS data acquisitions. All the different measurements model are combined to get a much more accurate estimation of the car positions. LiveKalman is the generated component implementing the Extended Kalman Filter functionalities and leveraging all the measurement models, including GNSS, vision, and sensors.

```
private:
    std::unique_ptr<LiveKalman> kf;
```

The parameters needed to apply the EKF forward step of the RTS smoother algorithm can be retrieved through the LiveKalman instance. These include the initial state and covariance, the Jacobian matrix, and the predicted state and covariance.

```
VectorXd init_x = this->kf->get_initial_x();
MatrixXdr init_P = this->kf->get_initial_P();
double filter_time = this->kf->get_filter_time();
VectorXd predicted_state = this->kf->get_x();
MatrixXdr predicted_cov = this->kf->get_P();
MatrixXdr HH = this->kf->H(H_input);
```

LiveKalman class comes with a *filter* object, which is an instance of the EKFSym class. Through this object, it can access all the Rednose library functionalities, including prediction and smoothing methods offered by the Extended Kalman Filter.

```

this->kf->predict_and_observe(sensor_time,
                                OBSERVATION_PHONE_GYRO,
                                {meas});

this->kf->predict_and_observe(sensor_time,
                                OBSERVATION_PHONE_ACCEL,
                                {meas});

this->kf->predict_and_observe(current_time,
                                OBSERVATION_ECEF_ORIENTATION_FROM_GPS,
                                { initial_pose_ecef_quat });

this->kf->predict_and_observe(current_time,
                                OBSERVATION_ECEF_POS,
                                { ecef_pos },
                                { ecef_pos_R });

this->kf->predict_and_observe(current_time,
                                OBSERVATION_ECEF_VEL,
                                { ecef_vel },
                                { ecef_vel_R });

this->kf->predict_and_observe(current_time,
                                OBSERVATION_NO_ROT,
                                { Vector3d(0.0, 0.0, 0.0) });

this->kf->predict_and_observe(current_time,
                                OBSERVATION_CAMERA_ODO_ROTATION,
                                { (VectorXd(rot_device.rows() +
                                rot_device_std.rows()) << rot_device,
                                rot_device_std).finished() });

this->kf->predict_and_observe(current_time,
                                OBSERVATION_CAMERA_ODO_TRANSLATION,
                                { (VectorXd(trans_device.rows() +
                                trans_device_std.rows()) << trans_device,
                                trans_device_std).finished() });

```

In LiveKalman, the method *predict_and_observe()* makes use of the *predict_and_update_batch()* method of the EKFSym class, the main method of the class which computes the predictions made by the Extended Kalman Filter.

3.6.4 Testing

The two test cases available for the Rednose library and executed through GitHub Actions when a new commit is made aim to test the performance of both the Python and the Cython implementations of the library, as well as the precision of the predictions made through the library.

The test case `TestCompare` instantiate creates two different implementations of the Extended Kalman Filter. Both of them use C code to perform their operations, with the difference that in one case the code is generated using the Cython wrapper, while in the other case the source code is generated by a Python method and invoked thanks to the C Foreign Function Interface (FFI) for Python.

This second methodology produces the same result as Cython, allowing to call C code from Python code, increasing the overall performances, but the advantage is that the implementation of this methodology does not require the knowledge of a third programming language, like in the case of Cython.

CompareFilter		TestCompare	
+ name: string		+ dt: float	
+ initial_x: array		+ ts: arange	
+ initial_P_diag: array		+ xs: empty	
+ Q: ndarray		+ x: float	
+ obs_noise: map<ObservationKind, ndarray>		+ switch: tuple	
+ filter_py: EKF_sym		+ test_compare()	
+ filter_pyx: EKF_sym			
+ generate_code(generated_dir)			
+ get_R(kind, n): ndarray			

Table 45 - `TestCompare` test case

The purpose of the test case is to show that the two implementations have similar performances.

```
self.assertAlmostEqual(kf.filter_py.get_filter_time(),
                      kf.filter_pyx.get_filter_time())

self.assertTrue(np.allclose(kf.filter_py.state(),
                           kf.filter_pyx.state()))

self.assertTrue(np.allclose(kf.filter_py.covs(),
                           kf.filter_pyx.covs()))
```

The second test case uses the Extended Kalman Filter to correct a set of measurements and create estimations that are much closer to the actual data than the measured data.

KinematicKalman	1..	0..	TestKinematic
<ul style="list-style-type: none"> + name: string + initial_x: array + initial_P_diag: array + Q: ndarray + obs_noise: map<ObservationKind, ndarray> + filter_py: EKF_sym + filter_pyx: EKF_sym + generate_code(generated_dir) 			<ul style="list-style-type: none"> + dt: float + ts: arange + vs + x + xs + xs_meas + xs_kf + vs_kf + xs_kf_std + vs_kf_std + test_kinematic_kf()

Table 46 - TestKinematic Test Case

In Figure 43 is shown how the Kinematic EKF can predict values that are much closer to the actual measurements acquired than a normal simulation is able to do.

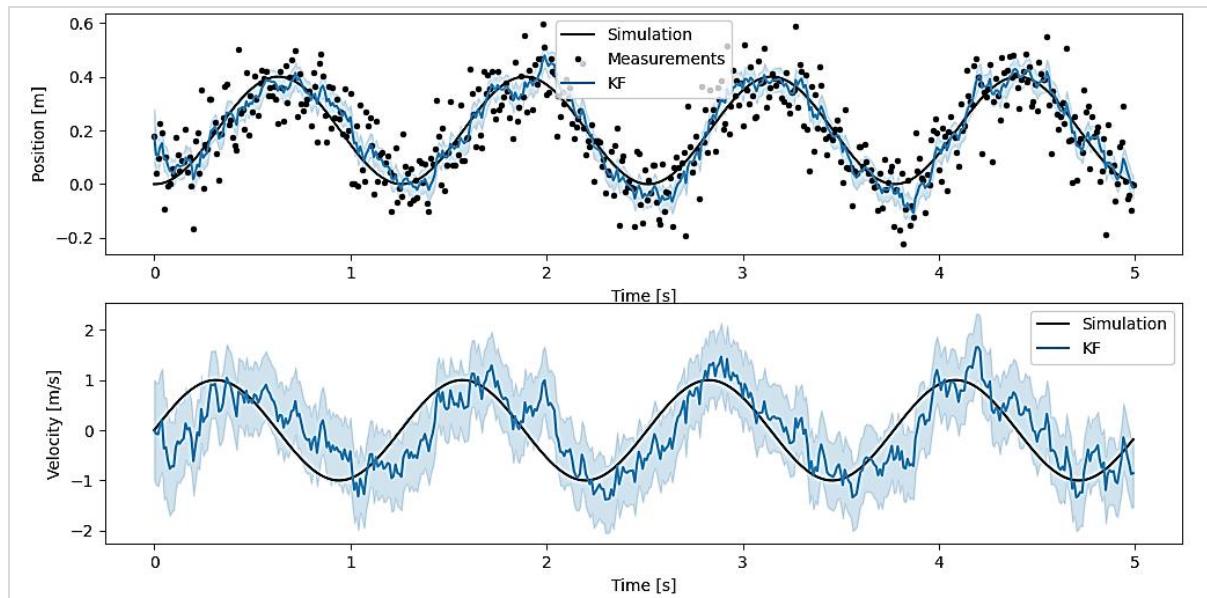


Figure 43 - Kinematic EKF simulation plot

3.6.5 Development and community contributions

Rednose was added to Openpilot only when Openpilot reached its release 0.7.x and replaced some of the functionalities that were originally performed by the simple Kalman filter located in the *common* directory.

initial commit (14/05/2020) <Willem Melching>
--

The Kinematic EKF was one of the first additions. This example of the EKF, which compares the prediction of parameters related to the position and speed of the car with the actual measurements taken, was added to showcase the capabilities of the filter and its usage.

add kinematic example and test (16/05/2020) <Willem Melching>
--

Add picture (16/05/2020) <Willem Melching>

To improve the usability of the Kalman filter, it was also added a Kalman filter base class, which comes with a *filter* property to which can be assigned an EKF object.

add kalman filter base class (16/05/2020) <Willem Melching>
--

remove unused constant (16/05/2020) <Willem Melching>
--

The tests available for the library were integrated with static analysis, performed before each commit, and using different Python linters, including *flake8*, *mypy*, and *pylint*.

run static analysis in CI (29/05/2020) <Willem Melching>

Fix docker container name (29/05/2020) <Willem Melching>

use github url (29/05/2020) <Willem Melching>
--

use same pre-commit config as Openpilot (04/06/2020) <Adeeb Shihadeh>
--

cleanup pre-commit config (22/06/2020) <Adeeb Shihadeh>
--

Along with the EKF_sym class, already available in Python and using SymPy to generate the C method to invoke, it was also added an EKF_sym class in C++ and then wrapper using Cython. They offer similar performances but having an implementation both in C++ and Python for the EKF_sym class offers a higher degree of flexibility, allowing Openpilot to leverage the Extended Kalman filter functionalities in modules written both in Python and C++.

EKF_sym class rewritten to c++ (#9) (08/04/2021) <Joost Wooning>

changes for locationd in c++ (#13) (20/04/2021) <Joost Wooning>
--

4 Self-driving cars: an overview of the Openpilot framework and its quality assessment

Openpilot needs many components that allow the software to interface with the car and exchange messages with it. These components, after the open-sourcing of the software, were organized in different repositories, allowing to better manage them and have a clear distinction of what role each component plays. In this excursus of the submodules that are available in the main Openpilot repository will be analyzed the functionalities that each one of them provides, how they were tested to ensure the required levels of reliability, and what was the development process they went through.

4.1 Package structure

The *selfdrive* directory is the largest repository of Openpilot and contains the definition of the Python and C++ processes executed on the Comma device. The choice of Python and C++ is common for applications executed on microcontrollers that require to interface with external actuators, since many microcontrollers can natively run C++ code. Also, C++ grants a high level of performance, while Python provides high programmer productivity. In most cases, the C++ libraries are wrapped using Cython, which can be defined as a superset of Python that compiles to C and C++. This allows calling C and C++ code from Python code, keeping both the advantages in the usability of Python and the performance of C++.

Language	Files	Blank	Comment	Code
Python	164	3.935	1.630	19.587
C++	103	3.458	1.444	16.736
C	11	958	212	10.275
C/C++ Header	121	2.244	3.056	9.736
OpenCL	7	88	60	556
SVG	15	0	0	242
JSON	2	0	0	104
HTML	2	11	0	89
Bourne Shell	8	17	3	83
QML	1	5	0	42
Cython	1	4	2	22
SUM:	435	10.720	6.407	57.472

Table 47 - Lines of code of *selfdrive*, by programming language

To describe the architecture of Openpilot we can take advantage of the 4+1 architectural view model of Kruchten, which provides a systematic way to describe a system according to different views. As for the **logical view** of the architecture, the only task that Openpilot performs is to drive the car. From version 0.8.7 is also possible to use a navigator.

Looking at the **development view**, Openpilot is made of different packages which allow communication with the car and run the predictive model.

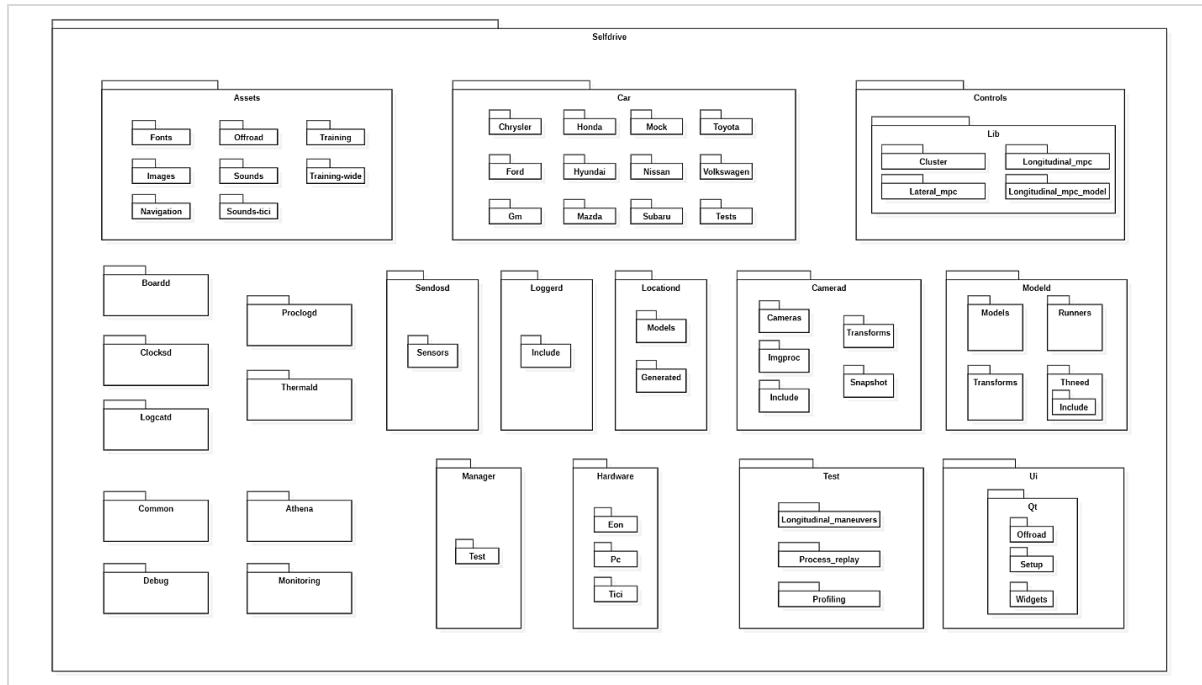


Figure 44 - Openpilot package diagram

Each package contains the different processes of Openpilot. The processes communicate by using the Cereal messaging specifications, but they also provide common functionalities that can be used by other processes to retrieve various details, such as the hardware version, the car model on which the device is mounted, and so on.

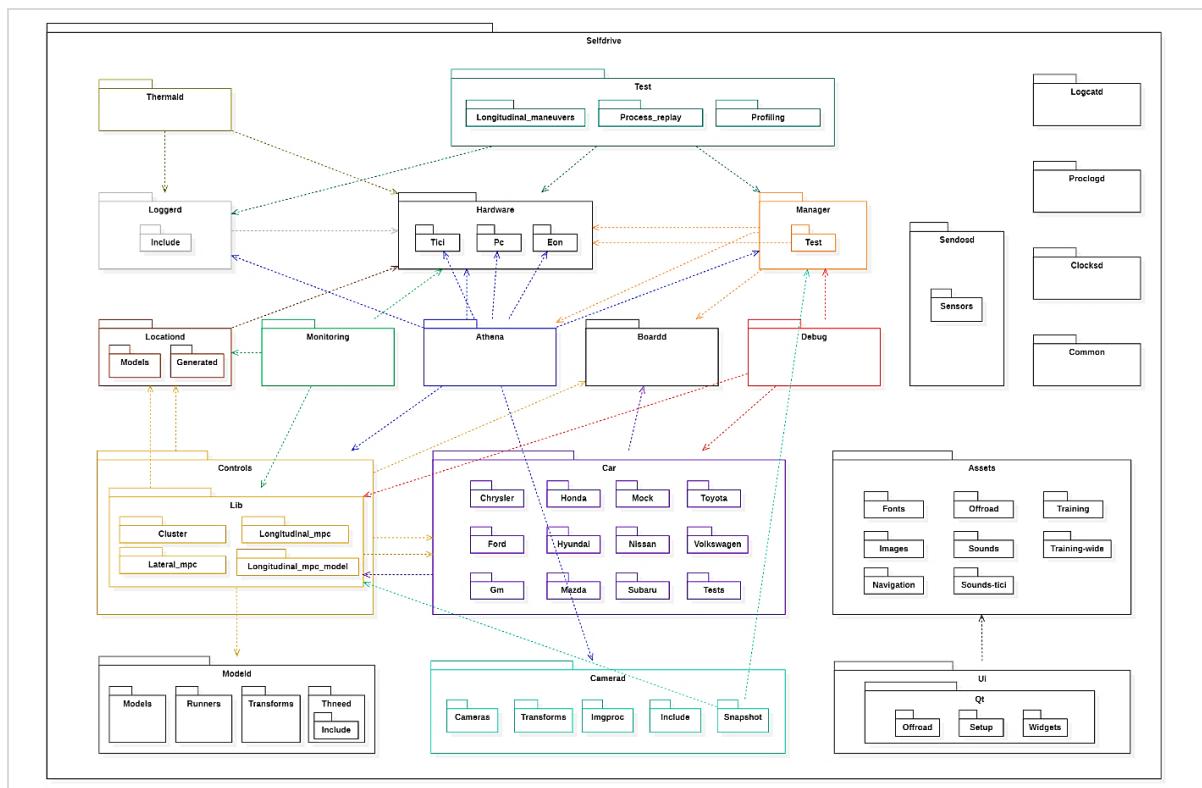


Figure 45 - Dependencies of the selfdrive packages

The **physical** components that are involved are essentially three: the Comma device (Comma Two or Comma Three), a Comma Pedal to communicate with the car, and the supported car itself.

The car provides the raw data that is given to the AI model to generate an output that is then transformed into actions by the car actuators. Raw data come from the car camera, radar, and sensors.

Data are then transmitted over the CAN bus, and the Panda device interfaces with the car and intercepts the data over the CAN bus. These data are then transmitted, over USB, to the Comma device.

The Openpilot software, running on the device, comes with different process daemons that manage the acquired data, give them to the AI model, and generate a message that is sent to the car over the same CAN bus. In Figure 46 is represented Openpilot's component diagram, showing the core nodes and components involved in the system.

The camera daemon, (*camerad*), sends the frames acquired by the car and device's camera to the model daemon (*modeld*), which generates the predictions based on AI algorithms, and the sensor daemon (*sensord*) elaborates the output of the car's sensors. All of these data are input into the control daemon (*controlsd*), which generates the messages that are sent to the car through Panda and over the CAN bus.

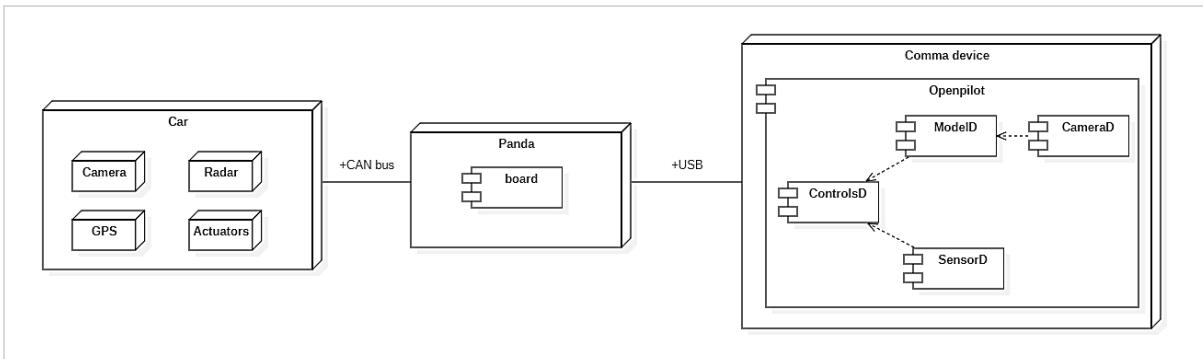


Figure 46 - Openpilot deployment diagram

Looking at the **process view**, selfdrive is characterized by the parallel execution of different processes, which take care of the different aspects that need to be controlled during the software execution.

Openpilot processes include both Python and C++ processes. Generally, the C++ processes are those which require a high level of performance and that can be executed natively on the Comma device. These include the processes taking care of the prediction and elaboration of the acquired frames of the camera, the processes directly controlling or showing content on the device, and other logging functionalities.

More specifically, the taxonomy in Figure 47 shows the different categories of processes and their type.

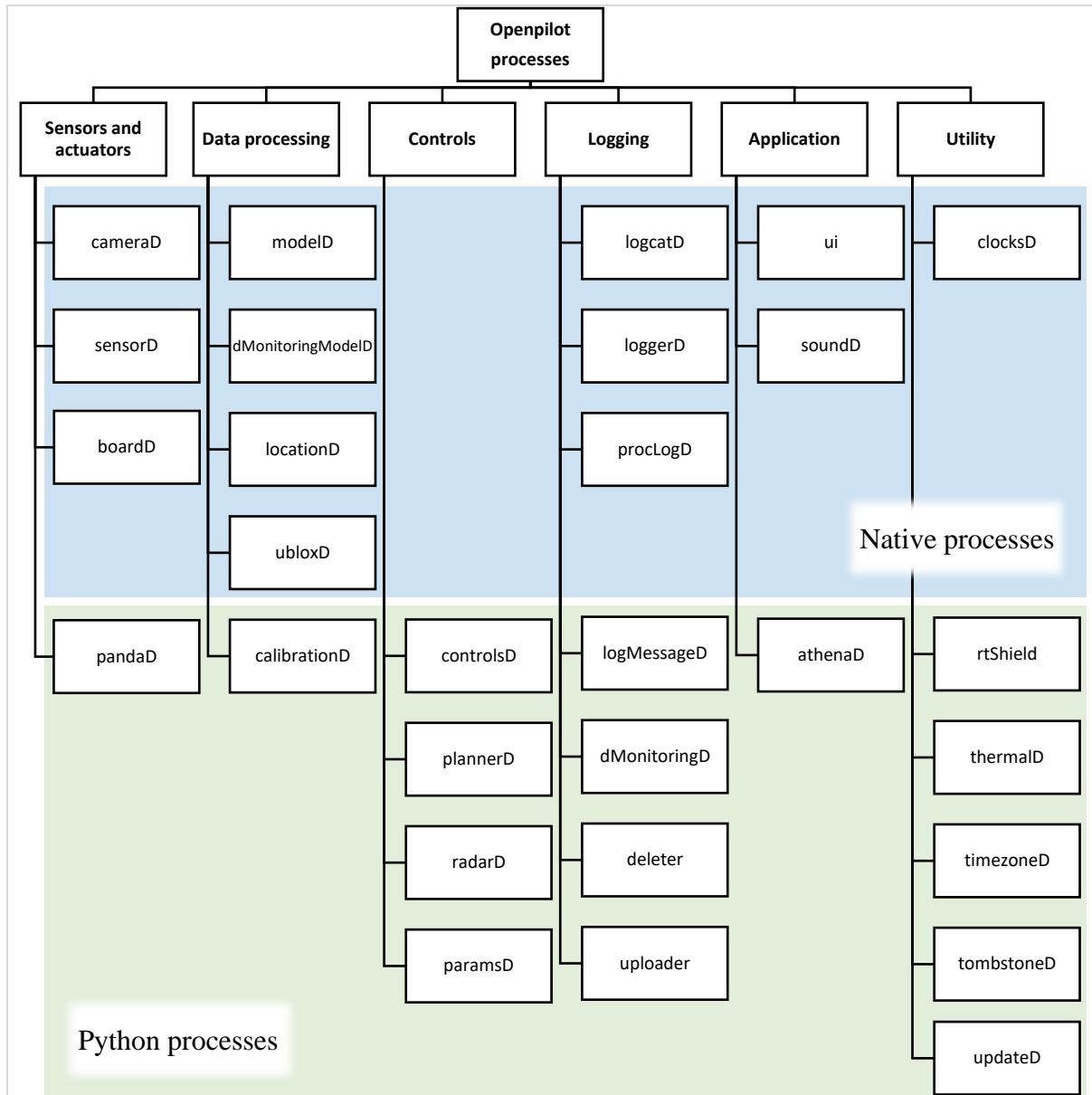


Figure 47 - processes taxonomy, organized by type and objective of the process

After launching the software and building the different modules, Openpilot starts its process manager, which runs for the whole execution of the software and ensures that each process is running, also sending and receiving state messages to and from the different components.

As we can see in Figure 48, the processes started are many and each one of them performs a specific action. Each of them will be further analyzed in a dedicated paragraph.

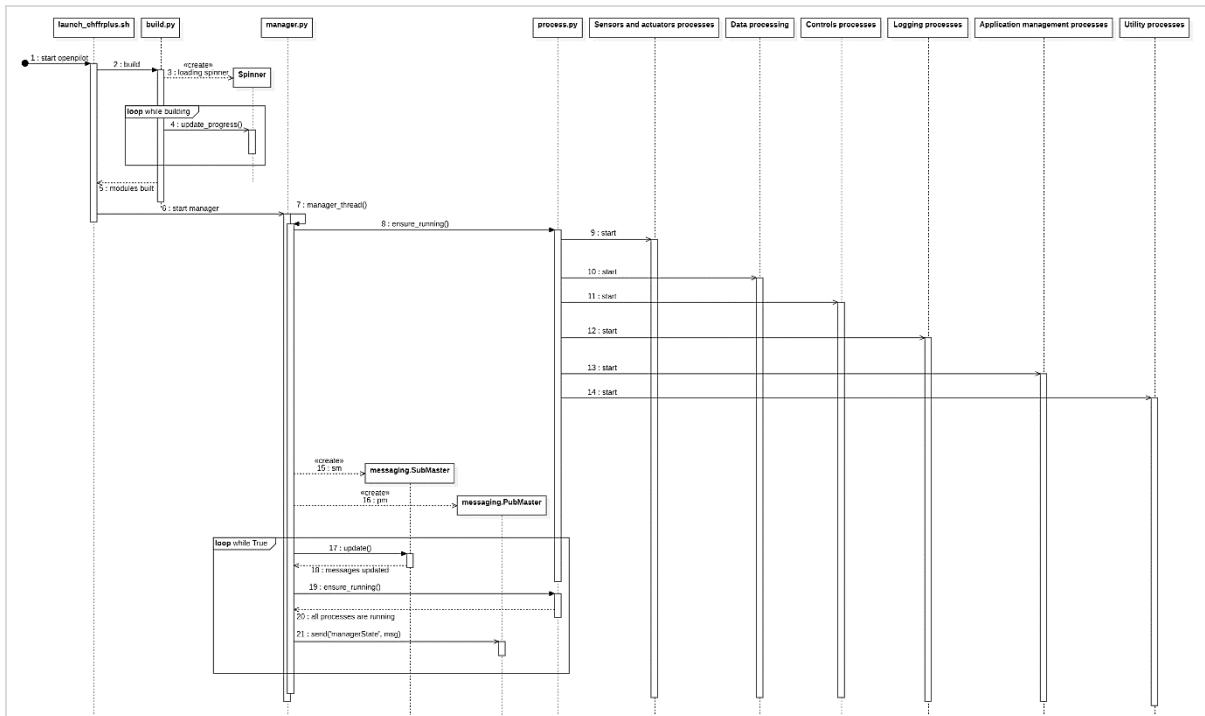


Figure 48 - Sequence diagram of the initialization of Openpilot. ManagerD daemon takes care of managing the choreography of processes that run during the execution of the software.

4.2 Implementation

To analyze the implementation of the different processes, each of them will be considered individually, focusing on how they communicate with each other and what functionalities provide and how.

4.2.1 AthenaD

This service allows real-time communication with the car. It runs also if the car is parked and not moving and allows to access different functionalities of the car from a dedicated application.

The Athena daemon leverages the Comma API to instantiate a WebSocket object that listens to inbound JSON-RPC requests. The specific WebSocket can be reached by specifying the unique *dongle_id* parameter.

```

dongle_id = params.get("DongleId", encoding='utf-8')

ATHENA_HOST = os.getenv('ATHENA_HOST', 'wss://athena.comma.ai')
ws_uri = ATHENA_HOST + "/ws/v2/" + dongle_id

api = Api(dongle_id)

ws = create_connection(ws_uri,
                      cookie = "jwt=" + api.get_token(),
                      enable_multithread = True,
                      timeout = 30.0)

```

An application can request an active WebSocket by sending a JSON-RPC message to the base URL <https://athena.comma.ai> and specifying the required service.

The JSON-RPC response will contain the result of the query. Below is shown an example of a response containing the result of a request for the health status of a car.

```
{  
    "jsonrpc": "2.0",  
    "result": {  
        "health": {  
            "gasInterceptorDetected": true,  
            "controlsAllowed": true,  
            "started": true,  
            "current": 77,  
            "startedSignalDetected": false,  
            "isGreyPanda": true,  
            "voltage": 12008  
        },  
        "logMonoTime": 12553712590731  
    },  
    "id": "db7cb7a3-5f54-4521-91ac-51877064da11"  
}
```

Internally, Athena manages the requests and responses through different threads. The functions that can be called remotely are added to the server by using a JSON-RPC Dispatcher. The available functions are:

- **getMessage()**: request for any message from the car.
- **getVersion()**: return the version of the system installed on the device.
- **setNavDestination()**: set a destination in Openpilot's navigator, by specifying the latitude and longitude of the point to reach.
- **listDataDirectory()**: list the directory and files in the device.
- **reboot()**: reboot the comma device from remote.
- **uploadFileToUrl()**: put a file into the upload queue and return the object. The queue of object to upload is managed by a specific thread.
- **listUploadQueue()**: list the files currently in the upload queue.
- **cancelUpload()**: empty the upload queue and put the elements in another queue, containing all the canceled uploads.
- **primeActivated()**: returns a boolean value indicating if Comma Prime is activated on the device.
- **getPublicKey()**: get the public RSA key, needed in combination with the private key to authenticate through SSH.
- **getSshAuthorizedKeys()**: get the list of public authorized keys.
- **getSimInfo()**: get the status of the SIM card in the Comma device, if one is present. Information includes the connection state and the MAC address of the SIM card.
- **getNetworkType()**: return the type of network to which the device is connected (Wi-Fi, mobile...).
- **getNetworks()**: returns the supported network connections.
- **takeSnapshot()**: acquire a picture from the road camera and the front-facing camera.

The function `getMessage()` uses Cereal to instantiate a subscriber to a socket corresponding to the specified service. Thanks to the Cereal library, the subscriber retrieves a message from the publisher that is sending messages on that same socket, and returns the result to the client making the JSON-RPC request.

```
socket = messaging.sub_sock(service, timeout=timeout)
ret = messaging.recv_one(socket)
```

4.2.2 BoardD/PandaD

This process represents the receiving side of the Panda firmware. It parses and sends data through USB by using the library `libusb`.

The thread managing the send of data uses a SubSocket object and leverages the Cereal library specifications to retrieve the data on the socket `sendcan`.

```
SubSocket * subscriber = SubSocket::create(context, "sendcan");
Message * msg = subscriber->receive();

capnp::FlatArrayMessageReader cmsg(aligned_buf.align(msg));
cereal::Event::Reader event = cmsg.getRoot<cereal::Event>();

panda->can_send(event.getSendcan());
```

When a message is received it is then parsed and adapted to the default word size, by splitting the message in multiple messages long at maximum as the maximum word size length.

The parsed message is then sent to Panda, which will receive it and forward it to the CAN bus of the car.

The thread managing the receipt of messages, instead, instantiates a PubMaster object, and when receiving a message from Panda it publishes it the socket `can`.

```
void can_recv(PubMaster &pm) {
    kj::Array<capnp::word> can_data;
    panda->can_receive(can_data);
    auto bytes = can_data.asBytes();
    pm.send("can", bytes.begin(), bytes.size());
}
```

The board component can talk both to a newer version of Panda hardware (like Panda Black and Panda Red) and an older version, like Panda Grey (also known as Pigeon to clearly distinguish it from the other Panda devices, since Pigeon is deprecated in the latest versions of Openpilot).

We already saw in the implementation details of the Panda library how the board initializes its interrupts and the transceiver to send and receive the messages on the UART: in this case, the receiver side is represented by this implementation of the Panda (and Pigeon) [Figure 49] and allows to send and receive messages to the board, that through the UART can communicate with the Comma device and the car.

Like in the case of the Python library of Panda, provided to test and communicate with the Panda devices easily and simply, this implementation provides the same methods and functionalities to query all the details of the board.

- get_RTC()
- get_fan_speed()
- get_state()
- get_firmware_version()
- get_serial()

The same methods are also available for the Python Panda library and can provide the same details about the board.

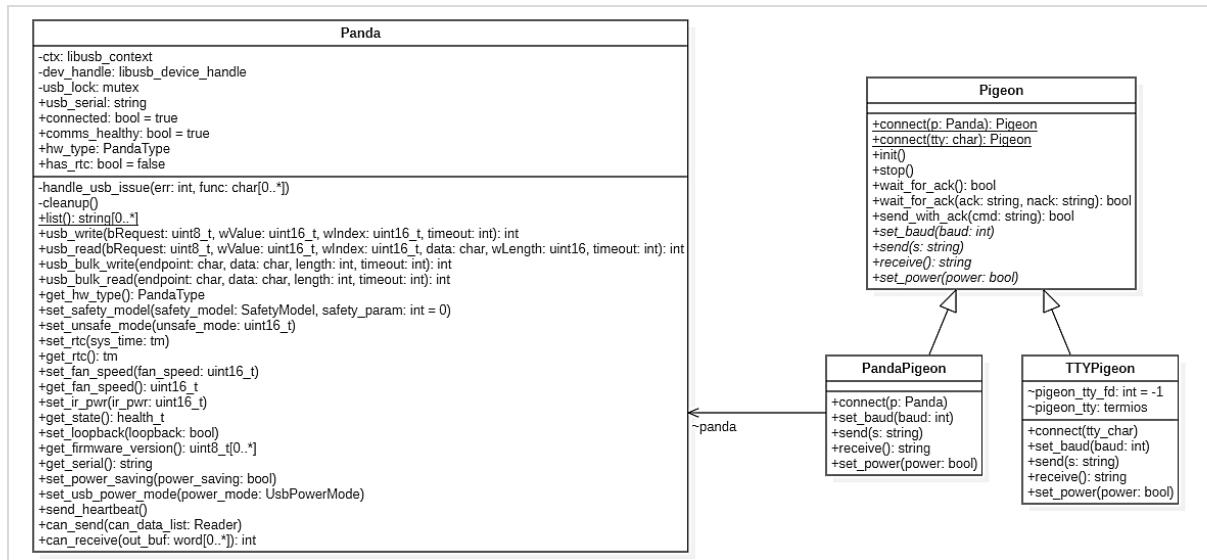


Figure 49 - Panda and Pigeon class diagram

The board daemon is started by the process **pandaD**, which is a Python wrapper of boardD that updates the Panda first. It uses the Python Panda library to configure the device and after that, it launches the main boardD process.

```

def main() -> None:                                     # pandaD main
    panda = update_panda()

    health = panda.health()                            # check health for lost heartbeat
    if health["heartbeat_lost"]:
        cloudlog.event("heartbeat lost", deviceState=health)

    cloudlog.info("Resetting panda")
    panda.reset()

    os.chdir(os.path.join(BASEDIR, "selfdrive/boarddd"))
    os.execvp("./boarddd", ['./boarddd']) # launch boardD

```

4.2.3 CameraD

The camera daemon captures both the road and driver camera and handles autofocus and autoexposure.

The camera daemon uses VisionIPC, in combination with the Cereal library, to send the frames data to the other component. In particular, the VisionIPC server sends the data frames directly to the model daemon, which uses the frames to compute the predictions.

```
vipc_server->send(cur_rgb_buf, &extra);  
vipc_server->send(cur_yuv_buf, &extra);
```

The frames are acquired by using the functionalities provided by the Qualcomm Snapdragon chipset. For Snapdragon chips no public SDK is available, so the Comma 2 had to reverse engineer the chipset to be able to access the provided camera functionalities.

Snapdragon 820 (used in Comma Two) and Snapdragon 845 (used in Comma Three) use different SDKs to access the cameras, both of which have been reverse-engineered and available in *qcom* and *qcom2*, respectively.

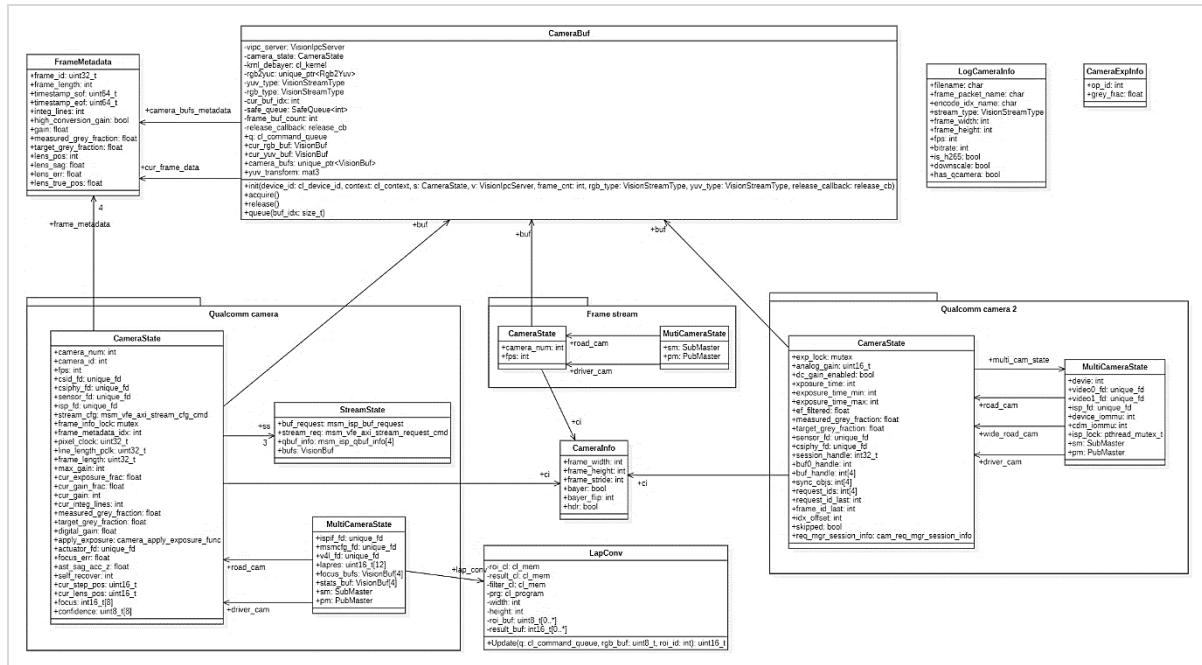


Figure 50 - camerad class diagram. The two cameras only differ for the way they handle the processor instructions calls.

In both cases, the acquisition of the frames is managed by different threads, one for each camera. The cameras are first initialized: in this phase are specified the parameters specifying the cameras' settings, including for instance the ISO and the frame rate.

If in the *qcom* camera specification we find the initialization of two cameras, the front-facing camera, and the road camera, in *qcom2* three cameras have to be initialized, since the Comma Three, equipped with the Snapdragon 845, has two road cameras and one front-facing camera.

For each camera are acquired an RGB and a YUV frame since both are used by different components. The model receives YUV encoded images, while the UI displays on the screen RGB encoded images.

After specifying the image quality settings and the hardware parameters are initialized, a SubMaster and a PubMaster are instantiated. The SubMaster subscribes to the socket *driverState*, while the PubMaster will publish data on the sockets *roadCameraState*, *driverCameraState*, and *thumbnail*. The PubMaster for the *qcom2* implementation also publishes data on the socket *wideRoadCameraState*, specific for the wide-angle camera of the Comma Three.

The cameras are then opened, and the acquisition of the frames begins. The VisionIPC server starts to listen on socket *camerad* to handle all the incoming requests and send the acquired frames as response.

A thread for each camera will encapsulate the acquired frame data, containing information on the exposure, time of the acquisition of the frame, and focus, in a message that is then sent through the PubMaster, previously instantiated, on the *roadCameraState* socket (in the case of the thread managing the road camera) or *driverCameraState* (for the thread managing the driver camera).

The process also continuously polls the video events and creates a queue of all the data events.

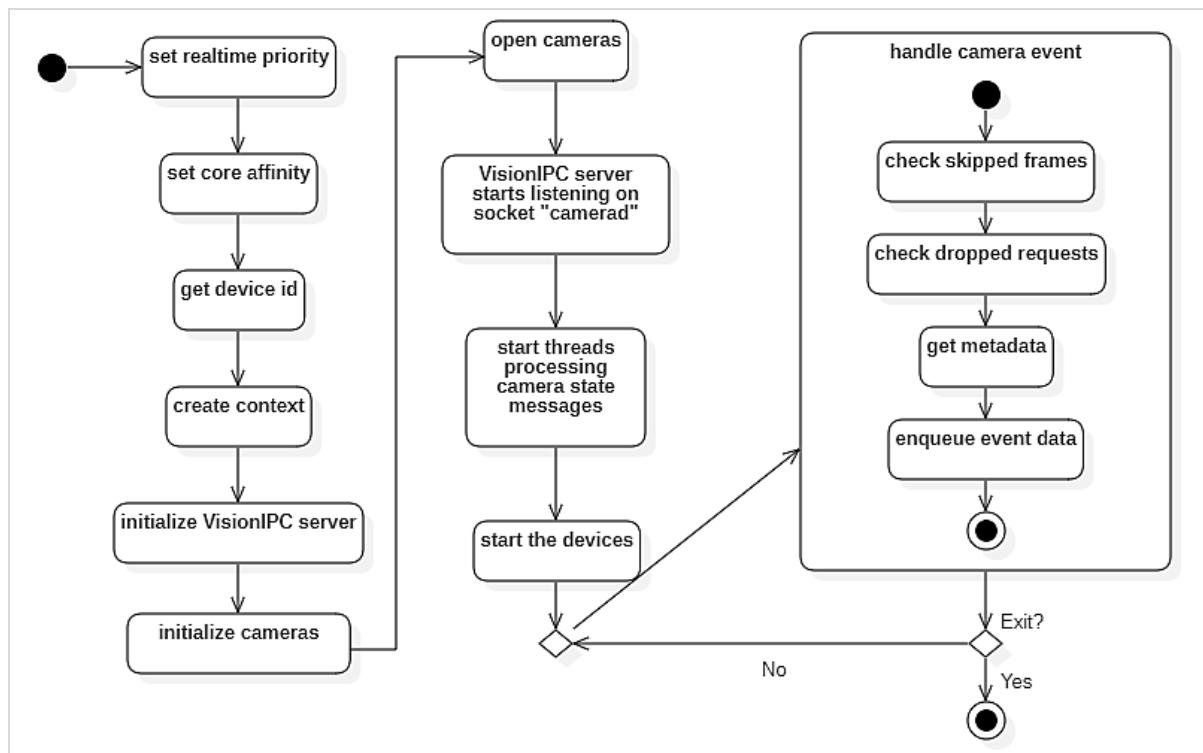


Figure 51 - camerad activity diagram

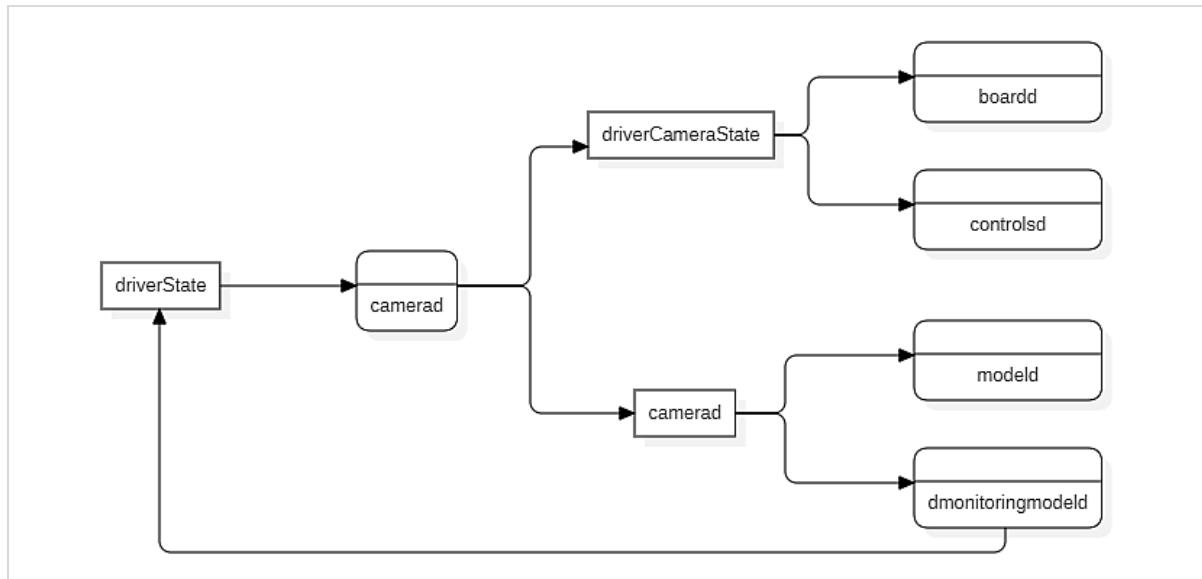


Figure 52 - camerad data flow diagram

4.2.4 ControlsD

This process represents the main 100 Hz loop driving the car. It receives a plan from the planner daemon and constructs the CAN packets required to actuate that plan.

The process uses a SubMaster to receive data on the states of the different car components and a PubMaster to send the computed CAN packets, containing the instructions for the actuators, to Panda.

Data arriving on the socket *can* are sampled at a frequency of 100 Hz. On the *can* socket can be obtained the *carState* of the car. The *carState* represent the main car abstraction used by Openpilot and includes all the details of the car, including information on the status of the different car parts and component, speed, acceleration, etc...

Besides, the *controlsd* process receives data from many other processes of Openpilot [Figure 53]: all these data are needed to compute the corrections to apply to the actuators of the car to actuate the planned predictions and to decide if is needed to disengage Openpilot due to some problems related to the system itself or the actions of the driver.

On the socket *deviceState* are transmitted information on the device that *controlsd* will use to decide if the battery level or the memory is enough to enable Openpilot.

Similarly, on the socket *pandaState* the board daemon transmits information on the Panda device, which can be used to determine if there are hardware problems with the Panda and eventually disengage Openpilot.

Model daemon, the process applying the AI prediction model to the acquired camera frames, sends on socket *modelV2* the predictions, which can include a lane change, a hard brake, and also information on the road like the number of lanes detected.

From the calibration process is retrieved the status of the calibration, while from the driver monitoring process gets information on the state of awareness of the driver: if the calibration is not completed or the driver is not aware it ignores the prediction of the model process.

The planner produces the lateral and longitudinal plans, which contain indications of the adjustments to be made to the steering and acceleration of the car.

The components that manage these changes are Longitudinal Control and Latitudinal Control. Longitudinal Control deals with the acceleration and braking of the car, while Lateral Control with the steering.

From the socket *liveLocationKalman* the process can get information on the status of the GPS and notify the event of a missing or unstable GPS signal.

Using the same logic, on the socket *radarState* the radar process communicates if there is any fault in the radar component, while on sockets *roadCameraState*, *wideRoadCameraState*, and *driverCameraState* are notified of the faults of the corresponding camera.

Through the socket *managerState*, the process keeps track of all the processes, ensuring that all of them are running.

To make precise corrections is also needed to what is the degree of error of the component, specific for each car. These data are provided on the socket *liveParameters* and include the stiffness factor, the steering ratio, the average offset in degrees that there is when steering.

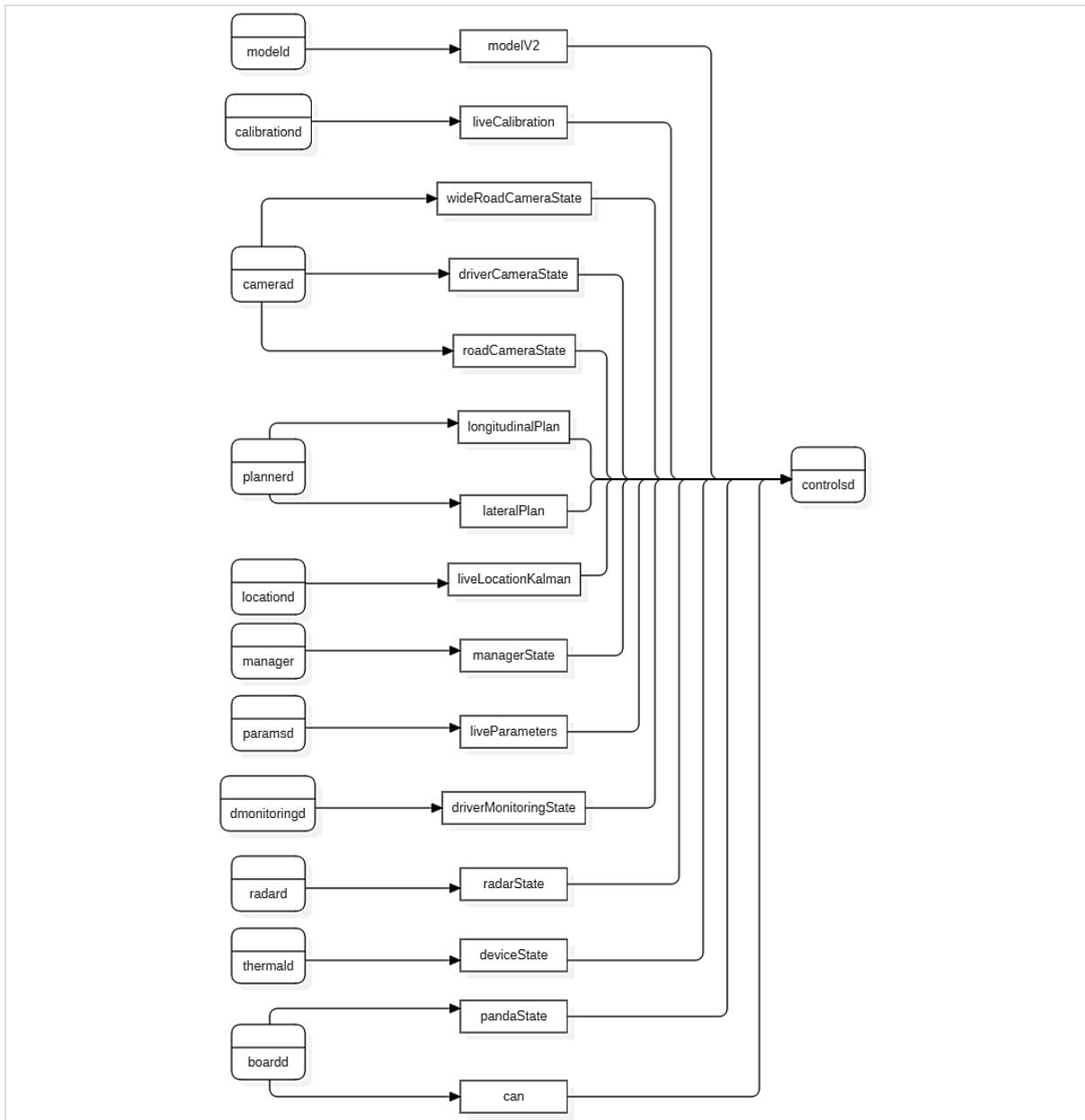


Figure 53 - controlsd inbound data flow diagram

In its main loop, simplified in *Figure 54*, all the data acquired at each iteration allow generating events that can trigger Openpilot to disengage if a problem is detected or can generate and forward messages indicating to the car if there is the need to steer and where, brake, or accelerate.

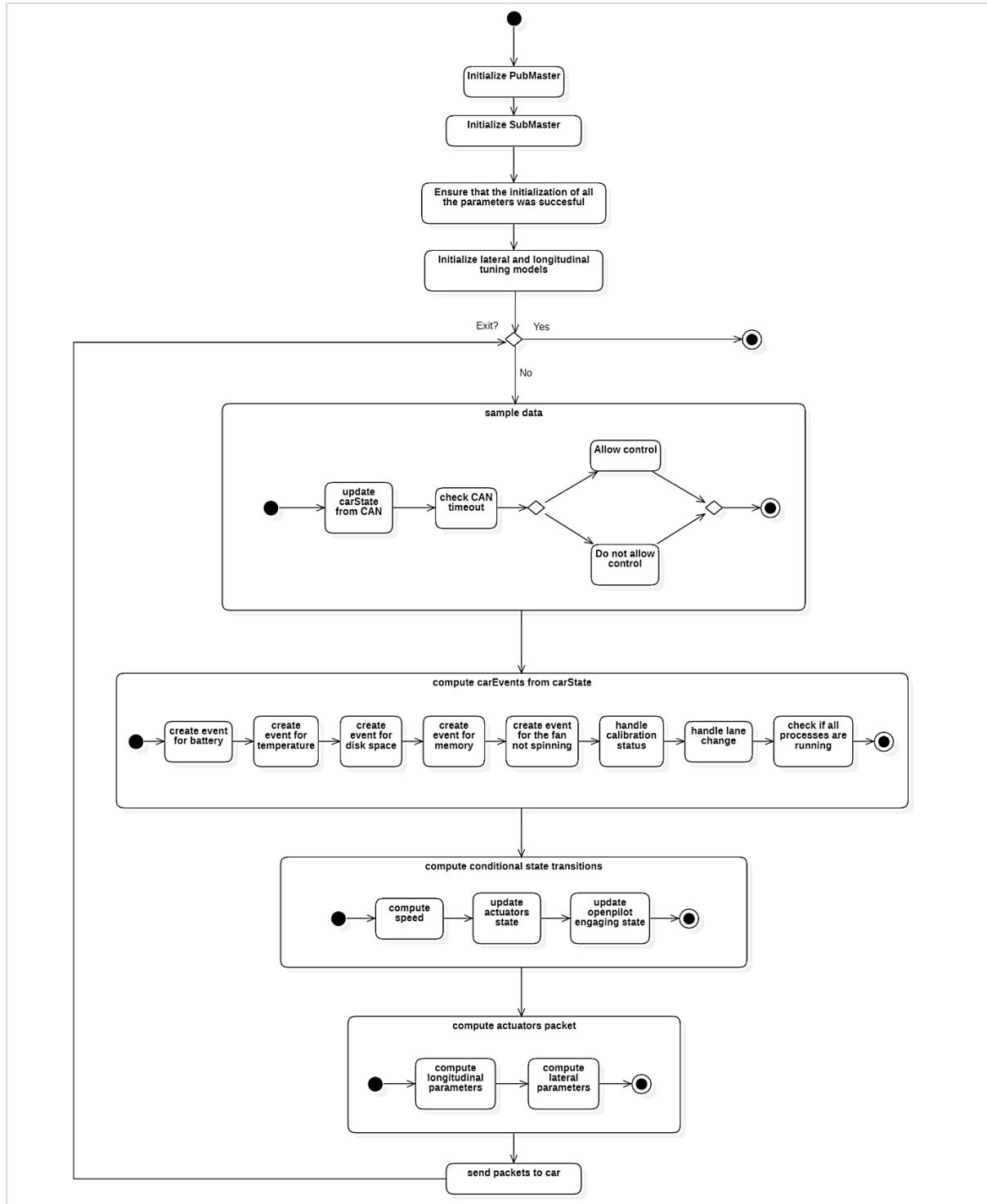


Figure 54 - `controlsd` activity diagram

The components `CarInterface`, `CarState`, and `CarControls` [Figure 55] represent the abstractions of the different aspects of a car that the process can use to communicate with it. They are located in `selfdrive/car`, and each car has its specific implementation of the components.

`CarInterface` provides the methods to get the basic information of the car and to interact with it. The `CarController` allows to generate and send a CAN message by leveraging the OpenDBC library, which provides the CAN message specifications for each car manufacturer. `CarState` instead provides information about the state of the car components.

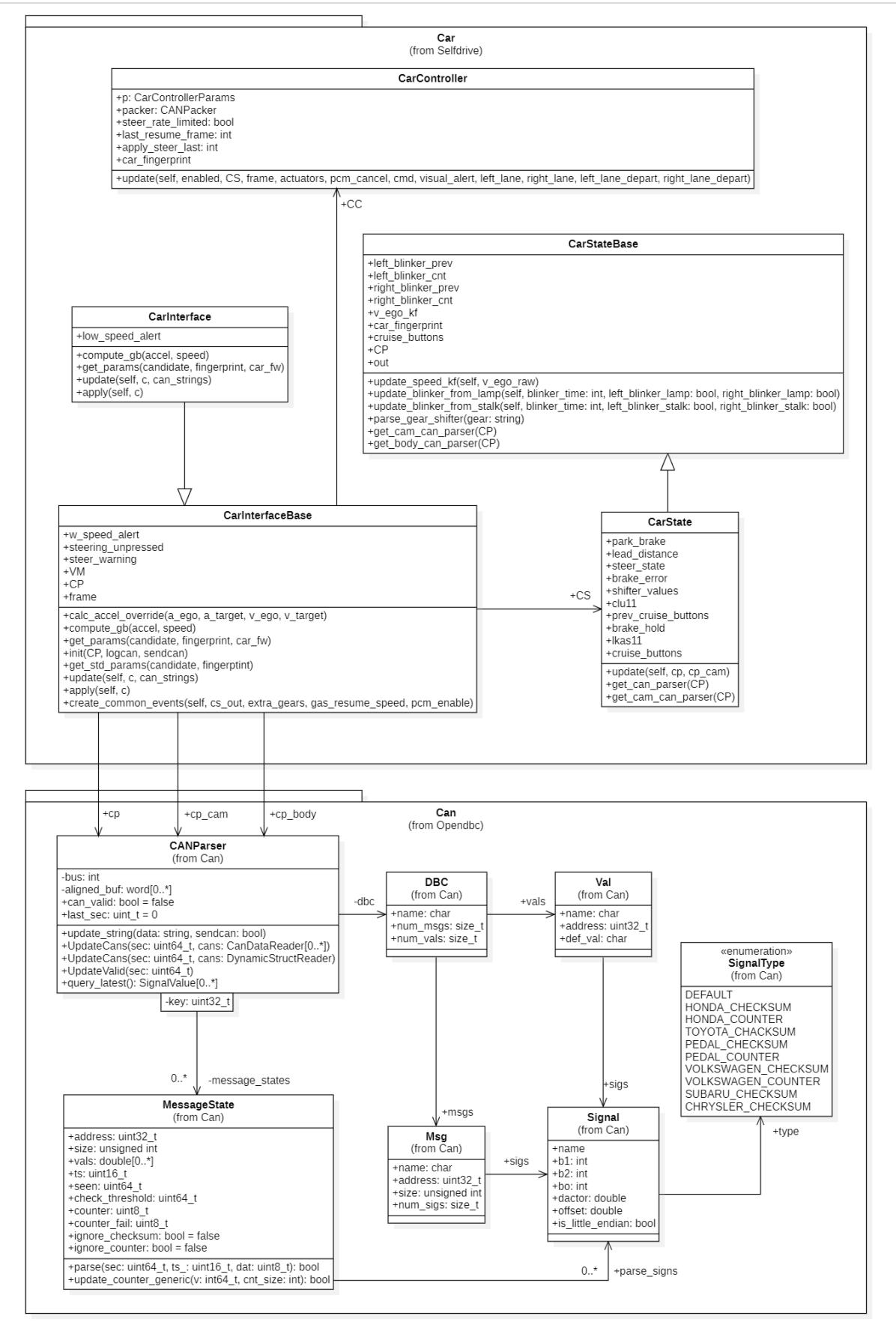


Figure 55 - CarInterface class diagram. The CarInterface leverages the CANParser component to elaborate the received messages.

Through the CarInterface *apply()* method, which then uses the *update()* method provided by the CarController object, the process builds the CAN packets, which are then sent on the socket *sendcan*. The daemon process managing the panda board subscribes to that same socket, receiving the messages to forward to the car.

The updated controls are then sent also on the socket *controlsState* and received by the processes *dmonitoring* and *plannerd*. In particular, these processes will stop their normal operations if the control state tells that Openpilot should disengage.

Similarly, the new computed CarState and CarParams are sent over the sockets *carState* and *carParam*, respectively, to transmit information about the changed conditions of the different actuators of the car.

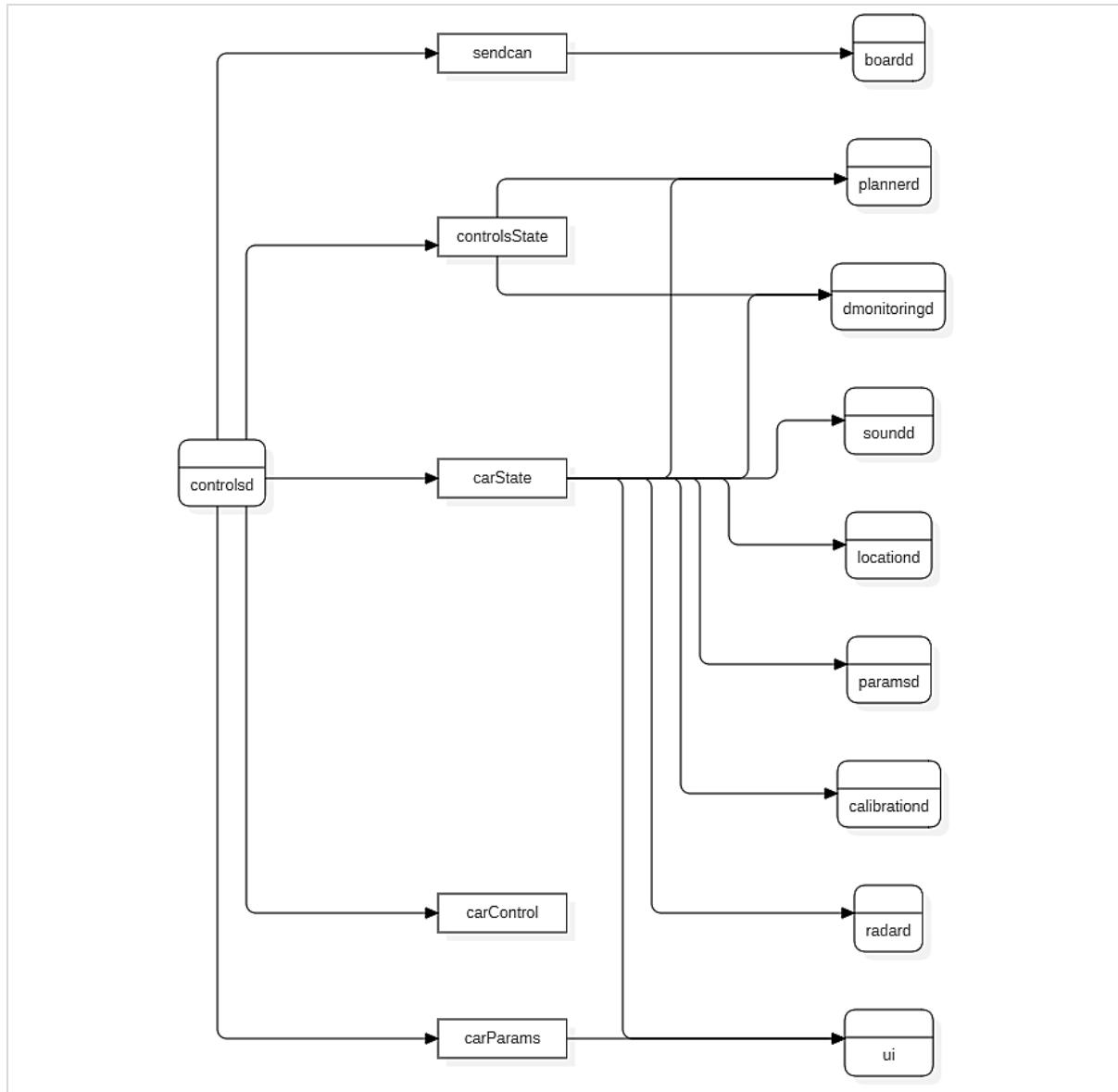


Figure 56 - *controlsd* outbound data flow diagram

4.2.5 PlannerD

After processing the camera images through the AI model, Openpilot has to compute a way to bring the car in a position that is coherent with that indicated by the output model. The planner process executes three Model Predictive Control (MPC) loops based on Automatic Control and Dynamic Optimization (ACADO), one for lateral control and two for longitudinal control.

If both the model and the radar are updated the planner will run the lateral and longitudinal planners.

```
if sm.updated['modelV2']:
    lateral_planner.update(sm, CP)
    lateral_planner.publish(sm, pm)
if sm.updated['radarState']:
    longitudinal_planner.update(sm, CP)
    longitudinal_planner.publish(sm, pm)
```

The lateral planner will first get the predictions by parsing the model retrieved on socket *modelV2* through the component LanePlanner. The *parse_model()* method will initialize the values of the class starting from the ModelDataV2 message specifications defined in the Cereal library. This message includes details on the predicted future position and orientation of the car and predictions on the position lane lines and road edges.

After that the model data is obtained, the process applies the lane change logic to determine the next state in which the car needs to be. The different states and the possible flows are represented in Figure 57.

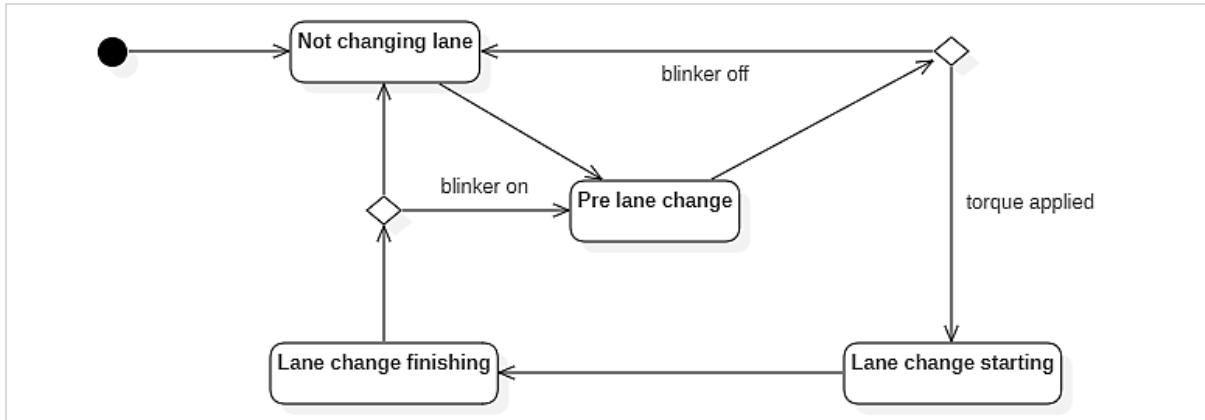


Figure 57 - lane change statechart diagram

When changing the lane, the process does not calculate the path to stay in the lanes, since they have to be crossed. If the model does not indicate the intention to cross a lane, the path is calculated through the LanePlanner.

The method *get_d_path()* calculates the direction on the y axis that the car has to travel to respect the predictions of the model's output.

The path is calculated by computing the weighted average of the probabilities that the car has to go left or right of a certain amount.

```
self.d_prob = l_prob + r_prob - l_prob * r_prob
lane_path_y = (l_prob * path_from_left_lane + r_prob *
               path_from_right_lane) / (l_prob + r_prob + 0.0001)
lane_path_y_interp = np.interp(path_t, self.ll_t[safe_idxs],
                               lane_path_y[safe_idxs])
path_xyz[:,1] = self.d_prob * lane_path_y_interp +
                (1.0 - self.d_prob) * path_xyz[:,1]
```

The computed path points indicating the deviation that the car has to take are sent over socket *lateralPlan* and received by both the control and model daemons.

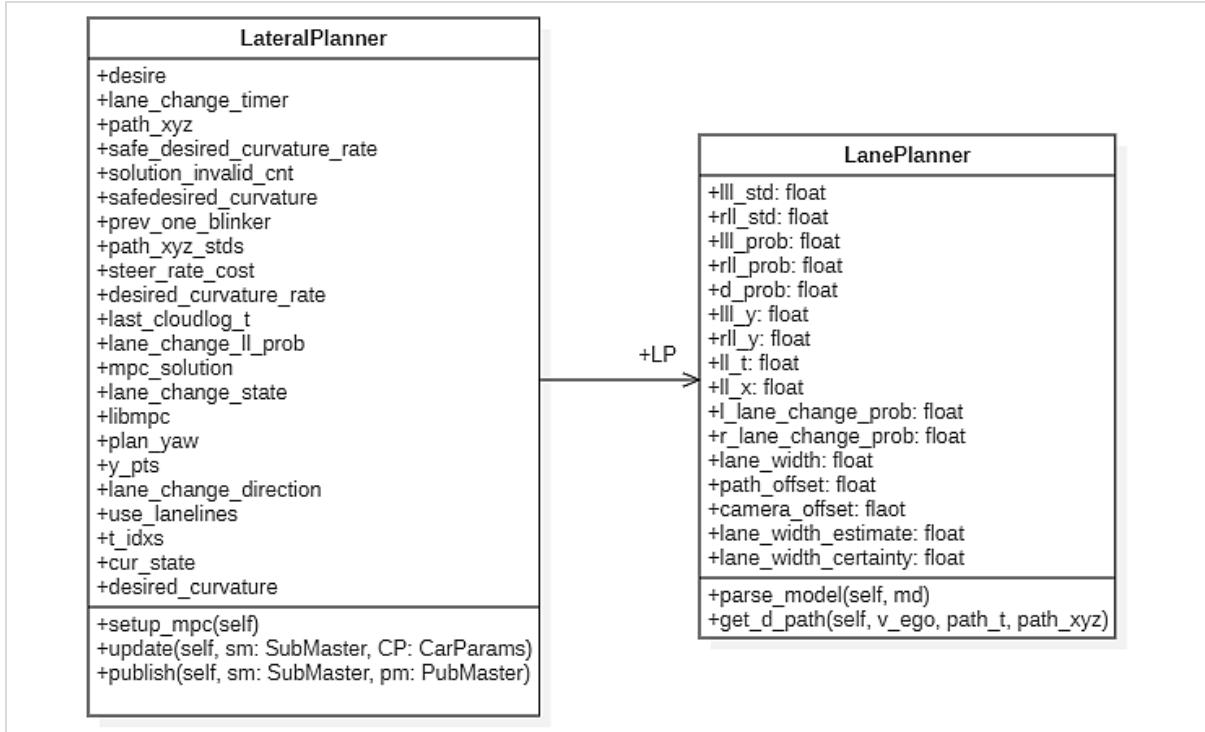


Figure 58 - LateralPlanner class diagram

For what concerns Longitudinal Planning, the Planner class provides the methods to get the current values of velocity and acceleration and computes the changes that the actuators have to apply to reach the target values predicted by the model.

The Planner uses a Model Predictive Controls approach to estimate the speed and acceleration of the car. By solving the control problem and updating the current state each sample time, the Planner is able to achieve a real-time implementation of the MPC technique. A real-time implementation allows working with a triple-integrator model based on the speed (v), acceleration (a), and jerk (j) of the vehicle seen as a particle in longitudinal displacement (d).

This system can be expressed using (4.1) [48].

$$\begin{bmatrix} \dot{d}_l \\ \dot{v}_l \\ \dot{a}_l \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} d_l \\ v_l \\ a_l \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} j_l \quad (4.1)$$

The system first initializes the current state of the car and the target values for speed and acceleration, then runs the two types of MPC algorithm available, which include the Lead Car MPC and Longitudinal MPC.

```
self.mpc[key].set_cur_state(self.v_desired, self.a_desired)
self.mpc[key].update(sm['carState'], sm['radarState'], v_cruise)
```

In the case of the LeadMPC, the estimations will be based on the values of speed, acceleration, and distance of the first car that the vehicle follows. If there is actually a car in front of the vehicle, then its current state is estimated by retrieving the radar acquisitions, if there is no lead car, instead, the parameters are initialized with a fictitious and fast lead car to keep the model running. The MPC based on lead cars predicts the future position, speed, and acceleration of the leading car.

The second class of MPC that the planner uses is a LongitudinalMPC where the only parameters used to make the estimations come from the current state of the vehicle.

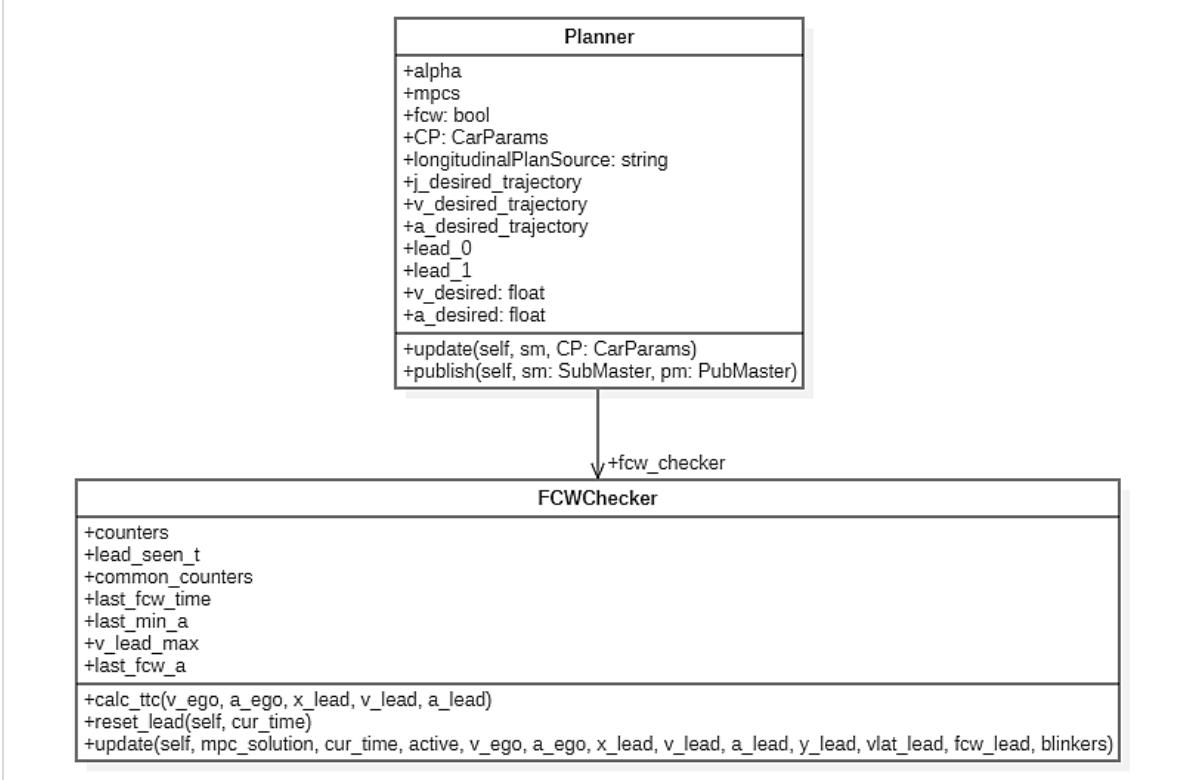


Figure 59 - LongitudinalPlanner class diagram

For solving the previous constrained optimization problem, the MPC solver ACADO toolkit is used. It is a self-contained library based on C++ designed to solve linear and non-linear models under multi-objective optimization functions.

Both the models are executed, and the applied solution will be that returning the lower acceleration. The computed solution is sent over the socket *longitudinalPlan*.

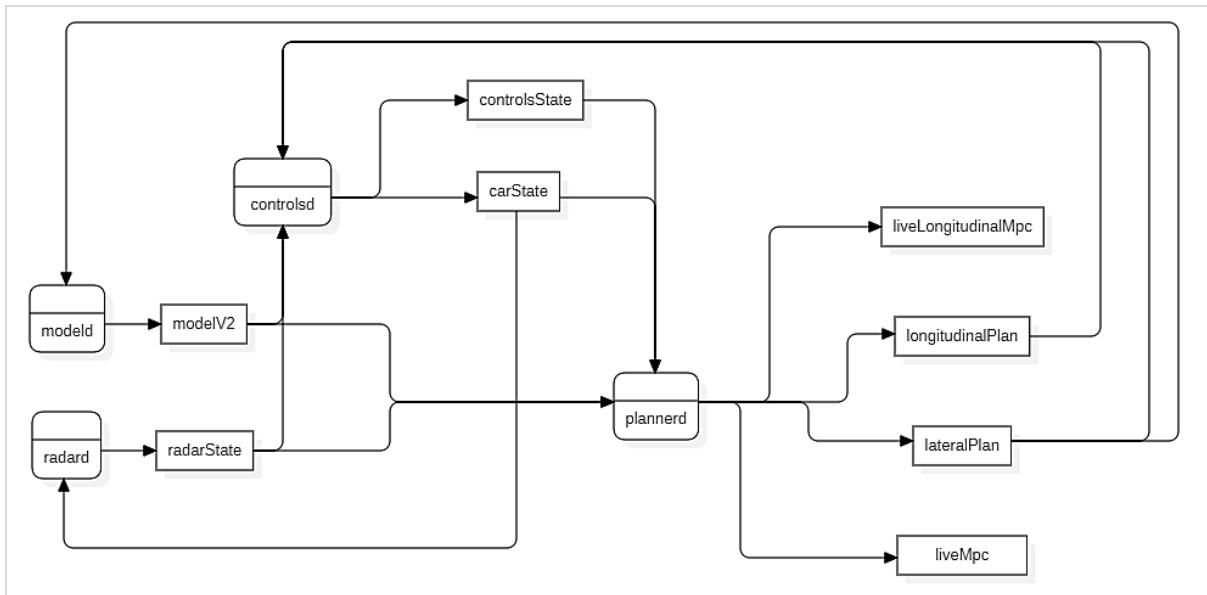


Figure 60 - *plannerd* data flow diagram

4.2.6 RadarD

This process parses the data acquired by the radar into a RadarState packet.

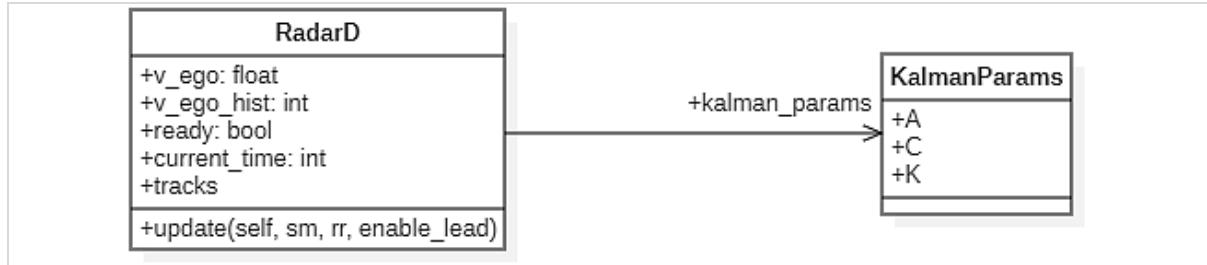


Figure 61 - RadarD class diagram

Since every car has a different radar, the radar interface is imported dynamically from the *selfdrive/car* folder at runtime. A RadarInterface object provides the details about the radar delays and acquisition intervals, as well as the methods to actually retrieve the radar acquisitions from the front and back radar, using a CANParser object to translate the messages traveling on the CAN bus.

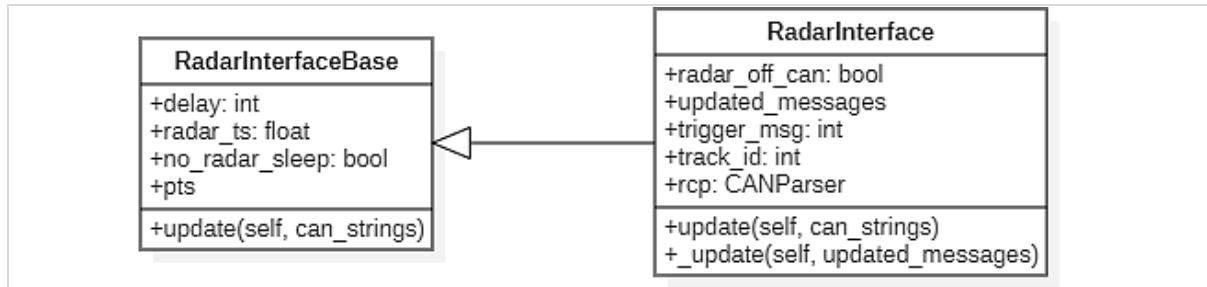


Figure 62 - RadarInterface class diagram

The main thread first subscribes to the socket *modelV2* and *carState*, needed to retrieve details on the timing recorded by the car for synchronization purposes. The process then initializes the RadarInterface, specific for the car in which Openpilot is being executed, and the main RadarD object, which represents an abstraction of the car's radar.

The core of the process is an endless loop which at each step retrieves the data acquired by the radar through its RadarInterface, obtaining a *RadarData* packet containing the details on the distance, speed, and optionally the acceleration of an object in front of the vehicle.

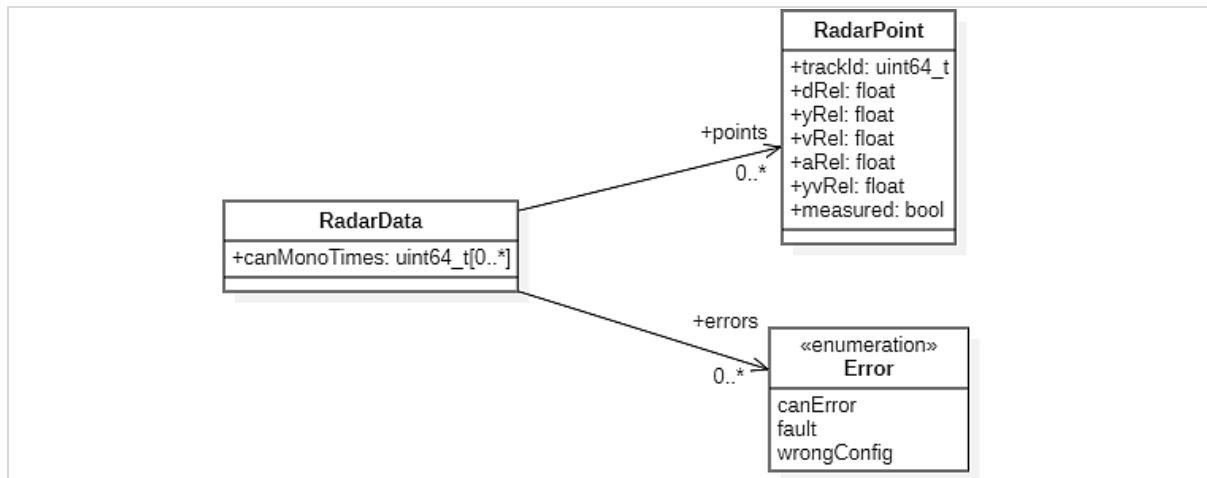


Figure 63 - RadarData message packet specifications

RadarD *update()* method retrieves the parameters acquired from the radar and creates the corresponding Track, which can be seen as a vector in space, having as direction the result of the composition of x and y and module equal to the speed.

Tracks are smoothed utilizing a simple Kalman filter, which has the role to take the current known state of the target and predict the new state of the target at the time of the most recent radar measurement. It then forms a weighted average of this prediction of state and the latest measurement of state, taking account of the known measurement errors of the radar and its uncertainty in the target motion models. A key assumption in the mathematics of the Kalman filter is that the measurement equations and the state equations are linear. [49]

A series of Tracks make a Cluster, which can be used to estimate more precisely the properties of a lead car. These estimations are then used to generate a RadasState message, where are specified all the details of a leading car, including its speed, acceleration, and distance from the car. To do so, the clusters are matched with the estimations coming from the AI model.

```
def get_lead(v_ego, ready, clusters, lead_msg, low_speed_override=True):
    if len(clusters) > 0 and ready and lead_msg.prob > .5:
        cluster = match_vision_to_cluster(v_ego, lead_msg, clusters)
    else:
        cluster = None

    [...]
```

The closest cluster is then selected and the created RadarState is sent over socket *radarState* to the planner and controls daemons.

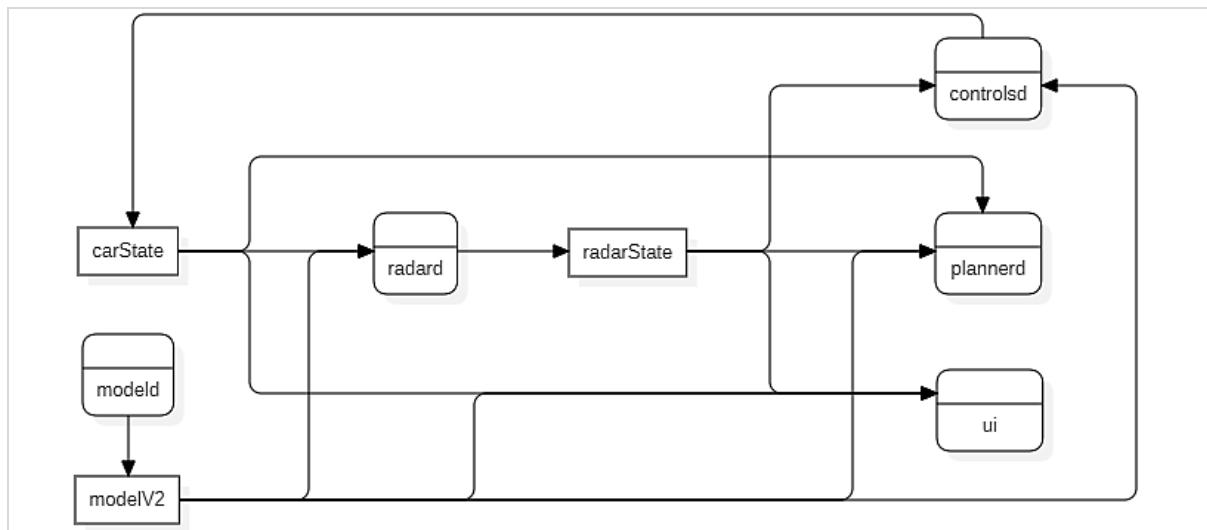
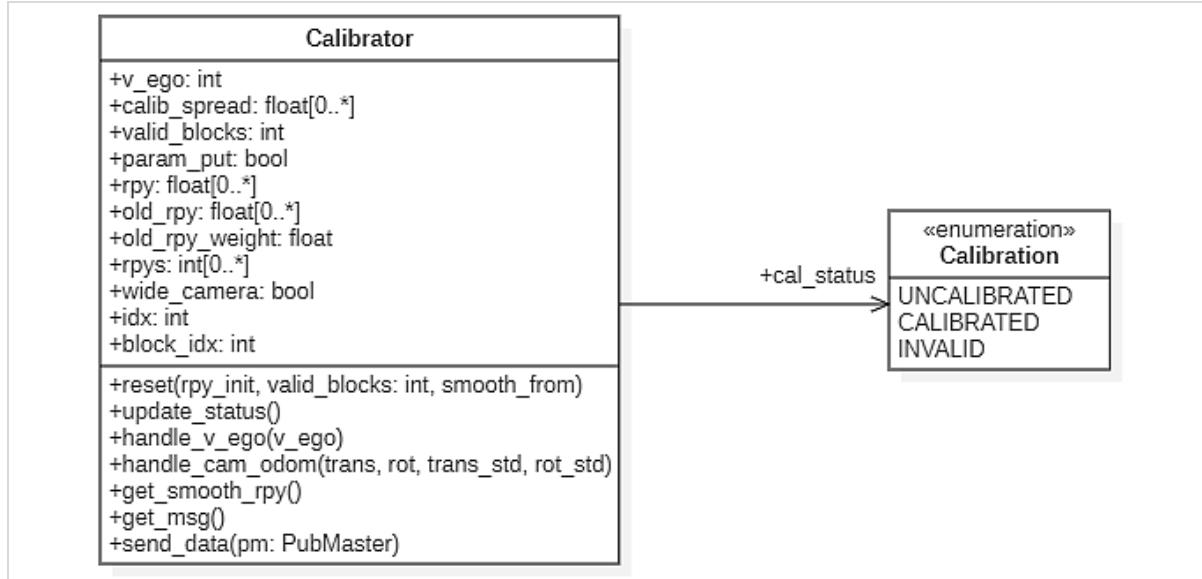


Figure 64 - radard data flow diagram

4.2.7 CalibrationD

This process canonicalizes the acquired frames by converting them into calibrated frames, which are then used by the other Openpilot components. This is important because users can mount their Comma devices in different positions and transforming them allows the model to ignore the error in the predictions that this could introduce.



The main thread of the calibration process sends data at a 4 Hz frequency containing the result of the calibration. For each frame, the algorithm first calculates the Euler angles, denoted by roll, pitch, and yaw, starting from the rotation and translation recorded thanks to visual odometry.

```

observed_rpy = np.array([0, #roll
                        -np.arctan2(trans[2], trans[0]), #pitch
                        np.arctan2(trans[1], trans[0]))] #yaw
  
```

The new calibrated Euler angles are obtained computing the dot (or scalar) product between the old measures of the angles and the new observed angles.

```

new_rpy = euler_from_rot(rot_from_euler(self.get_smooth_rpy())
                          .dot(rot_from_euler(observed_rpy)))
  
```

The methods to make the conversion from Euler angles to the corresponding rotation matrix (`rot_from_euler()`) and vice versa (`euler_from_rot()`) are provided by the transformation tools included in the `common` folder. These transformations are needed mainly to convert the old and new acquisitions of the Euler angles in the rotation matrixes that are used to calculate the scalar product between them.

In practice, the rotation matrixes on the three axis can be defined using the following equations:

$$R_x(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \psi & -\sin \psi \\ 0 & \sin \psi & \cos \psi \end{bmatrix} \quad (4.2)$$

$$R_y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix} \quad (4.3)$$

$$R_z(\varphi) = \begin{bmatrix} \cos \varphi & -\sin \varphi & 0 \\ \sin \varphi & \cos \varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (4.4)$$

where ψ , θ , and φ are the Euler angles. A general rotation matrix has the form of (4.5).

$$R = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \quad (4.5)$$

This matrix can be thought of as a sequence of three rotations, one about each principal axis.

$$\begin{aligned} R &= R_z(\varphi)R_y(\theta)R_x(\psi) \\ &= \begin{bmatrix} \cos \theta \cos \varphi & \sin \psi \sin \theta \cos \varphi - \cos \psi \sin \varphi & \cos \psi \sin \theta \cos \varphi + \sin \psi \\ \cos \theta \sin \varphi & \sin \psi \sin \theta \sin \varphi + \cos \psi \cos \varphi & \cos \psi \sin \theta \sin \varphi - \sin \psi \\ -\sin \theta & \sin \psi \cos \theta & \cos \psi \cos \theta \end{bmatrix} \quad (4.6) \end{aligned}$$

Given a rotation matrix R , we can compute the Euler angles, ψ , θ , and φ by equating each element in R with the corresponding element in the matrix product $R_z(\varphi)R_y(\theta)R_x(\psi)$.

The calibration is made by sampling multiple camera acquisitions and computing the average of the values of the available valid blocks. Each block is made of a hundred acquisitions.

```
self.rpy = np.mean(self.rpys[:self.valid_blocks], axis=0)

def get_smooth_rpy(self):
    if self.old_rpy_weight > 0:
        return self.old_rpy_weight * self.old_rpy +
               (1.0 - self.old_rpy_weight) *
               self.rpy
    else:
        return self.rpy
```

The minimum number of blocks required for the calibration is five, even if the optimal amount would be fifty blocks (or more) for stability purposes. If the number of valid blocks is not enough while computing a calibrated frame, its calibration status will be set to *uncalibrated*, it is set to *valid* otherwise. The generated packet, sent over socket *liveCalibration*, includes the details on the calibrated angles, the calibration state, and the number of blocks on which the calibration is based.

```
msg = messaging.new_message('liveCalibration')
msg.liveCalibration.validBlocks = self.valid_blocks
msg.liveCalibration.calStatus = self.cal_status
msg.liveCalibration.calPerc = min(100 *
                                  (self.valid_blocks*BLOCK_SIZE+self.idx) //
                                  (INPUTS_NEEDED * BLOCK_SIZE), 100)
msg.liveCalibration.extrinsicMatrix = [float(x) for x
                                         in extrinsic_matrix.flatten()]
msg.liveCalibration.rpyCalib = [float(x) for x in smooth_rpy]
msg.liveCalibration.rpyCalibSpread=[float(x) for x in self.calib_spread]
```

Calibrated parameters can also be cached as *CalibrationParams* by using the Params class utility.

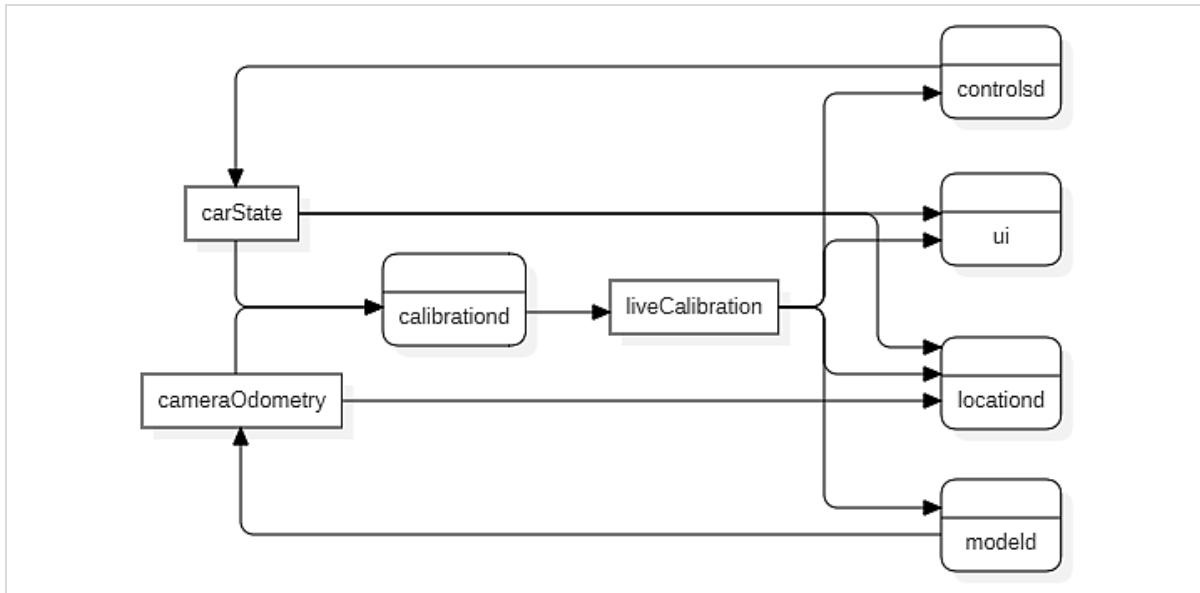


Figure 65 - *calibrationd* data flow diagram

4.2.8 LocationD

This process runs a global localizer, which estimates the vehicle position, speed, and acceleration and how they change in the three dimensions. It combines the data coming from multiple sources, including the camera, the GPS and inertial measurement unit (IMU) sensors.

The Localizer class [Figure 66] provides the methods to handle the different types of events recorded and process the corresponding messages.

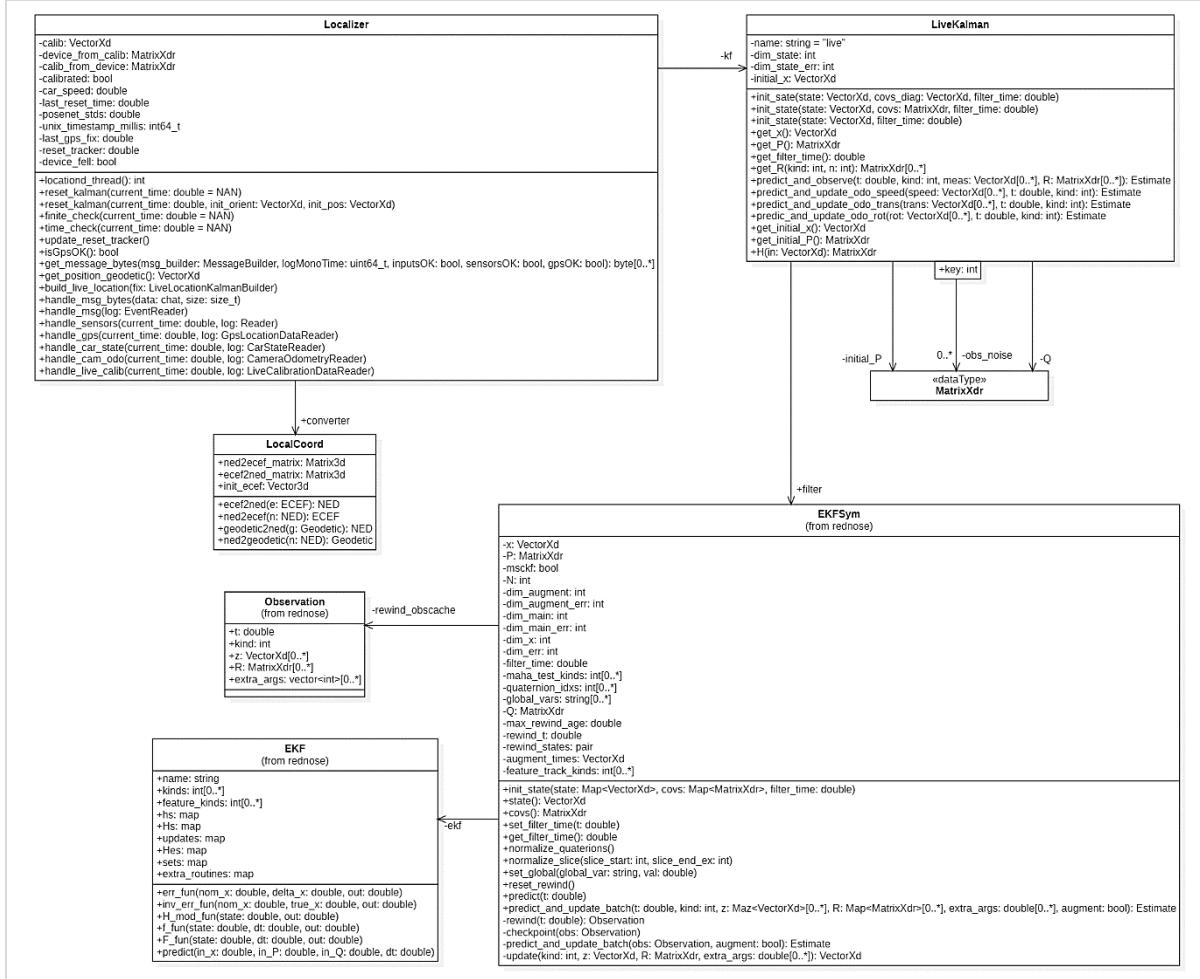


Figure 66 - locationd class diagram. The EKFSym component from Rednose library helps smoothing the acquired data.

The Localizer will handle the received events by checking first their type.

```

if (log.isSensorEvents()) {
    this->handle_sensors(t, log.getSensorEvents());
} else if (log.isGpsLocationExternal()) {
    this->handle_gps(t, log.getGpsLocationExternal());
} else if (log.isCarState()) {
    this->handle_car_state(t, log.getCarState());
} else if (log.isCameraOdometry()) {
    this->handle_cam_odo(t, log.getCameraOdometry());
} else if (log.isLiveCalibration()) {
    this->handle_live_calib(t, log.getLiveCalibration());
}
    
```

The sensors from which a message can be received are the uncalibrated gyroscope and the accelerometer. The uncalibrated gyroscope, which registers the **rotation** of the car on the three axis, is more indicated than the calibrated one to acquire data since they will be calibrated dynamically by the algorithm. The accelerometer, instead, measures the **acceleration** of the car on the three axis.

When a GPS message is received, the acquired data are first converted in ECEF coordinates, since this coordinates system is commonly adopted as the standard for global positioning systems. The acquired data allow determining the **position** and **speed** of the vehicle. The speed is also retrieved from the carState.

The messages related to the camera's acquisitions allow computing, through visual odometry, the vehicle's **rotation** and **translation**.

The process also retrieves the calibrated frames, elaborated by *calibrationD*, which can be defined as frames aligned with car frame in pitch and yaw, and aligned with device frame in roll. This transformation is useful to overcome small inconsistencies that could appear from a device to another, allowing to normalize the acquired frames.

In all the cases, the acquisitions are compared with strict safety checks that skip the message if the safety constraints are exceeded. Also, in all the cases the estimations are computed using an Extended Kalman Filter, provided by the Rednose library.

If all the camera, sensors, and GPS acquisitions are alive and valid, then a message is built and sent over socket *liveLocationKalman*. The generated message will contain a set of Measurement (a type defined in Cereal) corresponding to all the estimations made by the process at each frame.

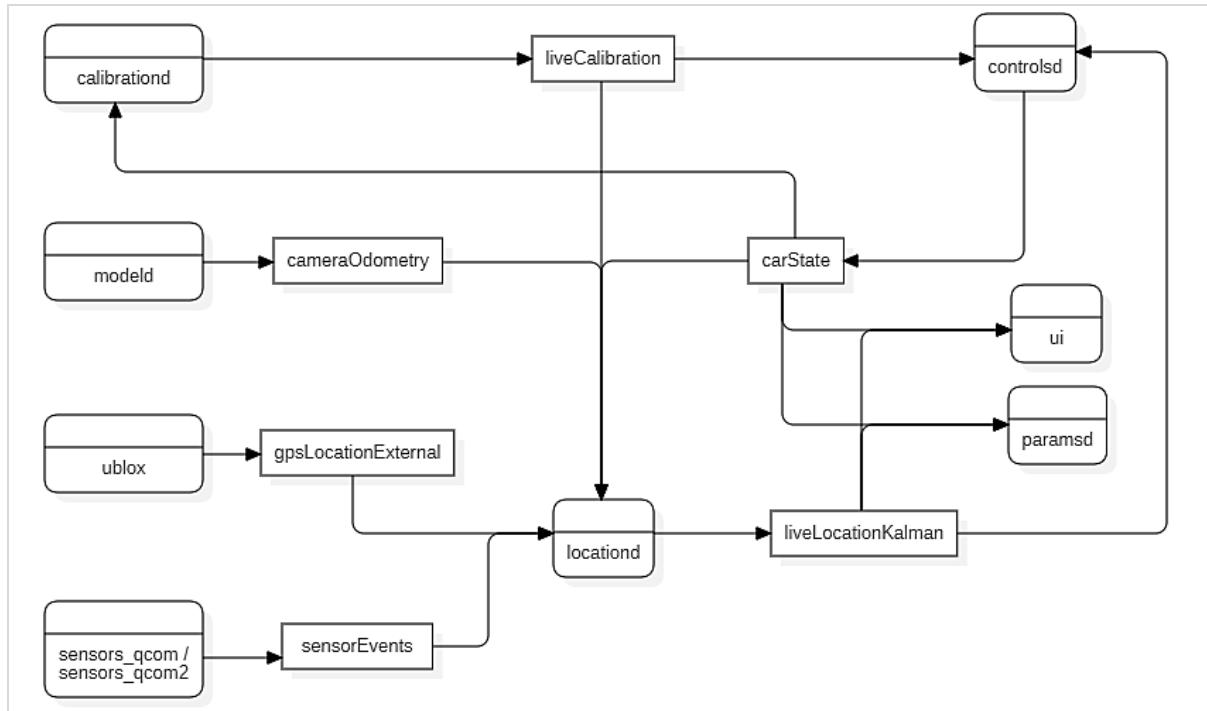


Figure 67 - locationd data flow diagram

4.2.9 UbloxD

Comma devices come with a u-blox chip, which is capable of acquiring data from up to three GNSS concurrently, granting a high level of accuracy.

u-blox data are acquired by the Panda, published on socket *ubloxRaw*, and parsed by the *UbloxMsgParser* component.

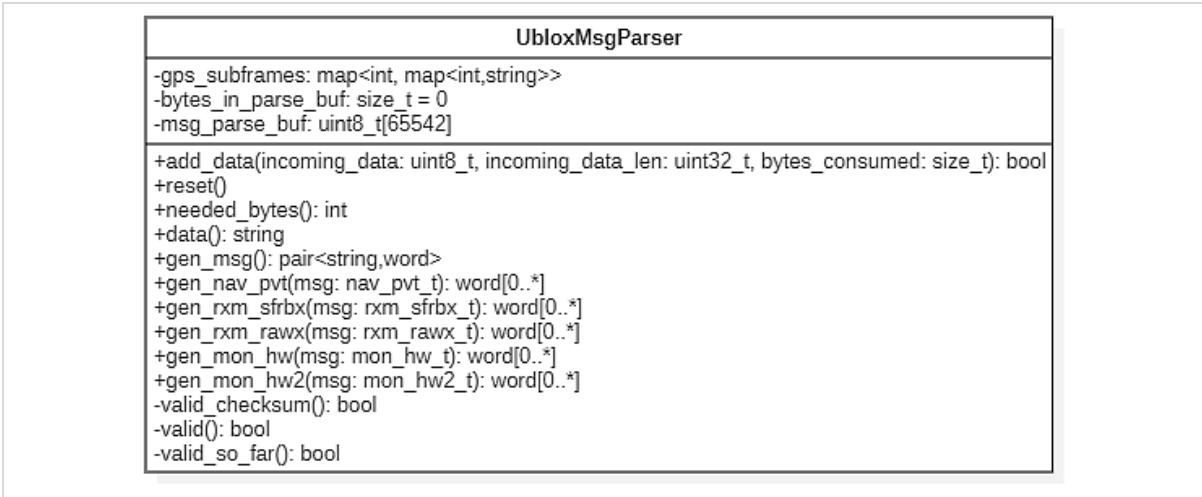


Figure 68 - *UbloxMsgParser* class

The parser has a buffer that is filled until the raw message is fully parsed. It may be needed to fill the buffer multiple times if the raw message size exceeds the buffer capacity.

According to the message type, the parser uses a different method to extract usable content from the raw data.

```
switch (ubx_message.msg_type()) {
case 0x0107:
    return {"gpsLocationExternal", gen_nav_pvt(static_cast<
nav_pvt_t*>(body))};
    break;
case 0x0213:
    return {"ubloxGnss", gen_rxm_sfrbx(static_cast<rxm_sfrbx_t*>(body))};
    break;
case 0x0215:
    return {"ubloxGnss", gen_rxm_rawx(static_cast< rxm_rawx_t*>(body))};
    break;
case 0xa09:
    return {"ubloxGnss", gen_mon_hw(static_cast<mon_hw_t*>(body))};
    break;
case 0xa0b:
    return {"ubloxGnss", gen_mon_hw2(static_cast< mon_hw2_t*>(body))};
    break;
default:
    LOGE("Unknown message type %x", ubx_message.msg_type());
    return {"ubloxGnss", kj::Array<capnp::word>()};
    break;
}
```

The methods displayed handle the different types of GNSS data arriving from the supported satellites constellations, including GPS, GLONASS, BeiDou, and Galileo.

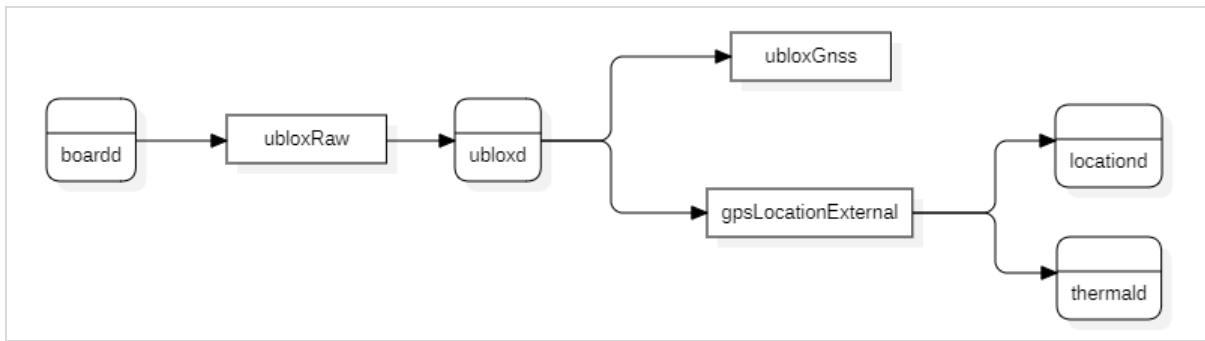


Figure 69 - ubloxd data flow diagram

4.2.10 ModelD

The main model takes in a picture from the road camera and answers the question “Where should I drive the car?” It also takes in a *desire* input, which can command the model to act, such as turning or changing lanes.

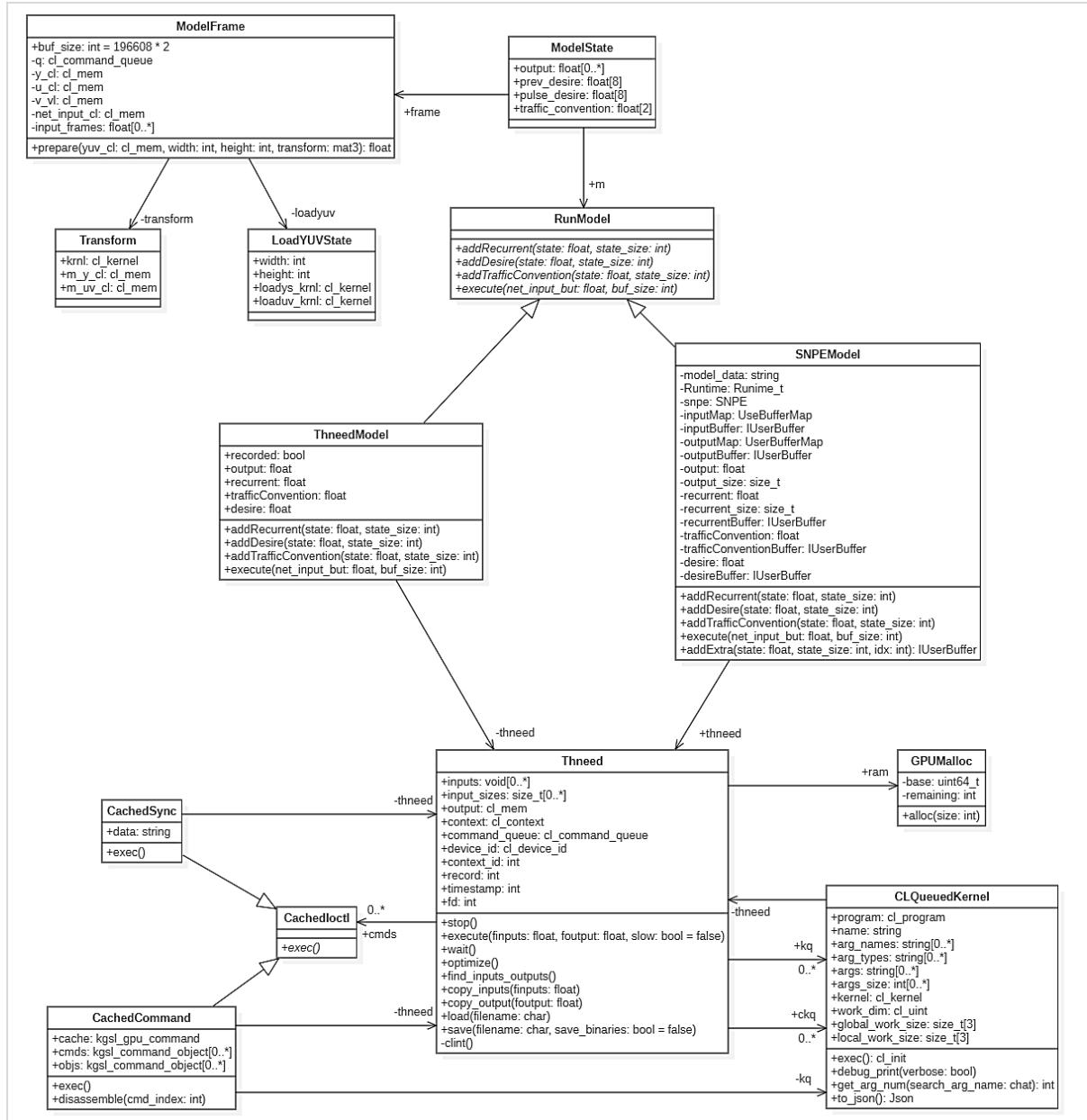


Figure 70 - modeld class diagram. RunModel component can use SNPEModel component, that leverages only the Snapdragon processor capabilities, or use the caching system provided by Thneed.

The input of the model is retrieved by a thread that takes the data from the socket *liveCalibration* containing the calibrated frames elaborated by the process *calibrationd*.

The representation of the camera frames is built by considering both the intrinsic parameters, that are the camera parameters that are internal and fixed to a particular camera/digitization setup, and the extrinsic parameters, that are the camera parameters that are external to the camera and may change concerning the world frame.

If the intrinsic parameters define the location and orientation of the camera concerning the world frame, the intrinsic Parameters allow a mapping between camera coordinates and pixel coordinates in the image frame.

Intrinsic and extrinsic parameters can be represented by using the matrixes (4.7) and (4.8), respectively:

$$M_{int} = \begin{bmatrix} -f/s_x & 0 & o_x \\ 0 & -f/s_y & o_y \\ 0 & 0 & 1 \end{bmatrix} \quad (4.7)$$

$$M_{ext} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & -R_1^T T \\ r_{21} & r_{22} & r_{23} & -R_1^T T \\ r_{31} & r_{32} & r_{33} & -R_1^T T \end{bmatrix} \quad (4.8)$$

where f is the focal length, (o_x, o_y) is the image center, (s_x, s_y) is the effective size of pixels in the horizontal and vertical direction, R is the rotation matrix, and T is the translation vector. [50]

In the specific case, the data generated by the calibration process represent the extrinsic parameters, while the intrinsic parameters are hardcoded and depend on the hardware (Comma Two and Comma Three have different camera setups, therefore different intrinsic matrixes).

The linear perspective transformation can be defined as the product of the two matrixes.

```
auto camera_frame_from_road_frame = cam_intrinsics *
extrinsic_matrix_eigen;
```

The matrix has to be further processed to consider the position of the camera, which is placed higher than the ground level.

```
Eigen::Matrix<float, 3, 3> camera_frame_from_ground;
camera_frame_from_ground.col(0) =
camera_frame_from_road_frame.col(0);
camera_frame_from_ground.col(1) =
camera_frame_from_road_frame.col(1);
camera_frame_from_ground.col(2) =
camera_frame_from_road_frame.col(2);

auto warp_matrix = camera_frame_from_ground *
ground_from_medmodel_frame;
```

The processed calibrated frames are then used by the model to make the predictions. The actual frames are sent using VisionIPC and can be corrected by using the calibrated frame information computed by the thread managing the calibration.

The VisionIPC client connects to the socket *camerad* and the model is initialized. The model which runs the neural network to make the predictions is called *Supercombo* and as the default setting it uses the Snapdragon Neural Processing Engine (SNPE), which is a Qualcomm Snapdragon software accelerated runtime for the execution of deep neural networks.

```

#ifndef QCOM
    s->m = make_unique<ThneedModel>("../models/supercombo.thneed",
        &s->output[0], output_size, USE_GPU_RUNTIME);
#else
    s->m = make_unique<DefaultRunModel>("../models/supercombo.dlc",
        &s->output[0], output_size, USE_GPU_RUNTIME);
#endif

```

To make the model faster and more efficient, it was also introduced Thneed, an SNPE accelerator. The Thneed model runs on the local device, caching a single model run and replaying it at a higher speed. Apart from the caching functionalities, the Thneed does not change any aspect of the default SNPE. The specifics of the model and the definition of its inputs and outputs are defined in the official Openpilot wiki [51].

The input parameters of the model are:

- **image stream:** two consecutive images ($256 * 512 * 3$ in RGB) recorded at 20 Hz:
 $393216 = 2 * 6 * 128 * 256$
 - Each $256 * 512$ image is represented in YUV420 with 6 channels: $6 * 128 * 256$
 - Channels 0, 1, 2, 3 represent the full-res Y channel and are represented in numpy as $Y[::2, ::2]$, $Y[::2, 1::2]$, $Y[1::2, ::2]$, and $Y[1::2, 1::2]$
 - Channel 4 represents the half-res U channel
 - Channel 5 represents the half-res V channel
- **desire:** one-hot encoded vector to command model to execute certain actions, bit only needs to be sent for 1 frame : 8
- **traffic convention:** one-hot encoded vector to tell model whether traffic is right-hand or left-hand traffic : 2
- **recurrent state:** The recurrent state vector that is fed back into the GRU for temporal context: 512

The produced output is composed of:

- **plan:** 5 potential desired plan predictions, for a total of $4955 = 5 * 991$ possibilities
 - predicted mean and standard deviation of the following values at 33 timesteps:
 $990 = 2 * 33 * 15$
 - (x, y, z) position in current frame (*meters*)
 - (x, y, z) velocity in local frame (*meters/s*)
 - (x, y, z) acceleration local frame (*meters/(s * s)*)
 - (r, p, y) roll, pitch , yaw in current frame (*radians*)
 - (r, p, y) roll, pitch , yaw rates in local frame (*radians/s*)
 - probability of this plan hypothesis being the most likely: 1
- **lane lines:** lane lines (outer left, left, right, and outer right): $528 = 4 * 132$
 - predicted mean and standard deviation for the following values at 33 x positions:
 $132 = 2 * 33 * 2$
 - y position in current frame (*meters*)
 - z position in current frame (*meters*)
- **lane line probabilities:** 2 probabilities that each of the 4 lane lines exists: $8 = 4 * 2$
 - deprecated probability
 - used probability
- **road-edges:** 2 road-edges (left and right): $264 = 2 * 132$

- predicted mean and standard deviation for the following values at 33 x positions:
 $132 = 2 * 33 * 2$
 - y position in current frame (*meters*)
 - z position in current frame (*meters*)
- **leads:** 2 hypotheses for potential lead cars: $102 = 2 * 51$
 - predicted mean and standard deviation for the following values at 0, 2, 4, 6, 8, 10s: $48 = 2 * 6 * 4$
 - x position of lead in current frame (*meters*)
 - y position of lead in current frame (*meters*)
 - speed of lead (*meters/s*)
 - acceleration of lead(*meters/(s * s)*)
 - probabilities this hypothesis is the most likely hypothesis at 0s, 2s or 4s from now : 3
- **lead probabilities:** probability that there is a lead car at 0s, 2s, 4s from now: $3 = 1 * 3$
- **desire state:** probability that the model thinks it is executing each of the 8 potential desire actions : 8
- **meta:** Various metadata about the scene: $80 = 1 + 35 + 12 + 32$
 - Probability that Openpilot is engaged: 1
 - Probabilities of various things happening between now and 2, 4, 6, 8, 10s: $35 = 5 * 7$
 - Disengage of Openpilot with gas pedal
 - Disengage of Openpilot with brake pedal
 - Override of Openpilot steering
 - $3 m/(s * s)$ of deceleration
 - $4 m/(s * s)$ of deceleration
 - $5 m/(s * s)$ of deceleration
 - Probabilities of left or right blinker being active at 0, 2, 4, 6, 8, 10s: $12 = 6 * 2$
 - Probabilities that each of the 8 desires is being executed at 0, 2, 4, 6s: $32 = 4 * 8$
- **pose:** predicted mean and standard deviation of current translation and rotation rates: $12 = 2 * 6$
 - (x, y, z) velocity in the current frame (*meters/s*)
 - (r, p, y) roll, pitch, yaw rates in the current frame (*radians/s*)
- **recurrent state:** the recurrent state vector that is fed back into the GRU for temporal context: 512

The model is trained using the comma2k19 dataset, a dataset of over 33 hours of drive. The dataset contains 10.000 camera acquisitions, which were manually labeled to the different elements of the frames. The features vectors are then fed to the Recurrent Neural Networks that generate the driving policy.

For feature extraction from frames, Comma uses a lot of skip connections and converts a 12 x 128 x 256 (two YUV format consecutive frames with 3 extra alpha channels).

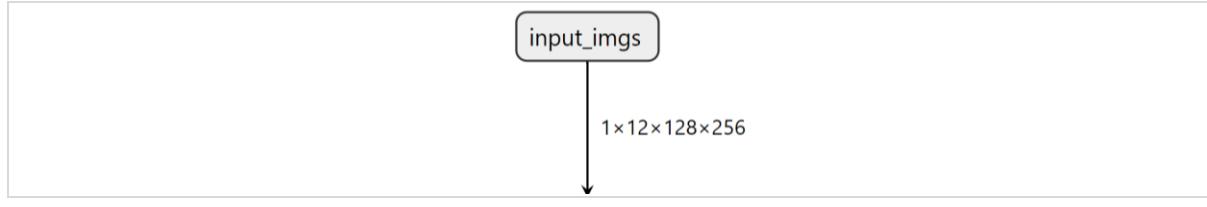


Figure 71 - supercombo model input

The model learns to encode all the relevant information required for planning into a compressed form. This vision encoding is later forked into several branches which are processed independently to output lanes, paths, etc.

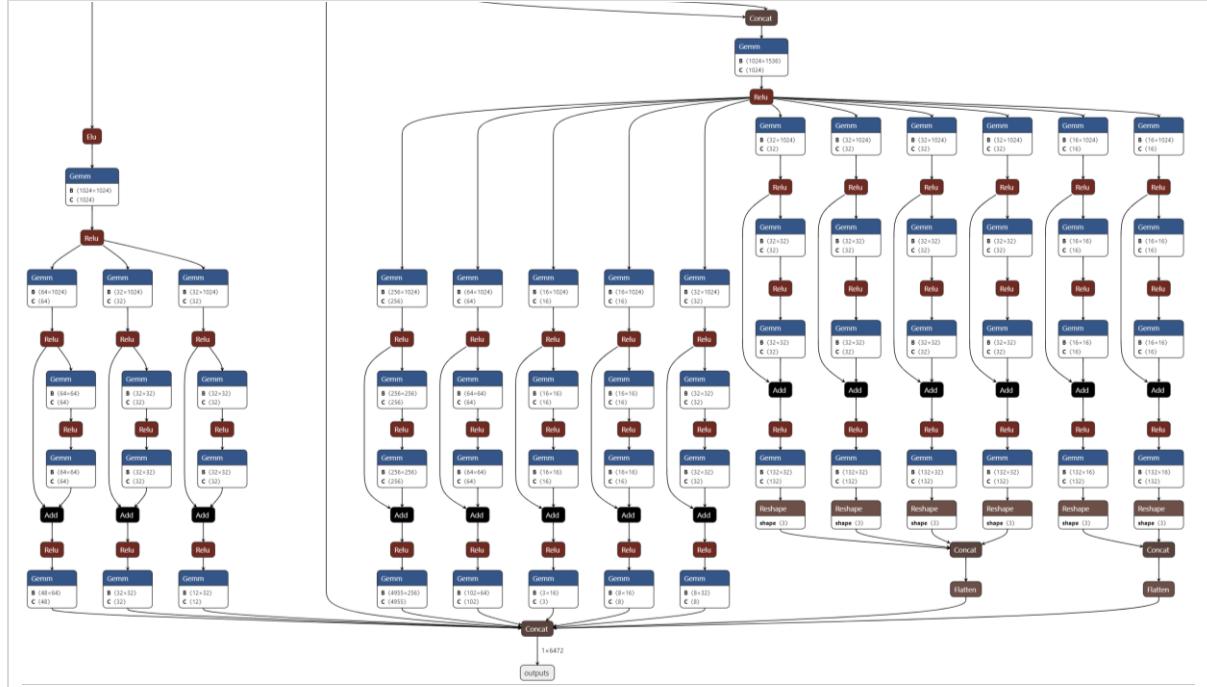


Figure 72 - supercombo model output

When the model receives the YUV frames from the VisionIPC server, they are first prepared and transformed using the transformation matrix obtained after processing the calibrated frame. This preparation step initializes the ModelFrame, which is a virtual camera frame with even width and height frame that can be used by the model.

The size of the frame is defined as $MODEL_WIDTH * MODEL_HEIGHT * 3 / 2$, where the $3/2$ factor takes into account the fact that YUV has one channel in full resolution and two color channels in half resolution, therefore the size of the buffer will be $1 + 0.25 + 0.25 = 1.5 = 3/2$ of the total amount of pixels, represented by $MODEL_WIDTH * MODEL_HEIGHT$. The model takes 2 frames at a time, so that it is able to have a perception of the motion and make better predictions.

```
auto net_input_buf = s->frame->prepare(yuv_c1,
                                         width,
                                         height,
                                         transform);
```

After the ModelFrame relative to the acquired YUV frames are ready, the model execution starts.

```
s->m->execute(net_input_buf, s->frame->buf_size);
```

If Thneed is used, a full execution is required only the first time since then the outputs are recorded and can be used to make faster predictions.

```
if (!recorded) {
    thneed->record = THNEED_RECORD;
    thneed->copy_inputs(inputs);
    thneed->clexec();
    thneed->copy_output(output);
    thneed->stop();
    recorded = true;
} else {
    thneed->execute(inputs, output);
}
```

The model, through the directory *ioctl()* (input/output control), can communicate with the Qualcomm Kernel Graphic Support Layer (KGSL). It also leverages the OpenCL functionalities supported by the Qualcomm platform to increase the performance and efficiency of the computation of the predictions. When the execution of the model is triggered, the command *clEnqueueNDRangeKernel()* enqueues a command to execute the OpenCL kernel on the device. The cached commands are elaborated by the KGSL and the result is retrieved from the OpenCL read buffer.

```
return clEnqueueNDRangeKernel(thneed->command_queue,
                               kernel,
                               work_dim,
                               NULL,
                               global_work_size,
                               local_work_size,
                               0,
                               NULL,
                               NULL);
```

The instruction on what type of predictions the model has to make are contained in the *desire*, which is computed by the lateral planner and indicates on what direction the car should proceed. A desire can assume different values that are defined in the Cereal messaging specification, shown in Figure 73.

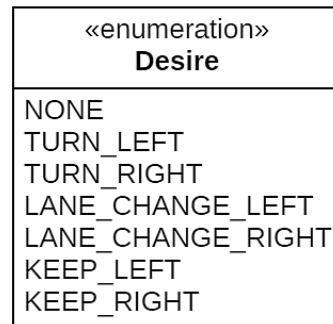


Figure 73 - Desire message specifications

From the output data is then extracted and published on socket *modelV2*.

```
ModelDataRaw net_outputs;
net_outputs.plan = &s->output[PLAN_IDX];
net_outputs.lane_lines = &s->output[LL_IDX];
net_outputs.lane_lines_prob = &s->output[LL_PROB_IDX];
net_outputs.road_edges = &s->output[RE_IDX];
net_outputs.lead = &s->output[LEAD_IDX];
net_outputs.lead_prob = &s->output[LEAD_PROB_IDX];
net_outputs.meta = &s->output[DESIRE_STATE_IDX];
net_outputs.pose = &s->output[POSE_IDX];
```

The model also runs posenet to generate the visual odometry parameters, estimated from the model output and published on socket *cameraOdometry*. With posenet we generally refer to computer vision techniques that estimate the position of an object, in the specific case of the rotation and translations of the frame estimated by the model.

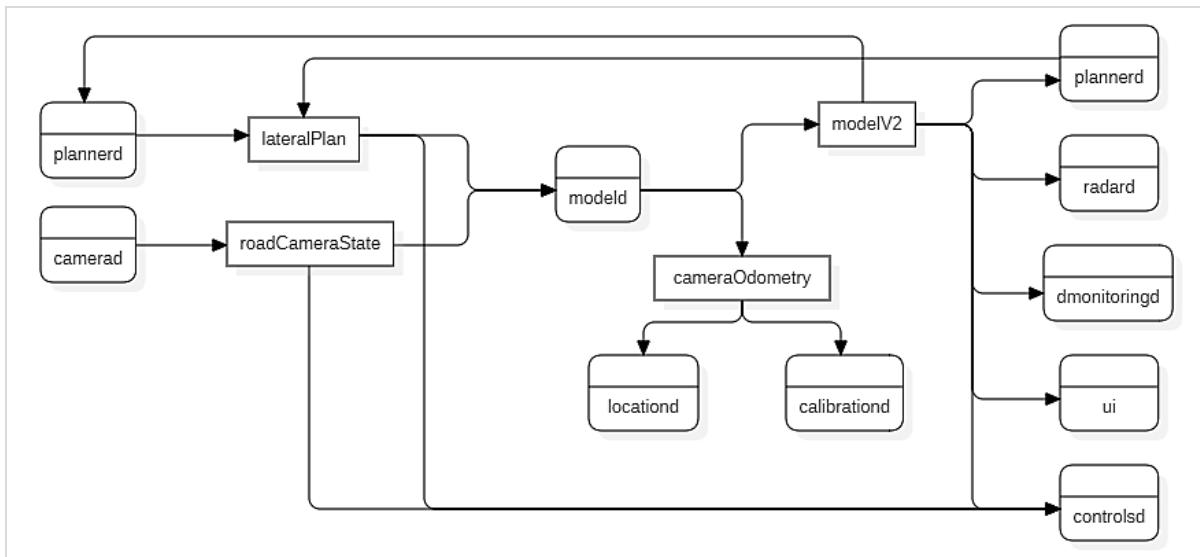


Figure 74 - *modeld* data flow diagram

4.2.11 DMonitoringModelID

The Driver Monitoring Model tracks the head pose, eye positions, and eye states using the model in models/monitoring_model_q.dlc. It runs on the Digital Signal Processor (DSP) to not use CPU or GPU resources needed by the other daemons, giving it of room to grow.

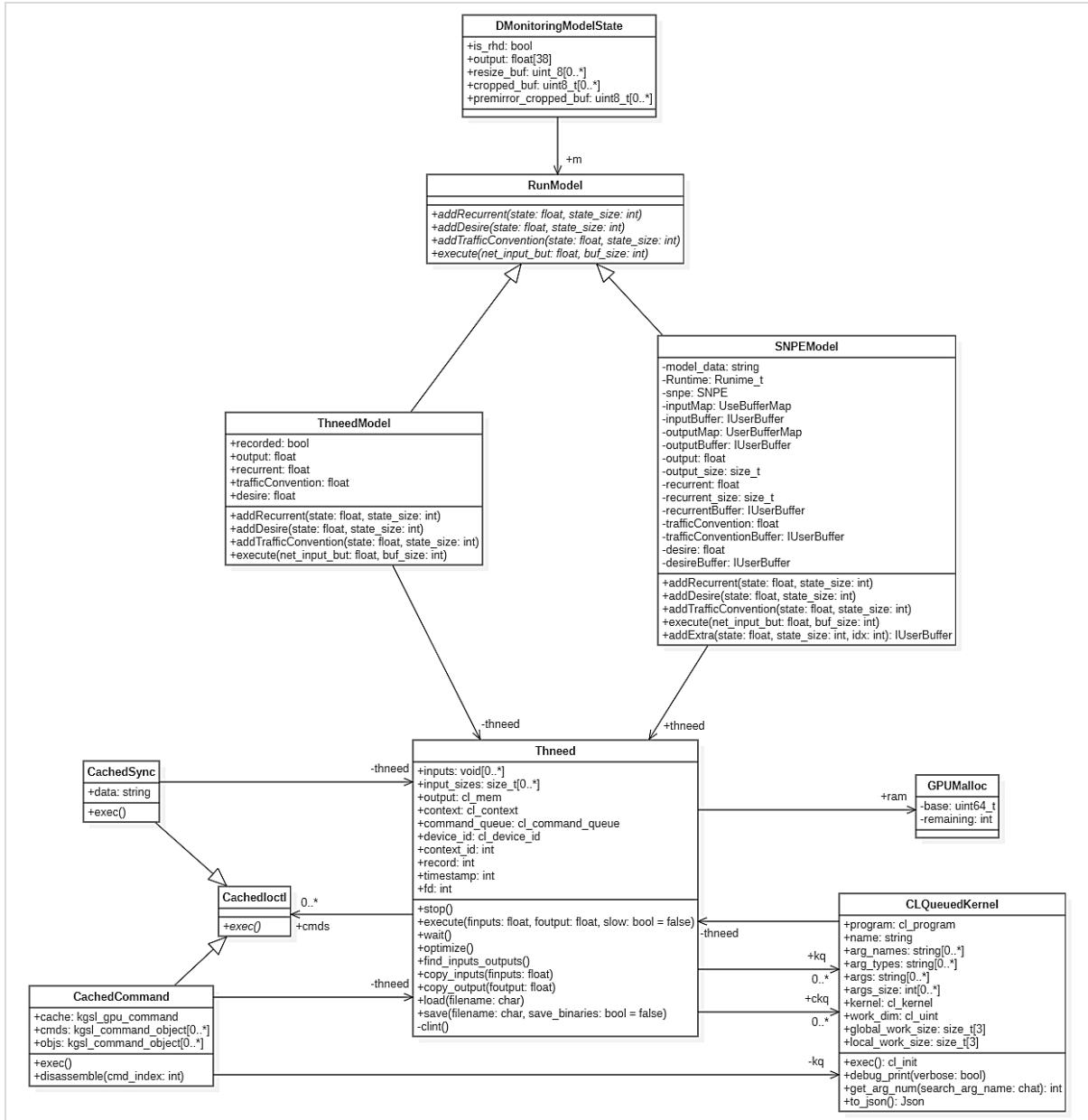


Figure 75 - dmonitoring class diagram. Similarly to the path prediction model, also here there is the possibility for RunModel to use SNPEModel only or Thneed.

The model receives the YUV frames through VisionIPC.

```

VisionIpcClient v ipc_client = VisionIpcClient("camerad",
                                               VISION_STREAM_YUV_FRONT,
                                               true);
  
```

The frames are acquired by the front camera of the Comma device and record the driver movements. The model is built using the RunModel component, the same used for building the *Supercombo* model, but in this case the model built is the *monitoring_model_q*.

The input of the model consists of six channels images of resolution 320x160, where channel 0-3 is the top left, bottom left, top right, the bottom right pixel of every four pixels of Y block, and channels 4-5 are U and V.

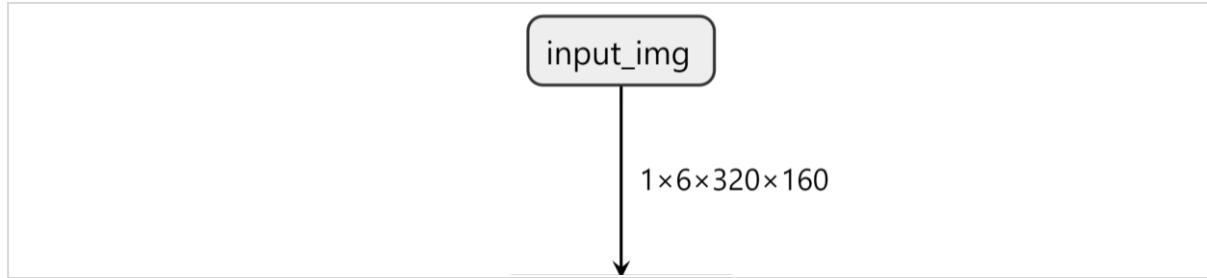


Figure 76 - monitoring_model_q input

The model elaborates each frame and computes the output containing the information about the driver head pose, eye position, it estimates the driver distraction and gives weight to all of these estimations.

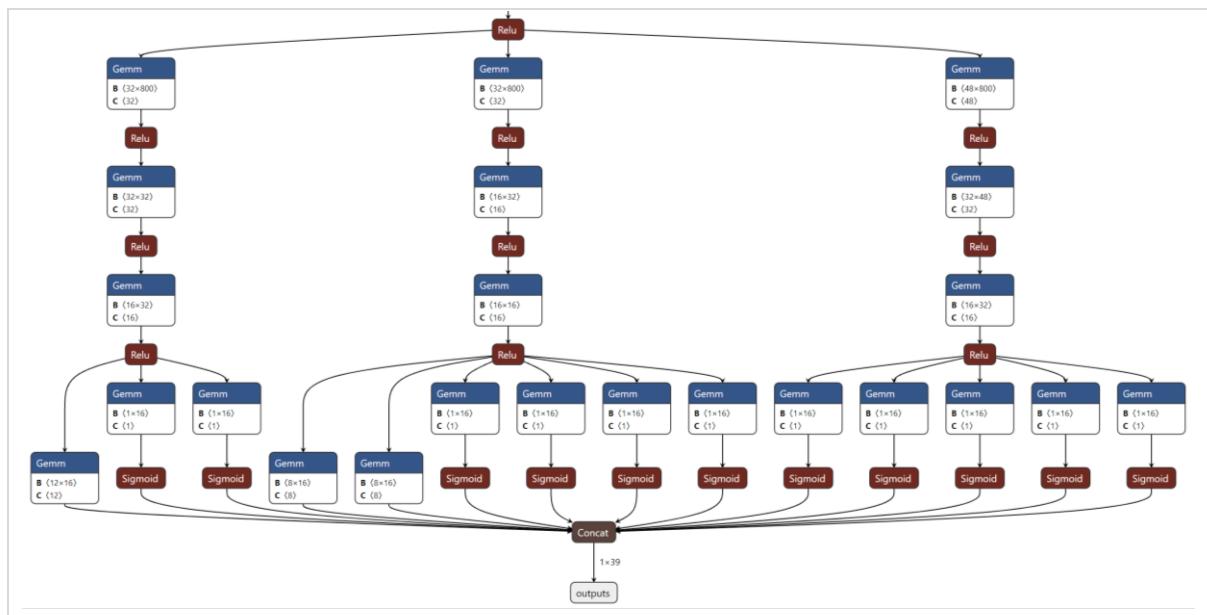


Figure 77 - monitoring_model_q output

```

ret.face_prob = s->output[12];
ret.left_eye_prob = s->output[21];
ret.right_eye_prob = s->output[30];
ret.left_blink_prob = s->output[31];
ret.right_blink_prob = s->output[32];
ret.sg_prob = s->output[33];
ret.poor_vision = s->output[34];
ret.partial_face = s->output[35];
ret.distracted_pose = s->output[36];
ret.distracted_eyes = s->output[37];
ret.dsp_execution_time = (t2 - t1) / 1000.;
```

The output is packed in a message and sent over *driverState* socket, ad elaborated by the *driverMonitoringD* process, which interprets the data and disengages Openpilot if the parameters indicating the driver distractions surpass a certain threshold.

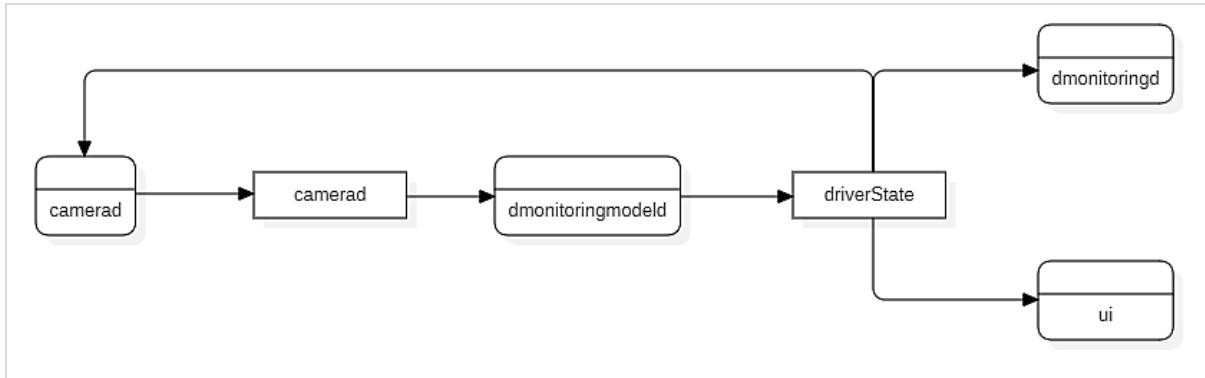


Figure 78 - *dmonitoringmodeld* data flow diagram

4.2.12 DMonitoringD

The driver monitoring process takes the data elaborated from the driver monitoring model and the other component monitoring the status of Openpilot. The status variables are held by an instance of the DriverStatus class, which includes all the details relative to the state of awareness of the driver.

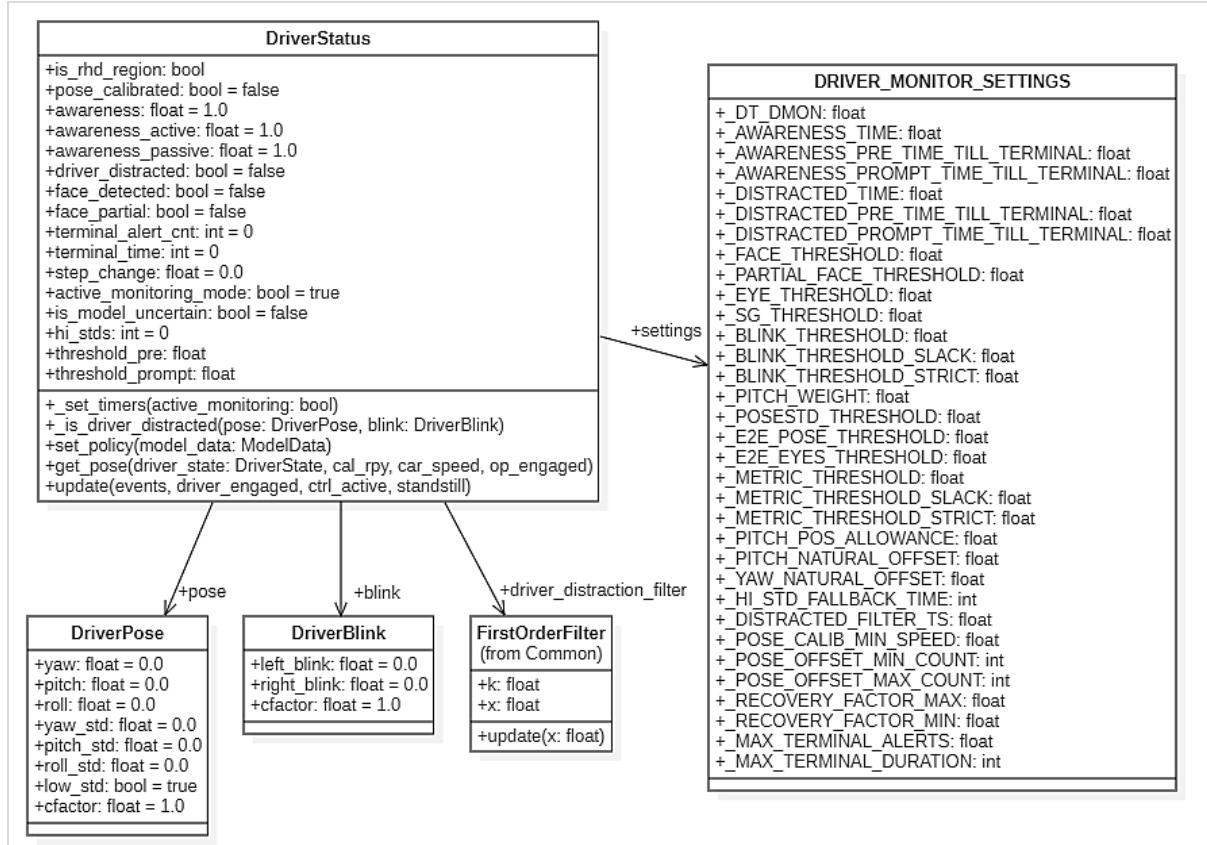


Figure 79 - DriverStatus class diagram

At each iteration, the process updates the driver status according to the driver awareness and the controls status.

```

driver_status.update(events,
                     driver_engaged,
                     sm['controlsState'].enabled,
                     sm['carState'].standstill)
  
```

The message containing the updated status details is built and sent over socket `driverMonitoringState`.

```
dat.driverMonitoringState = {
    "events": events.to_msg(),
    "faceDetected": driver_status.face_detected,
    "isDistracted": driver_status.driver_distracted,
    "awarenessStatus": driver_status.awareness,
    "posePitchOffset": driver_status.pose.pitch_offseter.filtered_stat.mean(),
    "posePitchValidCount": driver_status.pose.pitch_offseter.filtered_stat.n,
    "poseYawOffset": driver_status.pose.yaw_offseter.filtered_stat.mean(),
    "poseYawValidCount": driver_status.pose.yaw_offseter.filtered_stat.n,
    "stepChange": driver_status.step_change,
    "awarenessActive": driver_status.awareness_active,
    "awarenessPassive": driver_status.awareness_passive,
    "isLowStd": driver_status.pose.low_std,
    "hiStdCount": driver_status.hi_stds,
    "isActiveMode": driver_status.active_monitoring_mode,
}
```

The state of the driver is then retrieved by the control daemon that disengages Openpilot in the case in which the driver is not focused on the drive.

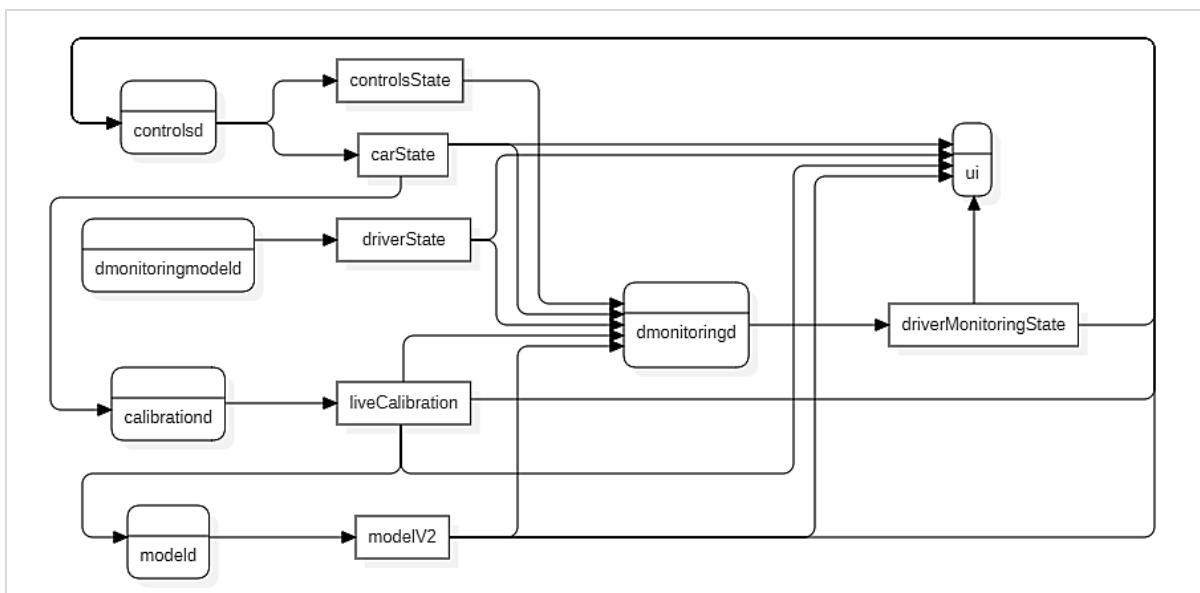


Figure 80 - DMonitoringD data flow diagram

4.2.13 LoggerD

This daemon subscribes to all the sockets and log all the messages intercepted. It also subscribes to all the device's camera and saves the drive recording.

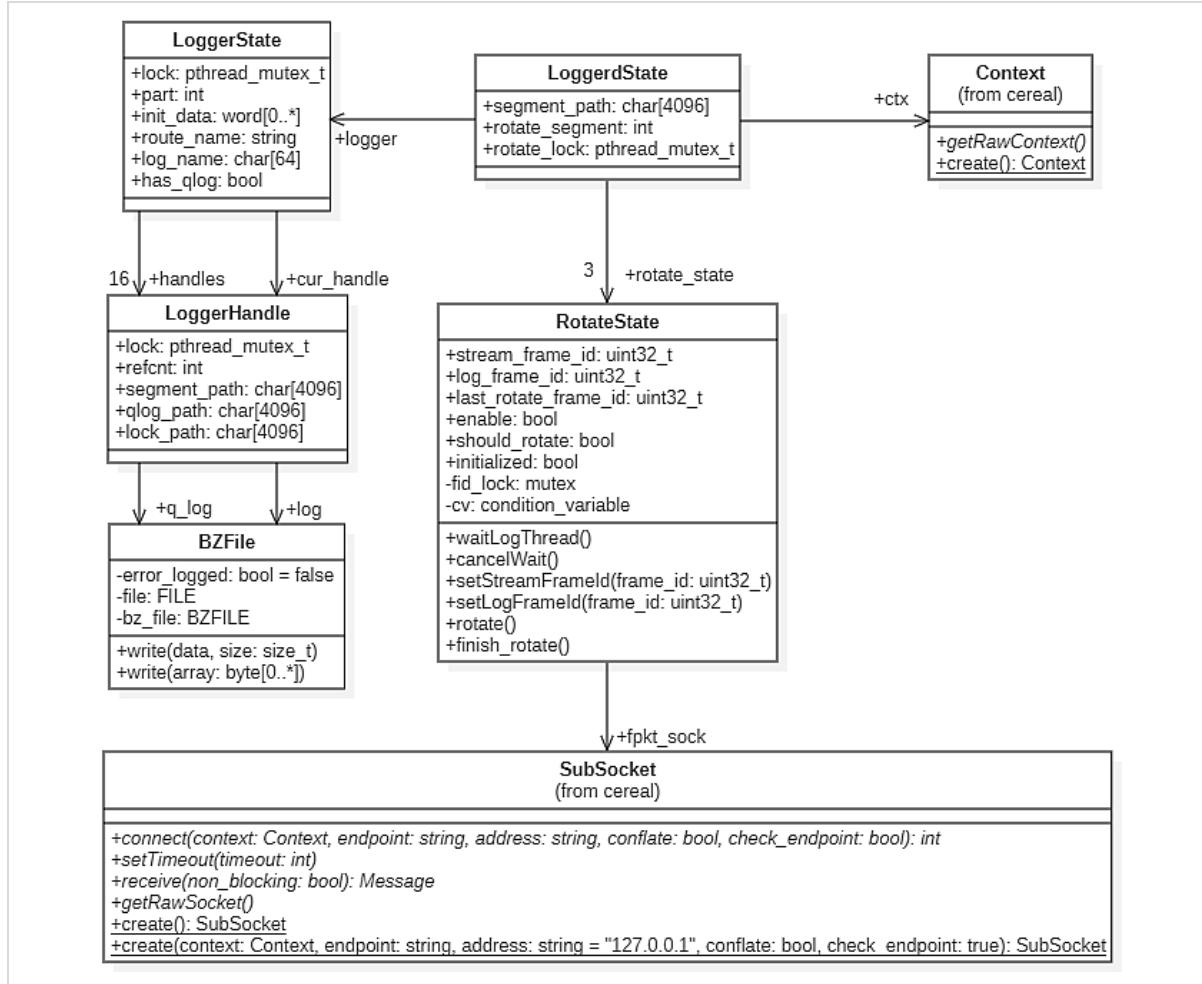


Figure 81 - LoggerdState class diagram

During the initialization phase, a socket is created for each of the services defined included in *cereal/services.h*. Then, if the service is one of *roadCameraState*, *driverCameraState*, or *wideRoadCameraState*, it also adds the socket to the *RotateState* object corresponding to the specific camera.

```

for (int cid=0; cid <= MAX_CAM_IDX; cid++) {
    if (string(it.name) == cameras_logged[cid].frame_packet_name) {
        s.rotate_state[cid].fpkt_sock = sock;
    }
}
  
```

The encoder thread is initialized for each camera connected to the device (road camera and driver camera only in the case of Comma Two and the additional wide road camera in the case of Comma Three). The encoder thread (*encoder_thread()* in *loggerd/loggerd.cc*) loads the camera information and instantiates a VisionIPC client for each of the cameras, receiving the frames on the corresponding socket.

```

LogCameraInfo &cam_info = cameras_logged[cam_idx];
VisionIpcClient v ipc_client = VisionIpcClient("camerad",
                                              cam_info.stream_type,
                                              false);

```

To each camera is assigned a video encoder, which is in charge of encoding the frames received through VisionIPC. The Comma devices encoder can leverage OpenMAX, which is a cross-platform API that provides comprehensive streaming media codec and application portability by enabling accelerated multimedia components to be developed, integrated, and programmed across multiple operating systems and silicon platforms. [52]



Figure 82 - VideoEncoder class diagram

When receiving a message on any of the socket, they are logged and saved in a log file using the *bzlib* library, which provides an interface for compressing and decompressing streams of data represented as lazy ByteString.

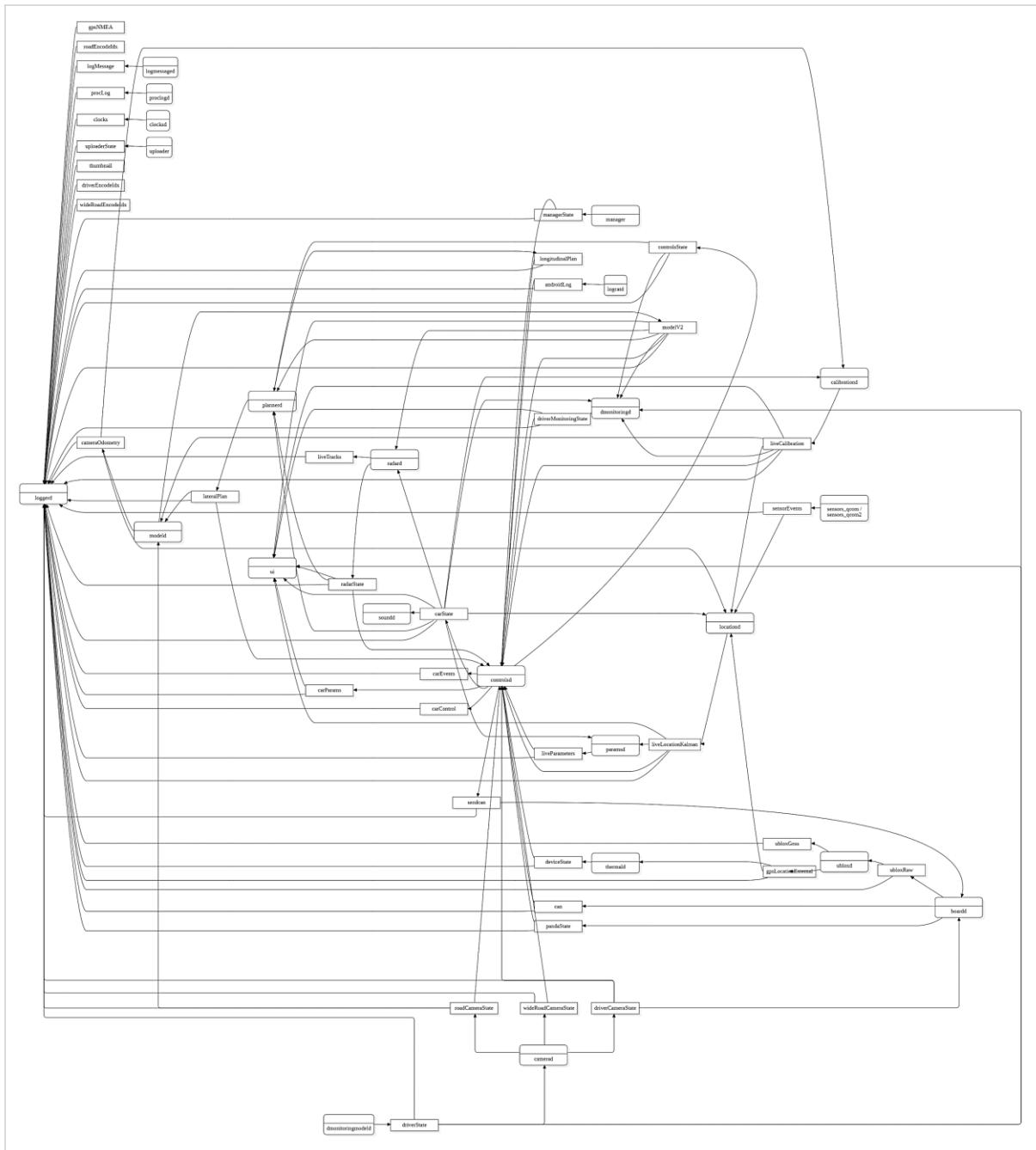


Figure 83 - selfdrive data flow diagram

Loggerd, together with controlsd, represents one of the main nodes where data are exchanged among the different processes. If controlsd receives data from all the sensors, cameras, and actuators of the car, sending back the computed corrections over the CAN bus, the logger daemon only receives the messages from all the sockets available without publishing any response message.

4.3 Testing

Testing in Openpilot is particularly crucial since it has to ensure that all the security constraints and traffic rules constraints are always respected, that the quality constraints of a Level 2 automated drive system are met, and this has to be true for all the cars supported by Openpilot.

GitHub Actions allows to perform automated checks for each change made to the master branch, and these checks range from unit tests to replaying a route for each car, ensuring that rules are respected.

A first GitHub action performs a static analysis of the code, highlighting the programming errors for all the programming errors used by Openpilot.

Check python ast.....	Passed
Check JSON.....	Passed
Check Xml.....	Passed
Check Yaml.....	Passed
Check for merge conflicts.....	Passed
Check for broken symlinks.....	Passed
mypy.....	Passed
flake8.....	Passed
pylint.....	Passed
cppcheck.....	Passed

Another GitHub Action checks the software against memory problems, and this is done through Valgrind, a tool used to debug and profile C and C++ code to automatically detect memory leaks and other problems with the memory management.

TestValgrind
+ extract_leak_size(log) + valgringlauncher(arg, cwd) + replay_process(config, logreader) + test_config()

Table 48 - TestValgrind test case

The check is performed on the process that downloads the GPS data from the server and ensures that the same number of frames is allocated and freed.

```
Memcheck, a memory error detector
Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
Command: ./ubloxd

HEAP SUMMARY:
in use at exit: 0 bytes in 0 blocks
total heap usage: 92,222 allocs, 92,222 frees, 16,753,646 bytes
allocated
All heap blocks were freed -- no leaks are possible
For lists of detected and suppressed errors, rerun with: -s
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

One of the main GitHub Action of the workflow run the unit tests defined in all the directories of selfdrive and other external directories like common, opendbc, and tools.

Different test cases are defined for the main processes of Openpilot. Boardd API tests compare the new version of the API with an old one, ensuring that both receive and send the same messages and also that the performances of the new API are better than that of the old API.

TestBoarddApiMethods
+ test_correctness()
+ test_performance()

Table 49 – TestBoarddApiMethods

Moreover, it is executed a loopback test checking that the analyzed board can properly send and receive messages. The loopback test requires a loopback device, in the specific case a Panda device connected. Controlsd unit tests verify the correct working of the alerts and the other general status parameter, as well as the correctness and safety of lateral and longitudinal maneuvers that are required to actuate the planner's outputs.

TestAlerts
+ test_events_defined()
+ test_alert_text_length()
+ test_alert_sanity_check()
+ test_offroad_alerts()
+ test_offroad_alerts_extra_text()

Table 50 - TestAlerts test case

This test case ensures that the alerts are displayed correctly on the device, with no overlapping or wrong size rendering. In the controls library are also available clustering functionalities, used by the radar process to cluster multiple acquisitions.

TestClustering
+ test_scipy_clustering()
+ test_pdist()
+ test_cpp_clustering()
+ test_cpp_wrapper_clustering()
+ test_random_cluster()

Table 51 - TestClustering test case

To test if controlsd is able to always maintain the set cruise speed, an ad-hoc test verifies for different cruise speed that in all the cases it is reached and maintained with an even acceleration and without crashing in a hypothetical lead car in front of the vehicle.

TestCruiseSpeed
+ test_cruise_speed()

Table 52 - TestCruiseSpeed test case

The test simulates a hundred seconds ride at a cruise speed each time higher than 5 m/s. It is implicitly imposed a limit for the acceleration that can be exerted to respect both the safety constraint and the comfort of the driver, which would fail in case of an excessive acceleration.

Plotting the result of the test in a graph shows us how the acceleration is always constant for all the cruise speed considered and that the acceleration does not exceed the $1,2 \text{ m/s}^2$.

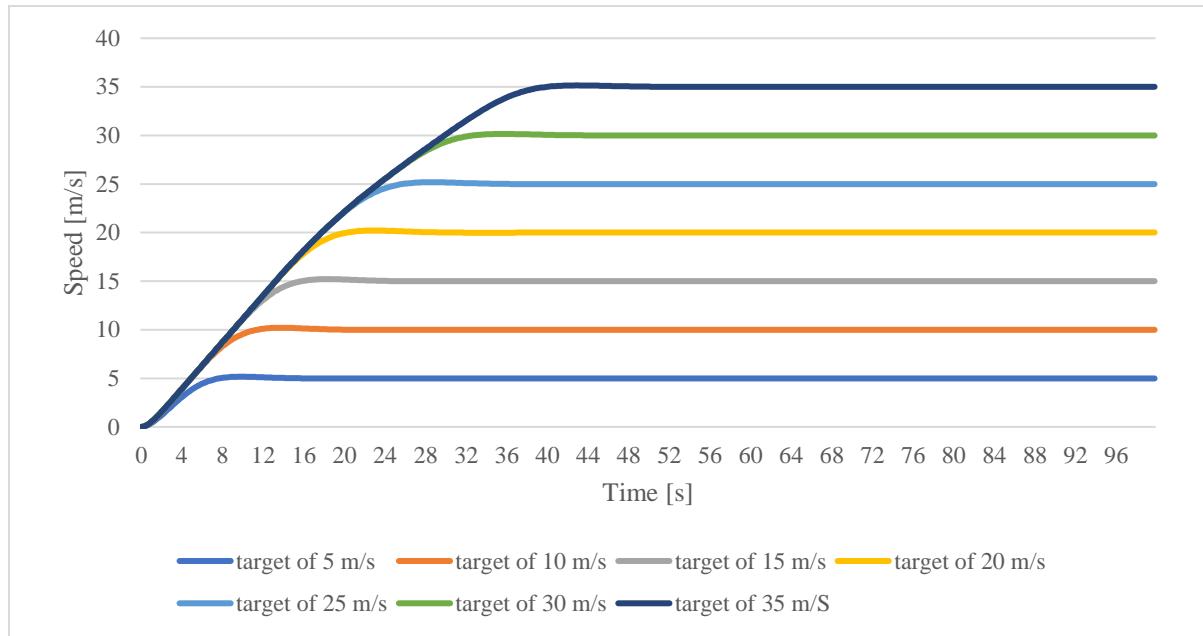


Figure 84 - Speed variation for different targets of cruise speeds

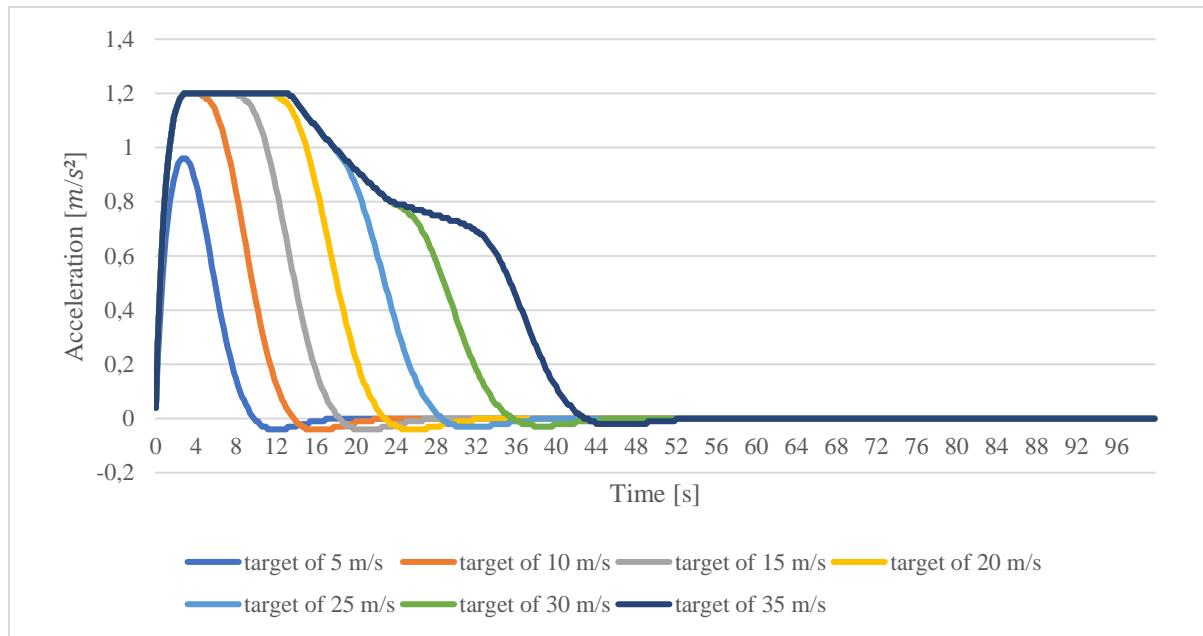


Figure 85 - Acceleration variation for reaching the target cruise speed

Another important test, crucial to ensure safety while driving, verifies an adequate distance when following another car is always kept. This distance has to be dynamic and change according to the speed of the lead car: the higher the speed, the greater the distance.

TestFollowingDistance
+ test_following_distanc()

Table 53 - TestFollowingDistance test case

We can also notice from the graph in Figure 86 that if the lead car is not moving, the vehicle will stop, maintaining a safe distance.

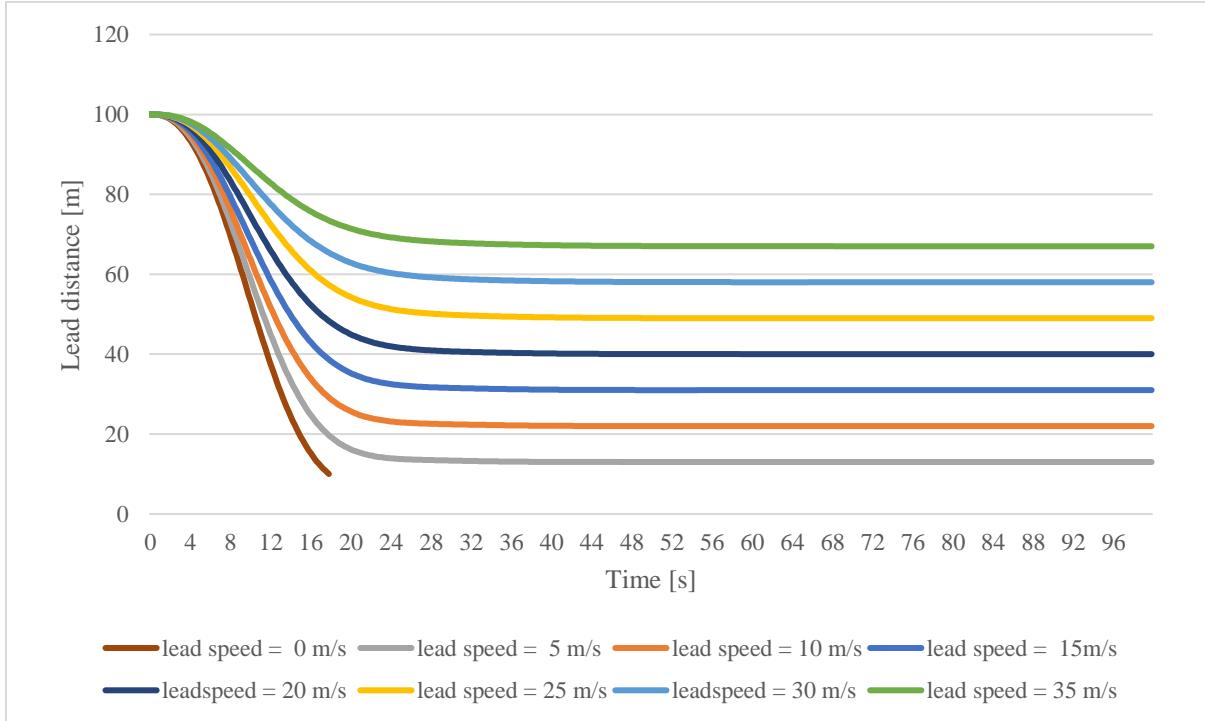


Figure 86 - Following distance at different lead speeds

In controlsd, the lateral planner is the main component that manages the steering of the vehicle. The test verifies that the vehicle is subject to the expected curvature.

TestLateralMpc
- _assert_null(sol, curvature)
- _assert_symmetry(sol, curvature)
+ test_straight()
+ test_y_symmetry()
+ test_poly_symmetry()
+ test_curvature_symmetry()
+ test_psi_symmetry()
+ test_no_overshoot()
+ test_switch_convergence()

Table 54 - TestLateralMpc test case

When the controlsd process is first initialized, it has to check the fingerprint of the car where the device running Openpilot is mounted, since each car has a different response to the applied torque.

TestStartup
+ test_startup_alert(expected_event, car_model, toggle_enabled, fw_versions)

Table 55 - TestStartup test case

The test case leverages the functionalities of parametrized testing offered by the *unittest* framework to run the test for different cars and under different conditions. The considered cases are a car officially supported by Openpilot (Toyota Corolla), one that supports only the dashcam functionalities (Mazda CX5), and one unrecognized car. For each case, a message is sent to the Panda device and a response message is expected.

The process dmonitoringd verifies other safety constraints relative to the behavior of the driver, who should be always focused on the drive. The test case verifies that the driver is always framed and attentive, and then simulates different situations in which the driver interacts with the steering wheel or goes out of the camera frame end ensures that Openpilot engages and disengages properly.

TestMonitoring
<ul style="list-style-type: none"> - _run_seq(msgs, interaction, engaged) - _assert_no_events(events) + test_fully_aware_driver() + test_fully_distracted_driver() + test_fully_invisible_driver() + test_normal_driver() + test_biggest_comma_fan() + test_sometimes_transparent_commuter() + test_last_second_responder() + test_pure_dashcam_user() + test_long_traffic_light_victim() + test_somewhat_indecisive_model()

Table 56 - TestMonitoring test case

In the case of a normal driver (*test_normal_driver()*), for instance, the simulated simulation sees the Comma device operating normally, then it displays a warning to the driver, who begin to pay attention. After that, another warning is displayed, the driver touches the wheel, and the warning stops.

The logger daemon is another important component that needs to be tested since all the messages and the camera acquisition pass through it and the logs help to diagnose potential problems and allow the users to replay their rides.

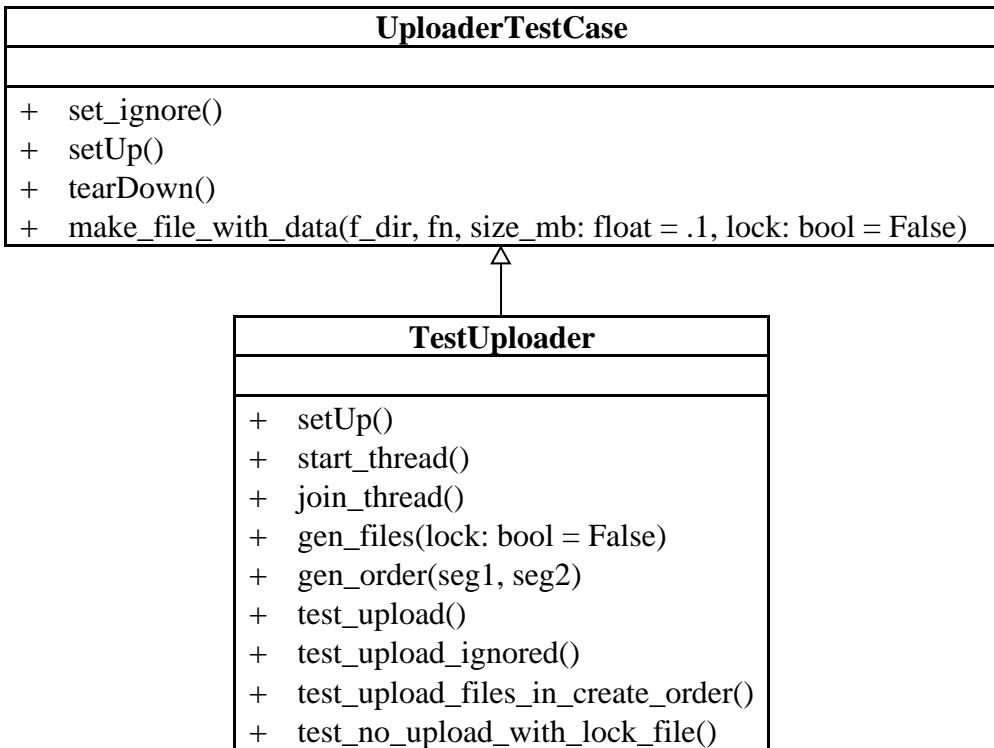


Table 57 - TestUploader test case

The test case relative to the upload process tries to upload a series of segments to a fake URL and waits for a response message with the outcome of the upload.

```

upload ('2019-04-18--12-52-54--153/qlog.bz2',
        '/tmp/tmp9bj6482/2019-04-18--12-52-54--153/qlog.bz2') over 1
{
    "event": "upload",
    "key": "2019-04-18--12-52-54--153/qlog.bz2",
    "fn": "/tmp/tmp9bj6482/2019-04-18--12-52-54--153/qlog.bz2",
    "sz": 1048576
}
checking '2019-04-18--12-52-54--153/qlog.bz2' with size 1048576
uploading '/tmp/tmp9bj6482/2019-04-18--12-52-54--153/qlog.bz2'
upload_url v1.3 http://localhost/does/not/exist {}
** WARNING, THIS IS A FAKE UPLOAD TO http://localhost/does/not/exist **
{
    "event":
    "upload_success",
    "key": "2019-04-18--12-52-54--153/qlog.bz2",
    "fn": "/tmp/tmp9bj6482/2019-04-18--12-52-54--153/qlog.bz2",
    "sz": 1048576,
    "debug": true
}

upload done, success=True

```

Similarly, the deleter process deletes two sample files and verifies that after the operation they are not present anymore.

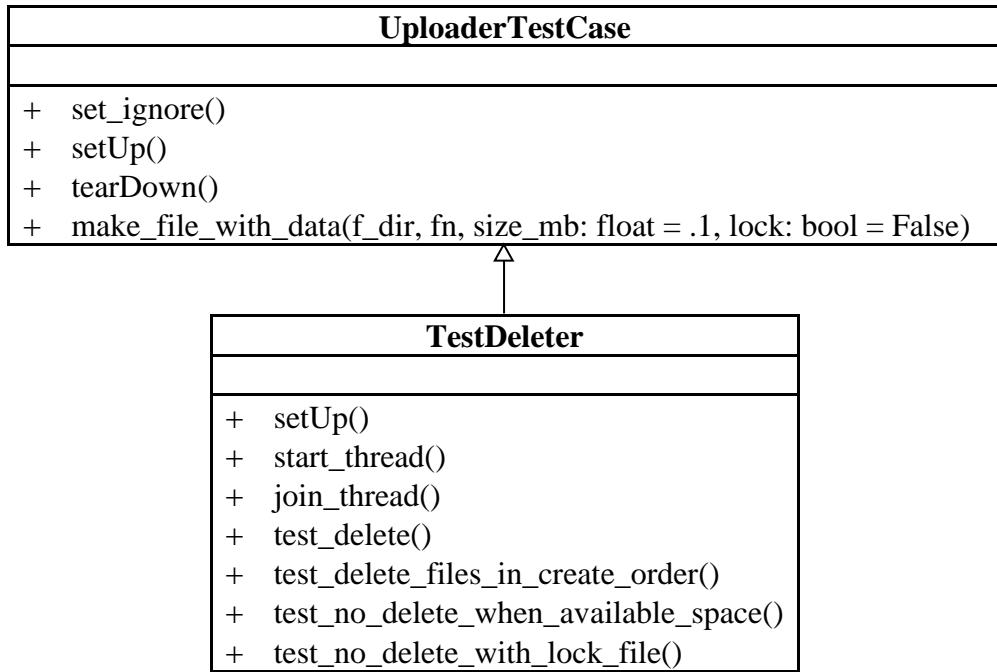


Table 58 - TestDeleter test case

```

deleting /tmp/tmpwhp7mz2c/2019-04-18--12-52-54--82
deleting /tmp/tmpf0tgti3t/2019-04-18--12-52-54--268
  
```

The encoder test case uses the Encoder class to encode the camera frame and verifies the correctness of the encoding process.

The checks ensure that when encoding, the number of frames received and elaborated is the same, that there are no duplicated frames, and that no frame is skipped.

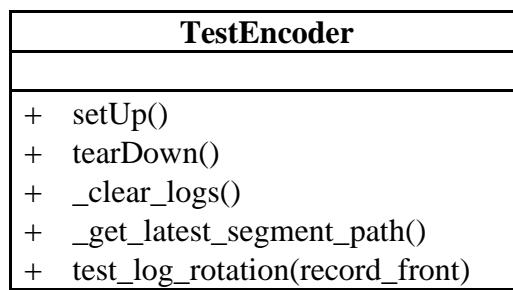


Table 59 - TestEncoder test case

The actual loggerd test case verifies the status of the log after performing different operations, like the initialization or the upload of a file. In particular, the test is executed after the test case relative to the uploader, which uploads known files.

The logger verifies that the file logged are the same.

TestLoggerd
- <code>_get_latest_log_dir()</code>
- <code>_get_log_dir(x)</code>
- <code>_get_log_fn(x)</code>
- <code>_gen_bootlog()</code>
- <code>_check_init_data(msgs)</code>
- <code>_check_sentinel(msgs, route)</code>
+ <code>test_init_data_values()</code>
+ <code>test_rotation()</code>
+ <code>test_bootlog()</code>
+ <code>test_qlog()</code>
+ <code>test_rlog()</code>

Table 60 - TestLoggerd test case

For what concerns the processes managing the calibration using the EKF, the test case relative to calibrationd test that the calibration parameters are correctly uploaded.

TestCalibrationd
+ <code>test_read_saved_params</code>

Table 61 - TestCalibrationd test case

Locationd has two test cases, one for the library and one for the process itself. The one testing the library functionalities ensures that the Localizer library is able to interpret all the messages correctly.

TestLocationdLib
+ <code>ffi: FFI</code> + <code>lib: FFILibrary</code> + <code>localizer: Localizer</code> + <code>buff_size: int = 2048</code> + <code>msg_buf</code> + <code>setUp()</code> + <code>localizer_handle_msg(msg_builder)</code> + <code>localizer_get_msg(t = 0, inputsOK: bool, sensorsOK: bool, gpsOK: bool)</code> + <code>test_liblocalizer()</code> + <code>test_device_fell()</code> + <code>test_posenet_spike()</code>

Table 62 - TestLocationdLib test case

The test case relative to the process tries to handle different messages, including messages with detail on the GPS positioning and fake camera odometry measurements.

TestLocationdProc	
+ MAX_WAITS: int	= 1000
+ pm:	PubMaster
+ setUp(self)	
+ tearDown(self)	
+ send_msg(self, msg)	
+ test_params_gps(self)	

Table 63 - TestLocationdProc test case

Athena test cases aim to test the functionalities provided by the Athena APIs, which allow to access remotely a Comma device and to read the essential information on the car ad device state.

TestAthenadMethods	
+ setUp()	
+ wait_for_upload()	
+ test_echo()	
+ test_getMessage()	
+ test_listDataDirectory()	
+ test_do_upload(host)	
+ test_uploadFileToUrl(host: string)	
+ test_upload_handler(host)	
+ test_upload_handler_timeout()	
+ test_cancelUpload()	
+ test_listUploadQueueEmpty()	
+ test_listUploadQueueCurrent(host: string)	
+ test_listUploadQueue()	
+ test_startLocalProxy(mock_create_connection)	
+ test_getSshAuthorizedKeys()	
+ test_getVersion()	
+ test_jsonrpc_handler()	
+ test_get_logs_to_send_sorted()	

Table 64 - TestAthenadMethods test case

The test case includes tests for the upload functionalities, both in the case of a single upload of a file and upload of multiple files and ensures that the queue behaves as expected. It also uses the APIs to query a parameter and verifies that the response is the correct one and verifies that the JSON-RPC messages are correctly interpreted.

Another test case tests specifically the registration of the device through Athena, ensuring that the device ID is correctly registered and that caching of the relevant information always ensures the availability of those.

TestRegistration
<ul style="list-style-type: none"> - _generate_keys() + setUp() + tearDown() + test_valid_cache() + test_missing_cache() + test_unregistered()

Table 65 - TestRegistration test case

Monitoring the performances and the temperatures of the devices is also a critical part of the testing process, since it has to be ensured that all the devices operate correctly under the possible conditions.

TestPowerMonitoring
<ul style="list-style-type: none"> + setUp() + mock_peripheralState(hw_type: int, car_voltage: int = 12) + test_pandaState_present() + test_offroad_ignition(hw_type: int) + test_offroad_integration_discharging(hw_type: int) + test_car_battery_integration_onroad(hw_type) + test_car_battery_integration_upper_limit(hw_type: int) + test_car_battery_integration_offroad(hw_type: int) + test_car_battery_integration_lower_limit(hw_type: int) + test_max_time_offroad(hw_type: int) + test_car_voltage(hw_type: int) + test_disable_power_down() + test_ignition() + test_harness_connection()

Table 66 - TestPowerMonitoring test case

The thermal process monitors the power consumption detected by the Panda device and verifies that is always under the predefined thresholds and that is null when the device is not being used. The tests are repeated for all the Panda versions available and for different conditions to test the detection when the vehicle is moving or is turned off.

A crucial part of the testing is represented by the *car unit tests*, which are unit tests defined for all the cars supported by Openpilot. These parametrized unit tests are executed for all the supported cars.

TestCarModel	
+ car_model: string	
+ can_msg: Message[0..*]	
+ CP: CarParams	
+ CI: CarInterface	
+ CC: CarControl	
+ setUpClass()	
+ test_car_params()	
+ test_car_interface()	
+ test_radar_interface()	
+ test_panda_safety_rx_valid()	
+ test_panda_safety_carstate()	

Table 67 - TestCarModel test case

For each car model, the test case tests that the parameters are detected correctly, that the safety model for the car is available in Panda, and that the car uses the correct version of the lateral tuning algorithm (pid, lqr, or indi). Here are reported some of the key parameters detected during the execution of *test_car_params()* test.

```
Analyzing HYUNDAI VELOSTER 2019
Car mass: 1613.88037109375
Car steerRateCost: 0.5
Lateral tuning algorithm: pid

Analyzing CHRYSLER PACIFICA HYBRID 2017
Car mass: 2378.0
Car steerRateCost: 0.699999988079071
Lateral tuning algorithm: pid
```

To test the car interface, it is counted the number of invalid messages sent using the CarInterface component. a Test passes if the number of invalid messages is less than fifty.

Similarly, the radar interface is tested by checking that the number of invalid messages stays under a certain threshold, twenty in this case.

More safety checks are made to ensure that all the messages are received correctly on the RX pin of the Panda board and that no failure happens when checking the parameters indicating that the gas pedal and the brake pedal have been pressed, and when checking the engagement state of Openpilot, ensuring that no message allowing it to take control of the car is lost.

Another important set of tests is represented by the *process replay* tests, run in their dedicated workflow in GitHub Actions. A process replay test is a regression test designed to identify any changes in the output of a process. This test replays a segment of drive and then stores a log of each process's output as a reference. Then it compares the output of the same set of processes calculated after the committed changes and it compares the output to the reference replay. If there are differences, the test fails and displays the changes.

In Figure 87 is shown the steer ratio computed in two different commits.

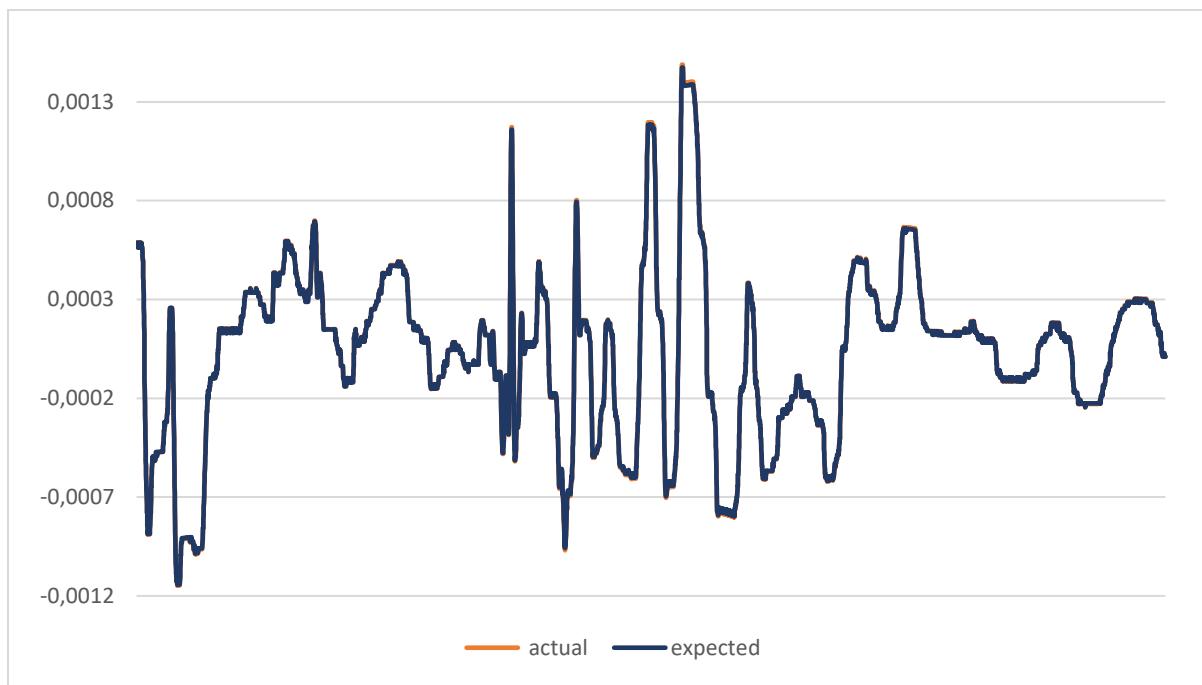


Figure 87 - Steer ratios of two consecutive commits

The values almost overlap, but small differences can be noticed throughout the log. These differences are more clear looking at the graph in Figure 88. Here the values plotted are computed as the difference between the expected value of the steer ratio and the actual one. Even a small variation for the steer ratio variable could lead to major issues while driving, therefore the test fails even for the smallest differences from the expected values of all the processes.

This type of validation is run for all the car models supported by Openpilot.

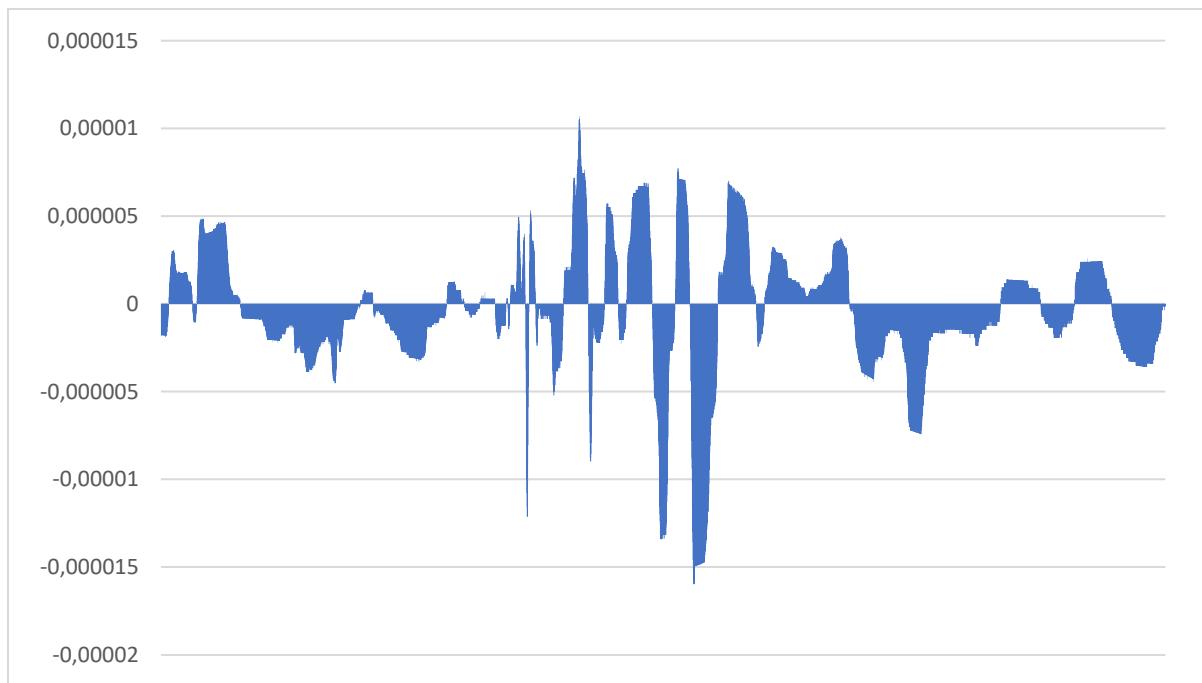


Figure 88 – Difference of steer ratio in two consecutive commits

An important part of testing is represented by the hardware-in-the-loop (HIL) tests, executed on a Jenkins server where a series of Comma Twos and Comma Threes continuously run real segments. Based on this execution can be determined if the performances are the one expected if the CPU usage and the temperature is optimal for all the devices. The devices connected to the Jenkins server run a series of tests, including process and model replay tests, unit tests and car unit tests, camera tests, and hardware tests. The devices tested are a total of twelve and keep them running for a long period and checking their behavior allows ensuring and high long-term reliability.

The execution on the Jenkins server can give more insights into what the performances of the different processes are and allows to plot the timing of the messages sent by them. In a recent convention held by Comma, the COMMA_CON, have been shown the timing of different processes which are crucial to Openpilot, obtained by analyzing the devices connected to the server. The results show that overall, the delay between a message and another follows a Gaussian distribution around a mean that should never be too high.

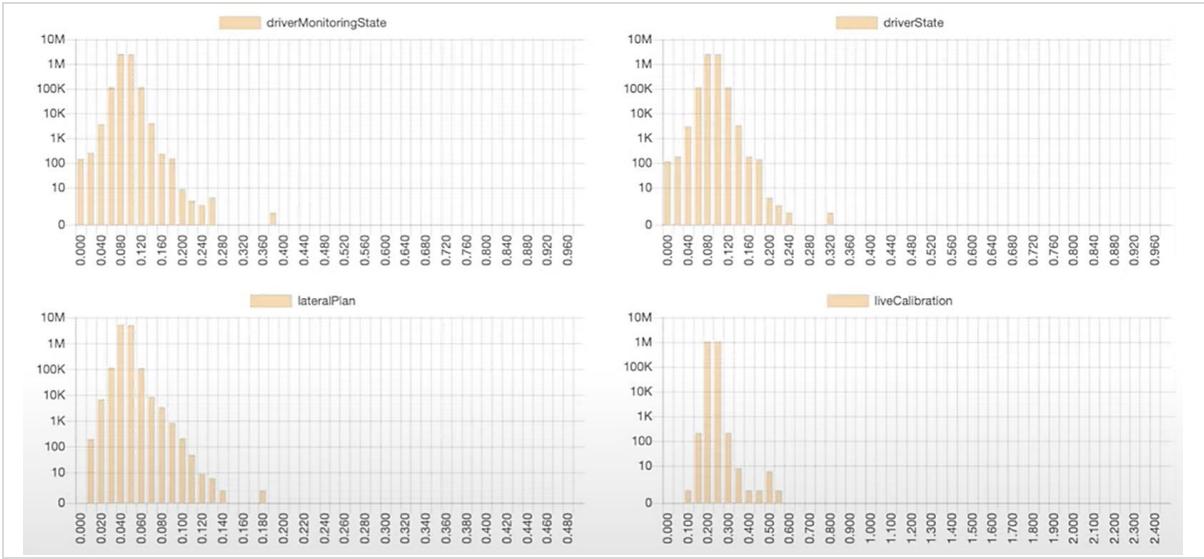


Figure 89 - Messages delays obtained through HIL testing, shown during COMMA_CON

One process that is crucial to Openpilot is boardd: since it allows the car to communicate with Openpilot, the timing of messages coming from all the other processes depends on it. The delays of the messages traveling through the *can* socket have to stay as low as possible, and this was the reason why a past release of the NEOS firmware was canceled: it suffered from a problem that would cause boardd to lag and have a latency of order of magnitude higher.

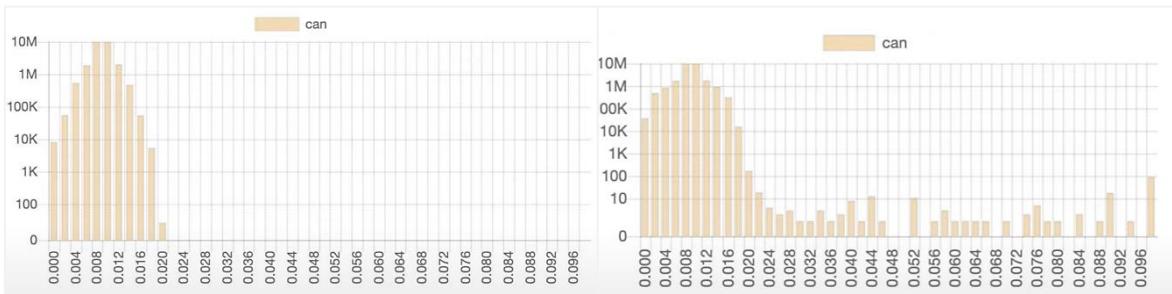


Figure 90 - Comparison between a healthy boardd and a lagging one, from COMMA_CON

Many tests were conducted also in real-world situations by different testers, both individual and organizations. A recent study by Consumer Report [53] shows how Openpilot excels in many of the tested categories and can outperform its competitors [*Figure 91*]. Their tests were conducted in 5 main categories:

- **Capability and performance:** it considers how well each system kept the vehicle in the center of the lane, as well as how smoothly and intuitively the ACC system could adjust its speed behind other cars.
- **Keeping the driver engaged:** the ability of the active driving assistance systems to monitor the driver and encourage the driver to stay actively engaged.
- **Ease of use:** how easy it was for drivers to engage the systems and adjust the settings. It was also considered the types and amount of information displayed to drivers, and how easy it was for drivers to know and understand what the system was doing.
- **Clear when safe to use:** how the systems communicate in real-time about when drivers should and should not be using the technology.
- **Unresponsive driver:** evaluation of the systems for their escalation process for warnings, steering control, and speed control.

Comma Two managed to obtain excellent results in all the categories, and presumably, Comma Three will be even better. However, is the only system that requires an after-market modification from the customers.

Even if such modifications should not be endorsed, the results of the tests show that are comparable, and in many cases superior, to many other OEMs, as shown in the table reporting the overall rating result of the Consumer Reports' study.

SYSTEM NAME	SCORE	CAPAB. & PERF.	KEEPING DRIVER ENGAGED	EASE OF USE	CLEAR WHEN SAFE TO USE	UNRE- SPONSIVE DRIVER
Comma Two Open Pilot	78	8	9	8	6	8
Cadillac Super Cruise	69	8	7	3	8	9
Tesla Autopilot	57	9	3	7	2	6
Ford/Lincoln Co-Pilot 360	52	8	4	3	4	5
Audi Driver Assistance Plus	48	8	3	3	2	6
Mercedes-Benz Driver Assistance	46	6	4	4	2	5
Subaru Eyesight	46	7	4	3	4	5
Hyundai Smart Sense, Kia Drive Wise	46	5	4	5	4	4
BMW Active Driving Assistance Pro	44	7	3	3	2	6
Porsche Active Safe	41	4	3	6	2	5
Volvo Pilot Assist	41	6	3	3	2	5
Toyota/Lexus Safety Sense 2.0	40	5	4	2	4	5

Figure 91 - Results of the study conducted by Consumer Report

4.4 Development and community contributions

The concept of making a normal car a self-driving one was made reality when George Hotz, for the first time, hacked its own Toyota Supra and made it drive on the highway autonomously. From the realization of the potential of machine learning and the high computational power offered by modern technology George founded Comma.ai. The first natural step to create a *superhuman driver* was to gather enough data to train a *superhuman self-driving model*. Comma.ai launched in 2016 the mobile application *chffr*: the objective was to give people a mobile dashcam for the car, while at the same time collecting data on how people drive daily. The first device released was the Comma One, but soon the production was interrupted after that NHTSA (the National Highway Traffic Safety Administration) questioned the compliancy of the device with regulations.



Figure 92 - The Comma One

The first version of Openpilot was released on the 29th of November 2016.

Openpilot release (29/11/2016) <Vehicle Researcher>

The functionalities offered were very limited and included adaptive cruise control and lane keep assist. There was not a real driving model making the predictions on where to drive the car and the supported cars were two, an Acura and a Civic. The device as well was not more than a prototype, all 3D-printed and based on a smartphone hardware.

The next releases of version 0.2.x focused on bug fixing and small optimizations. Most of the optimization were made to boardd process and the way it handled the messages traveling on the CAN bus and to the vision model and image acquisition, improving from time to time the quality of the acquired images interpreted by the model. The process managing elaborating the images was *visiond*, then substituted by *camerad*. Visiond not only managed the frame acquisition, but the whole vision pipeline of Openpilot.

With version 0.3.x, the camera acquisition system was refactored, and it was introduced VisionIPC to allow the connection of multiple clients to the cameras. In release 0.3.0 was also introduced the CarParams abstraction, which simplified the way to access the key parameters relative to the car status.

Version 0.3.7 introduced the Model Predictive Control, which improved the lateral and longitudinal controls by approaching the problem of predicting the direction that the car had to follow as an optimization problem, that given the position and velocity of the lead car tries to optimize the breaking and the steering of the vehicle.

Openpilot v0.3.7 release (01/10/2017) <Vehicle Researcher>
Openpilot v0.3.7 tweaks (03/10/2017) <Vehicle Researcher>
Openpilot v0.3.7 tweaks (04/10/2017) <Vehicle Researcher>

On 19th October 2017 was announced the EON Dashcam DevKit. The EON [Figure 93] is a dashcam that can connect to the car's communication system and record the car's data in sync with driving videos. It also includes a camera-based Driver Monitoring (DM) system that alerts the driver when distracted or asleep. EON is essentially a LeEco Le Pro 3 smartphone in a plastic case loaded with a custom version of Android maintained by the team of Comma.ai. It comes with a Snapdragon 820 processor, a front facing camera and a main camera offering good quality pictures, and a series of sensors that made it an optimal choice to make it the device where to run Openpilot.

It can communicate with the car through *Panda* and *Giraffe* (then substituted by much more modular and simpler car harness), which is a universal common interface allowing to send and receive signals over Wi-Fi or USB directly to/from BUSes available in a vehicle.



Figure 93 - EON dashcam DevKit

The EON was preloaded with a dashcam software, but it was easy to install Openpilot on and in this way it was possible to sell the device while being compliant with all the regulations.

Forward Collision Warning was introduced in version 0.3.9, as a result of all the improvements made on the longitudinal control model. Even without a machine learning model in place at the time of the release, the effort made in tweaking the algorithm to understand what the car in front of the vehicle was doing only relying on camera and sensors and using all the data collected to correct the algorithm based on the false positives detected, allowed to deliver the feature with a high level of reliability.

Openpilot v0.3.9 release (22/11/2017) <Vehicle Researcher>
Openpilot v0.3.9 tweaks (06/12/2017) <Vehicle Researcher>

Version 0.4.4 prepared Openpilot to the introduction of the driver monitoring model. This required to flip the original orientation of the device, which was thought with the camera on the right side, while it was proved that a higher accuracy could be achieved with the driver camera on the left side.

Fortunately, the case which hosted the NEON was symmetric and it was simple to reverse the orientation of the device.

Openpilot v0.4.4 release (14/04/2018) <Vehicle Researcher>

After a deadly accident, where a pedestrian was killed by a self-driving Uber car, Comma.ai decided to include drive monitoring in the future releases of Openpilot.

Driver monitoring was shipped in version 0.5.0. This feature, still in beta version at this stage of the development, was crucial to ensure that the driver was always paying attention while driving, but its performances were not optimal due to the restrictions dictated by the hardware of the device and of the optimization of the software.

Openpilot v0.5 release (13/07/2018) <Vehicle Researcher>

One important step of the development, especially for the community, was introduced in version 0.5.8 with the open sourcing of the *visiond* model. The open-source version of the model was based on the SNMP Snapdragon library, which offered the same functionalities, tweaked for Snapdragon processors, offered by the neural network that was previously included and in the model and executed through OpenCL. With the open sourcing of *visiond*, all the code of Openpilot was made opensource.

Openpilot v0.5.8 release (24/01/2019) <Vehicle Researcher>

With the series of version 0.6.0 it was introduced a model developed using machine learning and using as a ground truth the data coming directly from the Comma devices, so that the model was trained on the actual behavior of human while driving their car.

Openpilot v0.6 release (28/06/2019) <Vehicle Researcher>

Comma.ai continued to improve its hardware and the software. In April 2019 released The Comma the *Comma Connect app* (on iOS and Android), which provided video storage of drives and remote camera access, and in January 2020 it launched the Comma Two [Figure 94]. The device replaces the EON DevKit, but it retains all the EON's features. The Comma Two is powered via OBD-C, rather than a battery, as Hotz says that was a top complaint from the EON owners. It had a larger mount, as well as a custom fan-based hardware cooling solution. Like the EON, which used a camera to recognize drivers' faces and decelerate if it detected that they were distracted, the *Comma Two* performs facial recognition. It leverages two infrared sensors as opposed to an RGB sensor, enabling it to work during nighttime.



Figure 94 - The Comma Two device

Version 0.7.1 was the first to support the Comma Two. In the same release was also shipped the new *Supercombo* model, which merged posenet and driving model into a unique model, allowing a better lead estimation.

Openpilot v0.7.1 release(15/01/2020) <Vehicle Researcher>

In January 2020 Openpilot also switched from a cathedral style open sourcing to a bazaar style, sharing on the GitHub repository all the changes made and allowing all the contributors to see in real time the changes that were being made at each step of the development and not only when a new version was released.

root commit (17/01/2020) <George Hotz>

Version 0.7.7 introduced the live localizer, a precise vehicle abstractions providing all the key acquisitions of the car sensors.

get ready for live localizer <Harald Schafer> (11/02/2020)

WIP: Live localizer (#1074) <Willem Melching> (11/02/2020)

more fixes (11/02/2020) <Harald Schafer>

The vehicle characteristics indicated by the manufacturer are not always respected and they can vary even among different cars of the same model. Through the live localizer is possible to estimate those parameters, that include the wheels grip, the steering ratios, the offset in degrees of the steering wheel, while driving the car. This allows to make much better predictions and to control the car more precisely.

Openpilot v0.7.7 release (17/07/2020) <Vehicle Researcher>

Version 0.8.0 improved even more the driving model, including prediction in 3 dimensions to consider also slopes on the road that with the past models caused problems in interpreting the camera frames.

Openpilot v0.8.0 release (24/11/2020) <Vehicle Researcher>

In version 0.8.3 was delivered a model fully trained end-to-end, capable of making high quality lateral planning.

New KL model + laneless toggle (#20454) (24/03/2021) <Harald Schafer>

The model is trained using a simulator that takes real world video and computes how Openpilot would behave with the model. To do so, the acquired frames are warped to simulate the car movement and the path.

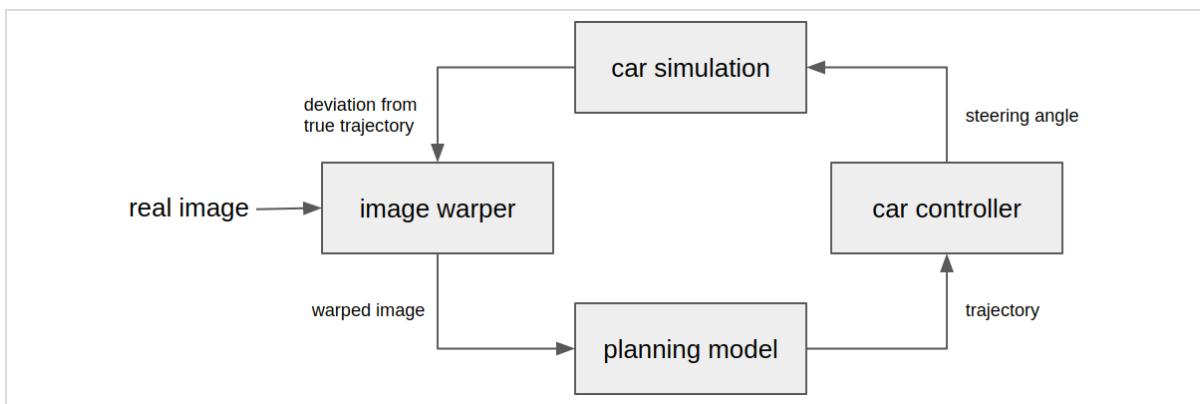


Figure 95 - Diagram of the warp-simulator

To train the model to recover from mistakes and go back on the optimal path (the path that a human would take), in the simulation were introduced on purpose noise and deviation that would make the car deviate from the optimal path and train the model to go back on the ideal trajectory. This method, even if it works in a simulation, resulted to be less effective in real-world situations, since training the model using warped images introduced a lot of artifacts. For a model is much simple to learn if an image is warped or not warped, and some prediction could be based not on the movement that the car had to make to go back on the optimal trajectory, instead on the changes to be made to make the image not look warped. To avoid this, the Comma.ai research team made a great effort in trying to make the model “blind” to the simulation artifacts. This was possible by applying the Kullback-Leibler divergence loss to the vector containing the features of the frame, filtering at training time the relevant features from all the others.

Version 0.8.7 was the first to support the new piece of hardware developed by Comma.ai, the Comma Three[*Figure 96*]. This last iteration of the device. It has dual camera with 360° vision, plus a narrow camera to spot objects far off in the distance. The three 1080p cameras have 120 dB of dynamic range, two generations ahead of a leading electric car maker. It can connect through Wi-Fi, LTE, and offers a high-precision GPS.



Figure 96 - The Comma Three device

A lot of support was also given by the community, especially when it comes to new supported cars. The number of supported cars went up from two in the first version of Openpilot to more than a hundred and forty cars, with even more cars not officially supported but still maintained by the community itself.

One of the biggest contributor is without a doubt **Dean Lee**, which made more than five hundred commits and his contributions go from simple bug fixing and small optimization to the refactor of core parts of the code. In particular, on many occasions he optimized the C++ code of different components of the project and converted Python code in C++ code, helping to increase the performances of the software.

Refactor ModelFrame struct to class (#20005) (19/06/2021) <Dean Lee>
UI: refactor onboarding (#21223) (19/06/2021) <Dean Lee>
util.cc: refactor read_file (#21321) (18/06/2021) <Dean Lee>
UI: refactor ButtonControl (#21315) (18/06/2021) <Dean Lee>
util.cc: refactor read_file (#21295) (17/06/2021) <Dean Lee>

qt/api.cc: refactor create_jwt (#21281) (16/06/2021) <Dean Lee>
Refactor C++ LogReader (#21152) (13/06/2021) <Dean Lee>
UI: refactor SoftwarePanel (#21244) (13/06/2021) <Dean Lee>
refactor FrameReader (#21141) (08/06/2021) <Dean Lee>
Refactor FrameReader (#21002) (25/05/2021) <Dean Lee>
camerad: refactor RGBToYUVState into a class (#20310) (13/04/2021) <Dean Lee>
refactor webcam, use common run_camera function (#20555) (06/04/2021) <Dean Lee>
refactor set_driver_exposure_target (#20327) (03/04/2021) <Dean Lee>
refactor draw_circle_image (#20473) (30/03/2021) <Dean Lee>
refactor function alloc (#20192) (12/03/2021) <Dean Lee>
boardd: refactor usb_connect, delete panda on failure (#19956) (29/01/2021) <Dean Lee>
Panda: refactor get_firmware_version, return std::optional<std::vector> (#19896) (28/01/2021) <Dean Lee>
panda: refactor get_serial, return std::optional<std::string> (#19895) (28/01/2021) <Dean Lee>
UI: refactor transform (#19658) (15/01/2021) <Dean Lee>
Refactor image texture stuff into class (#19719) (11/01/2021) <Dean Lee>
Refactor alert blinking (#19583) (08/01/2021) <Dean Lee>
refactor ui_draw_driver_view (#19597) (08/01/2021) <Dean Lee>
refactor ui_draw_image (#19656) (05/01/2021) <Dean Lee>
refactor qlog_counter (#19626) (04/01/2021) <Dean Lee>
refactor imgproc/utils (#2766) (15/12/2020) <Dean Lee>
ui: refactor model related functions (#2026) (18/08/2020) <Dean Lee>

Another active contributor is **Shane Smiskol**, who was also taken as an intern after all of his contributions. His contributions helped to develop the car interface abstraction, but also helped many community members on the Discord server with more than six thousand messages sent.

Update 17 Corolla safetyParam (#2175) (16/09/2020) <Shane Smiskol>
Fix toyota_eps_factor.py script (#2647) (29/11/2020) <Shane Smiskol>
Toyota: always learn offset to accurate steer angle sensor (#20087) (16/02/2021) <Shane Smiskol>
Toyota: simplify angle offsetting (#20102) (20/02/2021) <Shane Smiskol>
Chrysler: Default fingerprint argument to empty fingerprint (#20146) (24/02/2021) <Shane Smiskol>
Honda - Don't send cancel cmd when using comma pedal (#20922) (18/05/2021) <ShaneSmiskol>
Add missing Hondas and Toyotas to tests (#21044) (27/05/2021) <Shane Smiskol>
Update Honda Fit route (#21065) (29/05/2021) <Shane Smiskol>
Add some missing Hyundai routes (#21072) (29/05/2021) <Shane Smiskol>
Add missing Chrysler routes (#21074) (29/05/2021) <Shane Smiskol>
Split Avalon 2016-18 and Avalon 2019+ (#21058) (01/06/2021) <Shane Smiskol>
Merge Accord trims (#21105) (03/06/2021 22:35) <Shane Smiskol>
Use hyundaiLegacy safety model for Hyundai Elantra (#21108) (03/06/2021) <Shane Smiskol>
Toyota: Only use gas interceptor under 19 mph (#21101) (04/06/2021) <Shane Smiskol>

2020 Ioniq PHEV Support (#21147) (11/06/2021) <Shane Smiskol>

Toyota: handle models with permanent low speed lockout (#21512) (10/07/2021) <Shane Smiskol>

Erich Moraga is another important member of the community. His contributions include the addition of more than sixty car fingerprints, but he is also a very active member of the discord server and helped many other community members throughout the years.

Below are reported only some of his contributions, in particular the commits relative to the additions made to the fingerprints of Lexus cars, but many other contributions are relative also to models of other car manufacturer.

Add CAR.LEXUS_RX various missing firmware (#2189) (16/09/2020) <Erich Moraga>

Add engine f/w for CAR.LEXUS_RX (#2235) (28/09/2020) <Erich Moraga>

Added LEXUS_RX_TSS2 ESP & Engine f/w (#19618) (29/12/2020) <Erich Moraga>

2018 Lexus NX300: Add missing EPS & engine f/w (#20337) (14/03/2021) <Erich Moraga>

Add missing engine & EPS f/w for LEXUS_IS (#20398) (18/03/2021) <Erich Moraga>

Add missing CAR.LEXUS_NX f/w (#20546) (01/04/2021 11:05) <Erich Moraga>

Add multiple missing f/w for LEXUS_ESH (#20669) (13/04/2021) <Erich Moraga>

Add several missing LEXUS_ES_TSS2 f/w (#20865) (10/05/2021) <Erich Moraga>

Add missing LEXUS_NXH EPS f/w (#20941) (18/05/2021) <Erich Moraga>

Add missing LEXUS_NX_TSS2 engine & fwdCamera (#21490) (06/07/2021) <Erich Moraga>

Jason Young also helped to support many car models, especially Volkswagen cars. Him alone added the support to almost twenty cars, reverse engineering the messages traveling on the CAN bus of the cars and fingerprinting the different models.

Fixes and new message for VW MQB, fix for Accord Touring (#193) (17/10/2019) <Jason Young>

Add TSK_06 CRC validation for VW MQB (#234) (16/03/2020) <Jason Young>

Additional car params auto-detection in support of VW (#38) (31/03/2020) <Jason Young>

Add SWA_01 message detail and CRC support for VW MQB (#236) (02/04/2020) <Jason Young>

VW MQB: UDS fingerprinting support (#20271) (26/03/2021) <Jason Young>

VW MQB: Volkswagen Tiguan Mk2 (#20484) (26/03/2021) <Jason Young>

VW MQB: VW Jetta Mk7, Škoda Kodiaq Mk1 (#20487) (26/03/2021) <Jason Young>

VW MQB: Volkswagen Passat Mk8 (#20493) (26/03/2021) <Jason Young>

VW MQB: Škoda Superb Mk3 (#20500) (27/03/2021) <Jason Young>

VW MQB: Update to Volkswagen Golf Mk7 (#20498) (29/03/2021) <Jason Young>

VW MQB: Volkswagen Atlas Mk1 (#20881) (12/05/2021) <Jason Young>

VW MQB: Add FW values for 2019 Volkswagen Golf GTI (#20882) (12/05/2021) <Jason Young>

VW MQB: Volkswagen Golf Alltrack Mk7 (#20893) (13/05/2021) <Jason Young>

VW MQB: Add FW values for 2018 Volkswagen Golf Alltrack (#20905) (14/05/2021) <Jason Young>

VW MQB: Add FW values for 2020 Volkswagen Jetta (#20936) (18/05/2021) <Jason Young>
VW MQB: Add FW for 2018 Volkswagen Atlas (#20938) (18/05/2021) <Jason Young>
VW MQB: Add FW for 2018 Volkswagen Atlas (#21068) (28/05/2021) <Jason Young>
VW MQB: Add FW for 2017 Volkswagen Golf R (#21110) (02/06/2021) <Jason Young>
VW MQB: Add FW for 2020 Volkswagen Tiguan (#21222) (11/06/2021) <Jason Young>
VW MQB: Switch to comfort blinker signal (#21253) (14/06/2021) <Jason Young>
VW MQB: Add FW for 2016 Golf R (#21254) (15/06/2021) <Jason Young>
VW MQB: Volkswagen Touran Mk2 (#21263) (15/06/2021) <Jason Young>
VW MQB: Add FW for 2018 Volkswagen Golf (#21388) (23/06/2021) <Jason Young>
VW MQB: Add FW for 2016 Volkswagen Tiguan (#21418) (28/06/2021) <Jason Young>
VW MQB: Add FW for 2019 Volkswagen Atlas (#21491) (06/07/2021) <Jason Young>
VW MQB: Add detail to Blinkmodi_02 (#402) (14/06/2021) <Jason Young>
VW MQB: Add FW for 2016 Volkswagen Golf R (#21663) (20/07/2021) <Jason Young>
VW MQB: Add FW for 2016 Škoda Octavia RS (#21689) (23/07/2021) <Jason Young>
VW MQB: Auto network location detection (#21671) (25/07/2021) <Jason Young>
VW MQB: Volkswagen T-Cross Mk1 (#21709) (25/07/2021) <Jason Young>

Comma.ai, to incentivize new contributions, promoted many initiatives to award the most active community members with different prizes. For instance, in 2020, during the “*Comma awards*”, the team of Comma.ai assigned a silver statuette to the members who made more contributions during the year. The prizes were assigned to different categories and the award ceremony was held in a live stream. The following are the results for year 2020:

- Pencilers of the year:
 - <https://github.com/pjlao307>
 - <https://github.com/erikbernheim> (Erik Bernheim)
 - <https://github.com/doktorsleepelss>
 - <https://github.com/spektor56> (Brett Sanderson)
- Video of the year:
 - <https://youtu.be/3Y67XKPmtY8> (\$7.000 Tesla Autopilot vs \$1.000 Openpilot: Self-Driving Test!)
 - Award goes to:
 - <https://youtube.com/user/MyTechMethods1> (Andy Slye)
 - <https://youtube.com/user/Atticus1337> (Logan LeGrand)
- Fingerprinter of the year:
 - <https://github.com/ErichMoraga> (Erich Moraga)
- Contributing to Openpilot and now joining as interns:
 - <https://github.com/ShaneSmiskol> (Shane Smiskol)
 - <https://github.com/YassineYousfi> (Yassine Yousfi)
- Contributor of the year:
 - <https://github.com/deanlee> (Dean Lee)

Assessing the quality of the contributions in an open-source software (OSS) is not an easy task: unlike industrial software, OSS is often developed under a non-traditional structure and, as a result, is seen as the product of teams composed almost exclusively of developers. This picture is of course incomplete at best, as is well known by those involved with OSS. While young projects can thrive under the guidance of lone developers or small unsupported teams, more mature projects usually benefit from contributions to the project that transcend code. These non-code contributions may include, for example: moderating communication channels associated with the project or its issue tracker(s), fielding questions, outreach, infrastructure, governance, funding acquisition, documentation, or even mere attention, these contributions are all crucial determinants of a project's continued success. [54]

To have a complete picture of the community contributions made to openpilot over time, in the following analysis will be considered both code contributions and non-code contributions. To simplify the process of data mining, the analysis will be supported by an ad-hoc tool developed for this purpose, that from now on will be referenced as *contributors-profiler*. The tool will collect the data leveraging the GitHub APIs and for each contributor it will compute the key performance indicators (KPIs). For the code contributions, we will consider the number of commits, and the ratio between the successful GitHub actions executed after the commits and the total amount of actions triggered by the contributor's commits, as a proxy of the errors introduced with each commit. The non-code data will include the amount of issues made by each contributor and their activity on the Discord server, since it is the main place where the community interact. For the code contribution, the granularity will be of a week, both to speed up the execution and because a finer granularity won't be meaningful for this analysis' purpose.

One of the most active users in the community and also a member of the Comma.ai team is for sure **Adeeb Shihadeh**, with almost 1400 commits. Using the afore mentioned tool, we obtain the following output that shows how also the jobs success rate is very high.

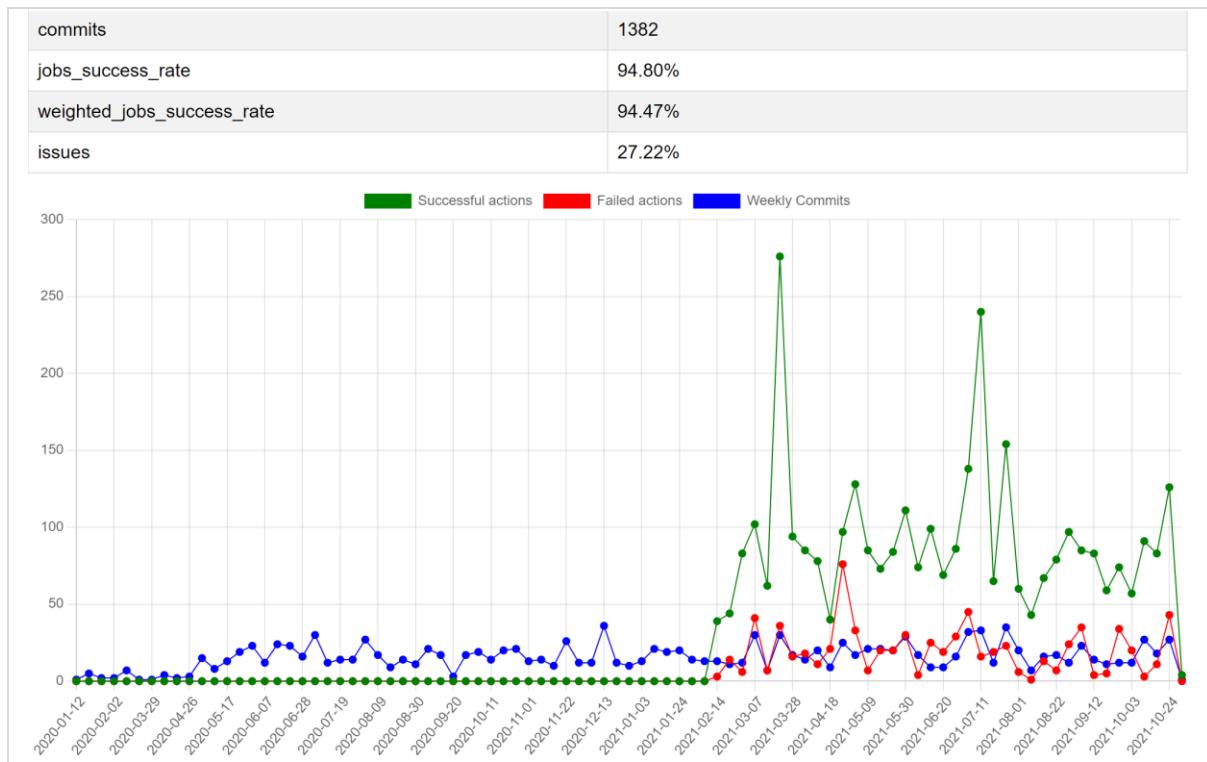


Figure 97 - output of the analysis of the contribution of Adeeb Shihadeh

This approach to estimate the quality on the commit based on the success of the jobs, however, has some limitations. The GitHub APIs have a limit of 1020 elements for each response: this is typically more than enough if we consider the average user and the considered granularity of a week, but in for the most active users this could lead to omit some jobs executions. Also, the number of requests for this high number of commits and jobs executed quickly surpasses the limit allowed by GitHub, blocking the execution of the tool.

The workflow testing the functionalities of Openpilot and its reliability was also introduced in a second moment, so as it can be evicted from

Another approach would be to consider the success rate of the workflows and not of single jobs, but in this case the result would consider as failed also the workflows failed for errors in the jobs relative to the build processes and less important tasks. On average there is a 10% of difference between the two approaches, and that makes the parameter not reliable.

Another performance indicator that could be used to assess the quality of the contribution could be the *entropy* of the commits. The software entropy takes its name from the entropy in real world, that can be defined as a measure of chaos that either stays the same or increases over time. In the case of software entropy, this indicator is a proxy of the chaos of a software, and measures of how specific each commit was in relation to the entire code base. Very specific commits only affect a small set of files, and thus have a low entropy. Commits that touch a large number of files are much less specific and have a higher entropy as a result. Software entropy impacts the overall quality of software systems. High entropy hinders developers from understanding the purpose of a piece of code and can cause developers to make sub-optimal changes and introduce bugs.

The overall software entropy of Openpilot can be measured using the tool *commit-entropy*, which allows to compute the average entropy per day and a 30-day rolling average. [55]

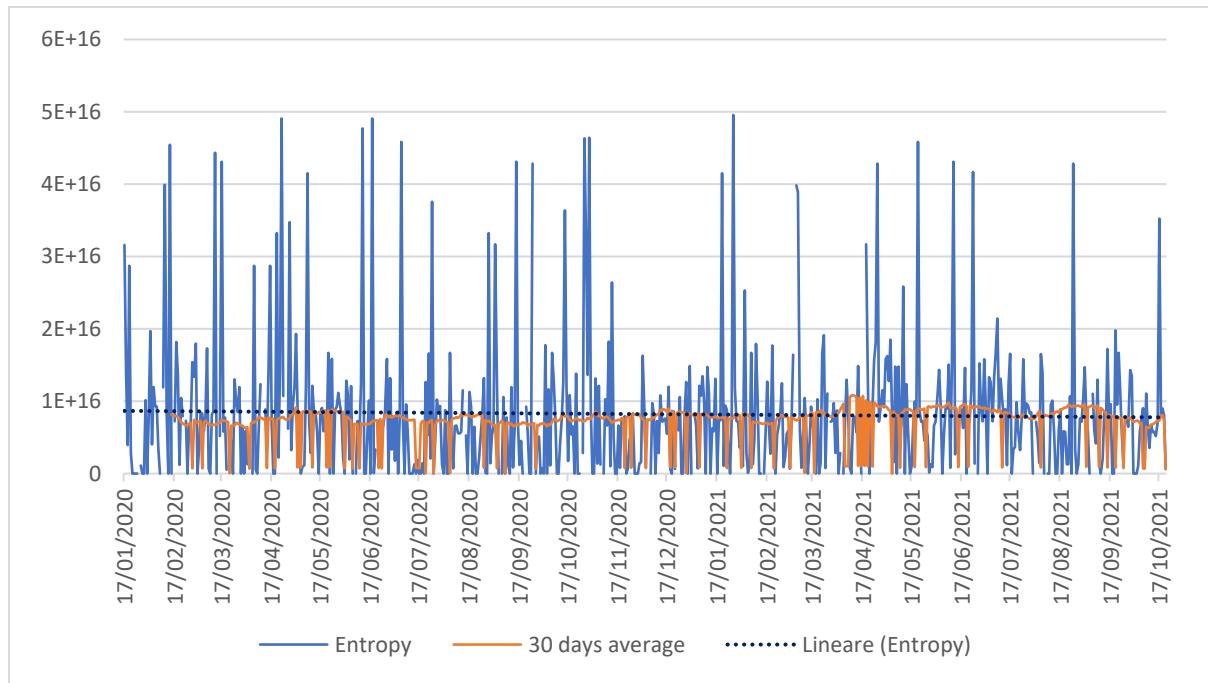


Figure 98 - Openpilot's software entropy

As turns out, the software entropy shows a descending trend, with the peaks close to the dates of the new releases.

The *commit-entropy* tool computes the entropy as $\log_2(\# \text{ of files changed})$, where the base 2 of the logarithm is given by the fact that the entropy is expressed in bits. Here is assumed that all the files have an equal probability of being changed.

To have more details on the software entropy that characterize Openpilot and to further investigate what is the entropy of the commits of each contributor, the *contributor-profiler* has been modified to compute the average entropy generated by the single user. This value can be considered as a proxy of the average quality of a commit, since the lower the entropy, the lower is the probability to introduce bugs in the code.

Combining these data with the results previously obtained with the data gathered on the Discord server we obtain a much clearer picture of the overall contributions made by the most active members of the Openpilot community.

Contributor	Commits	Issues sent	Discord messages	Entropy
adeebshihadeh	1.382	43	2.080	0,905
pd0wm	1.208	23	1.409	0,642
deanlee	560	17	0	1,005
HaraldSchafer	292	8	919	0,895
geohot	149	5	14.225	1,281
VirtuallyChris	138	1	4776	0,065
ZwX1616	126	0	132	1,049
jyoung8607	105	2	18.470	1,072
sshane	96	2	6.028	0,939
gregjhogan	95	3	978	0,674
ErichMoraga	71	0	56.451	0,070
grekiki	65	0	2120	1,317
robbederks	54	2	506	1,312
iejMac	39	0	0	1,259
rbiasini	27	0	0	1,586
vanillagorillaa	26	0	8.446	0,170
briskspirit	19	3	928	0,629
sunnyhaibin	19	1	2.925	0,654
mitchellgoffpc	17	0	0	1,072

Table 68 - Statistics of the main contributors

More meaningful values can be obtained by normalizing these results. To compute the normalization are needed the average and standard deviation for the considered measures, which can be easily retrieved using the GitHub APIs and the search functionalities of the Discord server¹.

	Commits	Issues sent	Discord messages
Total	5.085	5.871	848.141
Average (per user)	18,922	0,568	337
Standard deviation	112,470	3,355	13.165,863

Table 69 - Averages and standard deviations of the selected statistics

¹ Standard deviation of the Discord messages is overestimated due to the high amount of messages sent by Erich Moraga and the small sample considered.

The normalized results are reported in Table 70.

Contributor	Commits	Issues sent	Discord messages
adeebshihadeh	12,12	12,65	0,13
pd0wm	10,57	6,69	0,08
deanlee	4,81	4,90	-0,03
HaraldSchafer	2,43	2,22	0,04
geohot	1,16	1,32	1,05
VirtuallyChris	1,06	0,13	0,34
ZwX1616	0,95	-0,17	-0,02
jyoung8607	0,77	0,43	1,38
sshane	0,69	0,43	0,43
gregjhogan	0,68	0,72	0,05
ErichMoraga	0,46	-0,17	4,26
grekiki	0,41	-0,17	0,14
robbederks	0,31	0,43	0,01
iejMac	0,18	-0,17	-0,03
rbiasini	0,07	-0,17	-0,03
vanillagorillaa	0,06	-0,17	0,62
briskspirit	0,00	0,72	0,04
sunnyhaibin	0,00	0,13	0,20
mitchellgoffpc	-0,02	-0,17	-0,03

Table 70 - Normalized statistics for each contributor

These data are now comparable with each other, and by calculating the vectorial product among the three measures and weighting them for the sum of all the computed measures we obtain an indicator that takes into account both quantitative and qualitative data to estimate the total contributions made by a user.

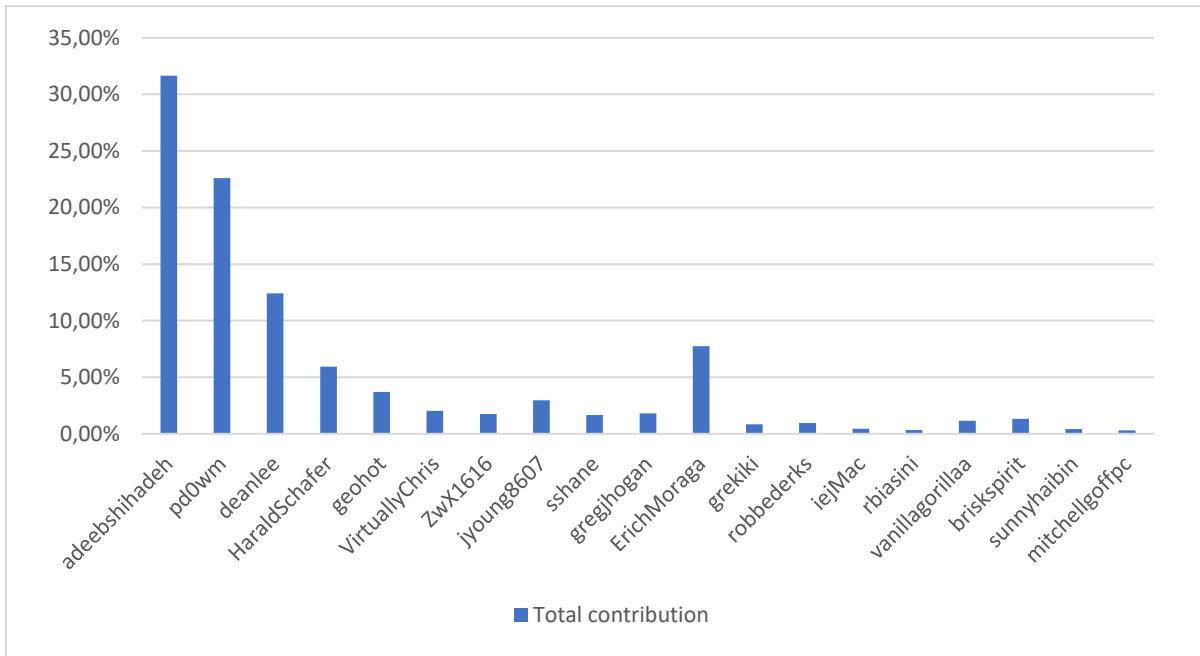


Figure 99 - Total contribution made by the most active users

5 Conclusions and future work

Openpilot turns out to be a truly vast and complex software. The aim of this thesis was to clearly define its framework, by focusing on the relationships that all the different components of which are made work and interact with each other.

The analysis of the packages gives insights on the main classes defined, describing their methods and the relationship with the other classes of the packages, and focusing on the theoretical basis behind the main design choices. Trying to see the big picture of the framework is not an easy task, given its size and the number of modules of which it is made, and that is the reason why a bottom-up approach was adopted for what concerns the analysis of the overall structure: analyzing the submodules first allows to have the necessary knowledge to better understand how openpilot leverages them to run its key processes.

Each submodule is itself a complex process that deals with different aspects of the software: we have Cereal, which clearly defines the messaging specifications through which the processes of Openpilot can communicate; the Common functionalities, providing utility functions that include a simple Kalman filter and functions to manipulate images; Rednose, that is an Extended Kalman Filter that smooths and improves the estimations of the different acquisitions; Laika, that pre-processes the GNSS data improving the precision of the positioning data; OpenDBC, that provides the specification of the messages exchanged on the CAN bus; Panda, that interfaces with the Panda device to let the Comma device communicate with the car. Each component is necessary to make Openpilot work reliably and safely.

Due to the risks that malfunctioning of Openpilot could bring, both for the driver and other people on the road, the software is heavily tested and has to respect strict security constraints. Testing the submodules is as crucial as testing openpilot itself: a malfunctioning of Cereal or Panda could lead to miss a break message when approaching a car or a person on the road; a miscalculation of the positioning of the car could make the model predict a wrong steering angle; a message not correctly encoded could lead to unexpected behavior of the car. This is the reason why many test cases are defined for all the modules of openpilot, and why each commit made to the repository has to pass these tests and be approved by the Comma.ai team. As turns out, all these precautions make Openpilot a very reliable software and its performances are comparable, and often higher, than the solutions offered by the other car manufacturers.

The community is a central part of the whole project: changes are made on the basis of the feedbacks given by the community members, and the community also contributed to the development of the software by sending their Pull Requests, many of which made to the final releases of Openpilot. With this thesis, one of the main objectives is to help the community even more, giving clear technical guidelines on what they will find in the GitHub repository and how the whole framework works, being the starting point of an official and constantly updated documentation of the software, maintained by the Comma.ai team and the community.

6 References

- [1] "comma.ai blog," Comma.ai, [Online]. Available: <https://blog.comma.ai/>.
- [2] Mindy News Blog, "How Machine Learning in Automotive Makes Self-Driving Cars a Reality," 12 February 2020. [Online]. Available: <https://mindy-support.com/news-post/how-machine-learning-in-automotive-makes-self-driving-cars-a-reality/>.
- [3] "A 2020 Theme: Externalization," 17 January 2020. [Online]. Available: <https://blog.Comma.ai/a-2020-theme-externalization/>.
- [4] S. Chengyao, "Decoding comma.ai/openpilot: the driving model," 11 November 2019. [Online]. Available: <https://medium.com/@chengyao.shen/decoding-comma-ai-openpilot-the-driving-model-a1ad3b4a3612>.
- [5] C. McCammon, S. Smiskol, quadmus and grekiki, "Tuning," 4 August 2021. [Online]. Available: <https://github.com/commaai/openpilot/wiki/Tuning>.
- [6] Sandeep, "PUBLISH-SUBSCRIBE (PUB-SUB) DESIGN PATTERN," 15 February 2016. [Online]. Available: <http://www.code2succeed.com/pub-sub/>.
- [7] P. Goliński, "PR #46: Use ZMQ on MacOS," 20 May 2020. [Online]. Available: <https://github.com/commaai/cereal/pull/46>.
- [8] P. Goliński, "PR #45: Fix potential segfault in MSGQPubSocket::connect," 20 May 2020. [Online]. Available: <https://github.com/commaai/cereal/pull/45>.
- [9] D. Lee, "PR #42: Sub pub master," 9 May 2020. [Online]. Available: <https://github.com/commaai/cereal/pull/42>.
- [10] D. Lee, "PR #58: Fix dlopen error after put socketmaster.cc in messaging," 10 June 2020. [Online]. Available: <https://github.com/commaai/openpilot-apks/pull/58>.
- [11] D. Lee, "PR #50: fix mac build," 6 June 2020. [Online]. Available: <https://github.com/commaai/cereal/pull/50>.

- [12] A. Shihadeh, "PR #84: allow prioritization of services in SubMaster," 31 August 2020. [Online]. Available: <https://github.com/commaai/cereal/pull/84>.
- [13] W. Melching, "Issue #41: Show warning when subscribing/publishing to queue that's not in service list," 8 April 2020. [Online]. Available: <https://github.com/commaai/cereal/issues/41>.
- [14] H. Schafer, "PR #107: Best practice," 3 January 2021. [Online]. Available: <https://github.com/commaai/cereal/pull/107>.
- [15] J. Wooning, "PR #135: some fixes and small changes for locationd in C++," 19 April 2021. [Online]. Available: <https://github.com/commaai/cereal/pull/135>.
- [16] S. Smikol, "PR #136: Automatically generate service ports," 19 April 2021. [Online]. Available: <https://github.com/commaai/cereal/pull/136>.
- [17] G. Hotz, "Issue #1038: Remove visionipc -- \$500 bounty," 2 February 2020. [Online]. Available: <https://github.com/commaai/openpilot/issues/1038>.
- [18] W. Melching and A. Shihadeh, "PR #101: Visionipc v2.0," 16 November 2020. [Online]. Available: <https://github.com/commaai/cereal/pull/101>.
- [19] W. Melching, "PR #183: Always free ION buffer," 26 July 2021. [Online]. Available: <https://github.com/commaai/cereal/pull/183>.
- [20] T. M. Zeng, "The Android ION memory allocator," 8 February 2012. [Online]. Available: <https://lwn.net/Articles/480055/>.
- [21] Wikipedia, "Kalman filter," 8 November 2021. [Online]. Available: https://en.wikipedia.org/wiki/Kalman_filter.
- [22] M. Vh, "PR #1890: Fix insecure temporary file creation," 18 July 2020. [Online]. Available: <https://github.com/commaai/openpilot/pull/1890>.
- [23] C. McCammon, "Issue #1882: Improve car battery management," 16 July 2020. [Online]. Available: <https://github.com/commaai/openpilot/issues/1882>.
- [24] R. Derks, "PR #1994: Car power integrator + power management refactor," 17 August 2020. [Online]. Available: <https://github.com/commaai/openpilot/pull/1994>.
- [25] Wikipedia, "Dilution of precision (navigation)," 25 May 2021. [Online]. Available: [https://en.wikipedia.org/wiki/Dilution_of_precision_\(navigation\)](https://en.wikipedia.org/wiki/Dilution_of_precision_(navigation)).

- [26] "ZED-F9P interface: u-blox F9 high precision GNSS receiver," 28 May 2020. [Online]. Available: https://www.u-blox.com/sites/default/files/ZED-F9P_InterfaceDescription_%28UBX-18010854%29.pdf.
- [27] S. Figiel, "PR #22: Remove get_prns_from_constellation," 9 May 2020. [Online]. Available: <https://github.com/commaai/laika/pull/22>.
- [28] "DBC Format," 11 October 2017. [Online]. Available: http://socialledge.com/sjsu/index.php/DBC_Format.
- [29] "data length code (DLC)," [Online]. Available: [https://www.datajob.com/it/definizione/39/data-length-code-\(dlc\).](https://www.datajob.com/it/definizione/39/data-length-code-(dlc).)
- [30] "Chffr: a dashcam that trains self-driving cars," 23 April 2019. [Online]. Available: <https://steemit.com/steemhunt/@oluwabori/chffr-chffr-a-dashcam-that-trains-self-driving-cars>.
- [31] A. Haden, "PR #18: Add chffr metrics," 10 September 2017. [Online]. Available: <https://github.com/commaai/opendbc/pull/18>.
- [32] J.-C. Thibault, "PR #49: Lots of correction, thanks to cabana!," 4 November 2017. [Online]. Available: <https://github.com/commaai/opendbc/pull/49>.
- [33] AdasCoder, "PR #147: Add files via upload," 14 March 2019. [Online]. Available: <https://github.com/commaai/opendbc/pull/147>.
- [34] R. Kołłataj, "PR #205: Add reference to CANdevStudio in README file," 16 January 205. [Online]. Available: <https://github.com/commaai/opendbc/pull/205>.
- [35] M. di Preez, "OBD II diagnostic interface pinout," 2 December 2017. [Online]. Available: https://pinoutguide.com/CarElectronics/car_obd2_pinout.shtml.
- [36] "CAN FD," 26 August 2021. [Online]. Available: https://en.wikipedia.org/wiki/CAN_FD.
- [37] Oracle, "Registering Interrupts," [Online]. Available: https://docs.oracle.com/cd/E23824_01/html/819-3196/interrupt-14.html. [Accessed 18 September 2021].
- [38] . K. Magdy, "STM32 USART / UART Tutorial," 13 June 2020. [Online]. Available: <https://deepbluembedded.com/stm32-usart-uart-tutorial/>.
- [39] Elettronica Open Source, "Transceiver CAN," 19 February 2020. [Online]. Available: <https://it.emcelettronica.com/transceiver-can>.

- [40] "Using Python and the libusb library with ADU USB Data Acquisition Products (Linux & Windows)," [Online]. Available: <https://www.ontrak.net/LibUSBPy.htm>. [Accessed 19 September 2021].
- [41] Zephyr, "ISO-TP Transport Protocol," 20 September 2021. [Online]. Available: https://docs.zephyrproject.org/latest/reference/networking/can_isotp.html.
- [42] P. Decker, "K-Line: Flexible Solutions for a Classic protocol," May 2015. [Online]. Available: https://assets.vector.com/cms/content/know-how/_technical-articles/K_Line_AutomotiveEETimesEurope_201505_PressArticle_EN.pdf.
- [43] Wikipedia, "MISRA C," 15 July 2021. [Online]. Available: https://en.wikipedia.org/wiki/MISRA_C.
- [44] C. Vickery, "PR #107: Implement WebUSB and upgrade WinUSB to 2.0," 6 April 2018. [Online]. Available: <https://github.com/commaai/panda/pull/107>.
- [45] G. Hogan, "PR #145: Unified Diagnostic Services (UDS) panda library (ISO 14229)," 16 October 2019. [Online]. Available: <https://github.com/commaai/panda/pull/145>.
- [46] R. Biasini, "PR #389: Power saving refactor," 21 November 2019. [Online]. Available: <https://github.com/commaai/panda/pull/389>.
- [47] C. Woei-Leong and H. Fei-Bin, "Implementation of the Rauch-Tung-Striebel Smoother for Sensor Compatibility Correction of a Fixed-Wing Unmanned Air Vehicle," November 2011. [Online]. Available: <https://doi.org/10.3390/s110403738>.
- [48] J. A. Matute, M. Marcano, A. Zubizarreta and J. Perez, "Longitudinal Model Predictive Control with comfortable speed planner," 2018. [Online]. Available: <http://dx.doi.org/10.1109/ICARSC.2018.8374161>.
- [49] Wikipedia, "Radar tracker," 21 September 2020. [Online]. Available: https://en.wikipedia.org/wiki/Radar_tracker.
- [50] P. Sala, "Camera Models and Parameters," 7 20 2006. [Online]. Available: <http://ftp.cs.toronto.edu/pub/psala/VM/camera-parameters.pdf>.
- [51] H. Schafer, "Neural networks in openpilot," 12 October 2021. [Online]. Available: <https://github.com/commaai/openpilot/tree/master/models>.
- [52] "OpenMAX overview," Khronos, [Online]. Available: <https://www.khronos.org/openmax/>.

- [53] Consumer Reports, "Active Driving Assistance Systems: Test Results and Design Recommendation," November 2020. [Online]. Available: <https://data.consumerreports.org/wp-content/uploads/2020/11/consumer-reports-active-driving-assistance-systems-november-16-2020.pdf>.
- [54] J.-G. Young, A. Casari, K. McLaughlin, M. Z. Trujillo, L. Hébert-Dufresne and J. P. Bagrow, "Which contributions count? Analysis of attribution in open source," 19 March 2021. [Online]. Available: https://www.researchgate.net/publication/350311749_WWhich_contributions_count_Analysis_of_attribution_in_open_source.
- [55] K. Wanrooij, L. Kian Seong and D. Stevens, "commit-entropy," 17 April 2016. [Online]. Available: <https://github.com/GripQA/commit-entropy>.
- [56] "What is GNSS?," 19 11 2020. [Online]. Available: <https://www.gsa.europa.eu/european-gnss/what-gnss>.
- [57] S. Schaer and W. Gurtner, "IONEX: The IONosphere Map EXchange," 17 09 2015. [Online]. Available: <http://ftp.aiub.unibe.ch/ionex/draft/ionex11.pdf>.
- [58] "NASA's Archive of Space Geodesy Data," [Online]. Available: <https://cddis.nasa.gov/>.
- [59] K. Nobuaki, "GPS/GNSS: Satellite Navigation," 28 07 2014. [Online]. Available: http://www.denshi.e.kaiyodai.ac.jp/gnss_tutor/pdf/basic_of_gnss.pdf.
- [60] "Interfacing with external C code," [Online]. Available: http://docs.cython.org/en/latest/src/userguide/external_C_code.html.
- [61] "Understanding and implementing CRC (Cyclic Redundancy Check) calculation," March 2019. [Online]. Available: http://www.sunshine2k.de/articles/coding/crc/understanding_crc.html.
- [62] "Robotics Knowledgebase," 15 August 2017. [Online]. Available: <https://roboticsknowledgebase.com/wiki/sensing/delphi-esr-radar/>.
- [63] "GNSS Science Support Centre," [Online]. Available: <https://gssc.esa.int/>.
- [64] "A panda and a cabana: How to get started car hacking with comma.ai," 7 July 2017. [Online]. Available: <https://comma-ai.medium.com/a-panda-and-a-cabana-how-to-get-started-car-hacking-with-comma-ai-b5e46fae8646>.
- [65] L. LeGrand, "A History of Comma.ai," 26 February 2020. [Online]. Available: <https://www.youtube.com/watch?v=OMcdMoa9wnc>.

7 Appendixes

Category	Category description	Channel	Channel description
onboarding	Includes the channels where a user that accesses for the first time to the server will be able to interact. In the onboarding channels, only the Comma.ai staff users can write new messages, while other users can only visualize them.	<i>guidelines</i>	Explains the guidelines of the server and the community rules. After accepting them, the role community member will be assigned to the user.
		<i>announcements</i>	Channel where the Comma.ai staff make their announcements, which include new product releases, updates on the development, and articles that they want to share with the community.
general	Includes the channels accessible by the community members, including:	<i>lobby</i>	A channel intended to help to redirect to the correct channel for a discussion.
		<i>Openpilot-experience</i>	A channel for posting Openpilot videos and experiences
		<i>installation-help</i>	A channel to ask questions on the installation process and get help in case of problems during the installation. To get extended support from the community, the manufacturer-specific channel groups are recommended.
		<i>comma-shipping</i>	An informative channel to help the community quickly reach the support pages regarding the shipping and returns.
		<i>Openpilot-faq</i>	Users here can post frequently asked questions that don't have a response on the official FAQ page of Openpilot.
		<i>community-wiki</i>	General questions and discussions on the official wiki of Openpilot.
		<i>for-sale</i>	Here users can sell their unused official comma devices so that

Category	Category description	Channel	Channel description
			other members of the community can buy them second-handed and at a lower price.
		<i>bug-report</i>	An informative channel where can be found the guidelines to correctly report a bug. Bugs can be reported by opening a new issue on GitHub.
		<i>comma-aps</i>	Channel dedicated to the comma's applications, including comma prime, explorer, cabana, and comma connect.
		<i>comma-clubhouse</i>	A voice channel where the community can hang out.
development	Here can be found the channels accessible by developers (users with a dev role) and where discussions about the development of Openpilot take place. Each aspect of the development is addressed by a specific channel	<i>feature-requests</i>	Channel to discuss new ideas for Openpilot and leave feedback on existing features.
		<i>dev-Openpilot</i>	The main channel to discuss the development of Openpilot. It is not a channel to ask for help, instead, it is only dedicated to the development of new features and bug fixing.
		<i>tuning</i>	A channel where to discuss how to properly approach the tuning of a car. Tuning means adjusting the control parameters to best suit a specific car.
		<i>comma-api</i>	The channel where developers can discuss the usage of comma API and ask for help in case of issues related to the APIs.
		<i>join-development</i>	The channel from where users can accept to join the development channels after a bot verification. After the verification, the bot will automatically add the dev role to the user.
		<i>flexray</i>	FlexRay is an alternative to CAN and is adopted by many car manufacturers, especially Audi, BMW, and Mercedes-Benz. In this channel, the main discussed topics regard the specific of the FlexRay protocol and how

Category	Category description	Channel	Channel description
			Openpilot can use it instead of the default CAN protocol.
		<i>serial-steering</i>	Like in the case of FlexRay, some other car models adopt a protocol different than CAN. This is the case of Honda and Acura, which send steering control data over 2 LIN communication bus lines instead of the CAN bus. This channel is dedicated to those developers that want to port Openpilot on these car models, allowing them to ask the Comma.ai staff and other developers in the community for clarification and help on the topic.
		<i>topic-custom-models</i>	Channel to discuss the creation, training, and test of new models to perform machine learning on the acquired images, to allow Openpilot to distinguish key elements such as cars and lines.
		<i>comma-pencil</i>	The channel where to discuss changes to be made to comma10k, a repository that includes 10.000 PNGs of real driving captured from the comma fleet. Users can contribute to comma10k by manually labeling the elements in the images using a precise color code.
		<i>fw-mods</i>	Channel for discussion on how to use firmware mods and how to be safe while using them.
		<i>torque-interceptor</i>	This channel is for talking about how to build a torque interceptor the right way. A torque interceptor is a device that uses a sensor to detect if the user is applying torque to the wheel, multiplies that torque appropriately in software, and applies that torque with a motor to the steering column. Safety is critical, therefore only developers with experience with ASIL-D Hardware and software

Category	Category description	Channel	Channel description
			engineering are recommended to work on this project.
		<i>topic-maps</i>	Channel to discuss HD Mapping and how HD maps can be used to lower the computation time and the bandwidth that Openpilot needs to take decisions.
		<i>Openpilot-simulation</i>	A channel for simulation discussion. The simulator provided by Comma.ai is CARLA and in this channel, the developers can share their experience, report issues ask for help on how to run the simulation.
		<i>custom-forks</i>	A channel for custom fork discussion. All forks discussed here must be using upstream Honda/Toyota/Hyundai panda safety and immediately disengage when the brake is pressed. Also, they should not allow complete disabling of driver awareness features.
hardware	Here users can discuss the hardware needed to run Openpilot. They have to be developers (they must have the dev role in the Discord server) to be able to interact with these channels:	<i>hw-two-eon</i>	Channel to discuss the 2 main devices where Openpilot can be installed, EON and comma two.
		<i>hw-panda</i>	Discussion on panda hardware, the universal car interface used in by EON and comma two.
		<i>hw-pedal</i>	A channel to discuss the comma pedal interceptor, a gas pedal interceptor for Honda/Acura that allows to virtually press the pedal, enabling stop and go cruise control on select cars.
		<i>hw-unofficial</i>	A channel to discuss the hardware developed by the community and based on the EON style form factor. Also, on the official wiki, there are the guidelines to correctly flash NEOS on custom hardware and some replacement parts commonly used by the community and suitable for the EON form factor.

Category	Category description	Channel	Channel description
vehicle specific	In this category can be found channels to discuss the brand-specific topics, such as porting Openpilot to a new car model and solving issues with specific brands. Each channel of this category corresponds to a specific brand or group of brands that have similar characteristics in terms of technology adopted by the manufacturers, and these include:	<i>chrysler-jeep-ram</i>	Chrysler and Jeep related discussions
		<i>ford</i>	Ford related discussions
		<i>gm</i>	GM related discussions
		<i>honda-acura</i>	Honda and Acura related discussions
		<i>hyundai-kia-genesis</i>	Hyundai, Kia and Genesis related discussions
		<i>mazda</i>	Mazda related discussions
		<i>nissan</i>	Nissan related discussions
		<i>subaru</i>	Subaru related discussions
		<i>tesla</i>	Tesla related discussions
		<i>toyota-lexus</i>	Toyota and Lexus related discussions
		<i>volkswagen-audi</i>	Volkswagen and Audi related discussions
		<i>volvo</i>	Volvo related discussions
		<i>old-cars</i>	Old cars related discussions
		<i>other-cars</i>	Other cars related discussions
non English	This category comprehends channels for non-English users and general topics can be discussed here.	<i>lang-chinese</i>	Discussion channel in Chinese
		<i>lang-french</i>	Discussion channel in French
		<i>lang-german</i>	Discussion channel in German
		<i>lang-russian</i>	Discussion channel in Russian
		<i>lang-spanish</i>	Discussion channel in Spanish
		<i>lang-korean</i>	Discussion channel in Korean
		<i>lang-portuguese</i>	Discussion channel in Portuguese
		<i>lang-vietnamese</i>	Discussion channel in Vietnamese
		<i>lang-japanese</i>	Discussion channel in Japanese

Table 71 - Discord server's channels description

Commit summary	Date	Contributor
add packet timings to Plan and PathPlan	13/06/2019	Willem Melching
add socket valid to plan and pathplan	14/06/2019	Willem Melching
canError is a better name than commIssue	15/06/2019	Riccardo Biasini
radarCommIssue events is renamed to radarCanError	17/06/2019	Riccardo Biasini
commIssue is standard for socketsDead	17/06/2019	Riccardo Biasini
added comm issue events	17/06/2019	Riccardo Biasini

Commit summary	Date	Contributor
removerted very recent changes. subLog tracking will be done differently	19/06/2019	Riccardo Biasini
Bug fix	19/06/2019	Riccardo Biasini
Add common 'valid' field to better diagnose when the published data should be trusted	20/06/2019	Riccardo Biasini
Added canValid bool to carState	22/06/2019	Riccardo Biasini
cleaner name for mpcSolutionValid	26/06/2019	Riccardo Biasini
Deprecated model valid bool from PathPlan	26/06/2019	Riccardo Biasini
Added carEvents for qlogs	27/06/2019	Riccardo Biasini
in CarParams, use the enum for SafetyModel	28/06/2019	Riccardo Biasini
Added carParams to event union type	28/06/2019	Riccardo Biasini
add lead stuff	09/07/2019	Harald Schafer
back	10/07/2019	Harald Schafer
Log can errors from panda	11/07/2019	Riccardo Biasini
add second model lead	15/07/2019	Harald Schafer
different name	15/07/2019	Harald Schafer
added tooDistracted event	16/07/2019	Riccardo Biasini
added HW type to support various panda versions	16/07/2019	Riccardo Biasini
add model prob	17/07/2019	Harald Schafer
add whether point is detected by radar	18/07/2019	Harald Schafer
add posenet debug fields to LiveParameters	18/07/2019	Willem Melching
rename to posenetValid	18/07/2019	Willem Melching
add alert for invalid posenet	18/07/2019	Willem Melching
add speed	20/07/2019	Harald Schafer
Blackpanda (#4)	24/07/2019	Riccardo Biasini
hasGps is a better name than hasGpsAntenna	24/07/2019	Riccardo Biasini
add eye stuff	24/07/2019	Harald Schafer
add longitudinal plan source	24/07/2019	Willem Melching
add decelForModel	24/07/2019	Willem Melching
Add fields for LQR lateral control	25/07/2019	Willem Melching
remove hwType from ThermalData. Decided to have health at higher freq instead. This will make last 24H of collected data unreadable. Sorry.	25/07/2019	Riccardo Biasini
deprecate old dm model output	25/07/2019	ZwX1616
add camera rpy angle msg	26/07/2019	ZwX1616
angle calib desc	26/07/2019	ZwX1616
use enum for alert sounds	29/07/2019	Adeeb Shihadeh
add blink msg	02/08/2019	ZwX1616
add soundsUnavailable event	06/08/2019	Adeeb Shihadeh
add eps torque to carstate	09/08/2019	Willem Melching
addtimes	14/08/2019	Harald Schafer
add lqr output to LQRState	30/08/2019	Willem Melching
add desire to controlsState	04/09/2019	Willem Melching
move desire to pathplan	04/09/2019	Willem Melching
angleModelBias is deprecated	04/09/2019	Willem Melching
Add dashcamOnly flag	05/09/2019	Riccardo Biasini
Add lane change states to pathPlan	06/09/2019	Willem Melching
Add lane change events	06/09/2019	Riccardo Biasini

Commit summary	Date	Contributor
allow to specify a passive safety model in CarParams	08/09/2019	Riccardo Biasini
add gm passive safety model	08/09/2019	Riccardo Biasini
placeholders for Mazda, Nissan and vw safety models	10/09/2019	Riccardo Biasini
Added invalidGiraffeToyota event	12/09/2019	Riccardo Biasini
Read RPM from fan connected to Panda	20/09/2019	Riccardo Biasini
add HW type for UNO	20/09/2019	Riccardo Biasini
Added usbPowerOn to health	01/10/2019	Riccardo Biasini
added event about internet connection	01/10/2019	Riccardo Biasini
add ldw to visual HUD alerts (#5)	01/10/2019	Chris Souers
usbPowerMode is more useful to log and it comes from panda	02/10/2019	Riccardo Biasini
change order of UsbPowerMode to preserve panda lib behavior	02/10/2019	Riccardo Biasini
Safety cereal (#9)	02/10/2019	Riccardo Biasini
let's spell wolkswagen	02/10/2019	Riccardo Biasini
Add fields and states required for robust Volkswagen safety compliance (#8)	04/10/2019	Jason Young
add none/invalid USB power mode	04/10/2019	Riccardo Biasini
Fix wrong event enum	05/10/2019	Riccardo Biasini
distinguish between ignition_line and ignition_can	23/10/2019	Riccardo Biasini
added safety model to health packet	25/10/2019	Riccardo Biasini
add ir pwr field	28/10/2019	ZwX1616
add meta	08/11/2019	Harald Schafer
oops bad number	08/11/2019	Harald Schafer
Add struct to log FW version	12/11/2019	Riccardo Biasini
add steeringRateLimited to car.capnp	13/11/2019	Willem Melching
add saturated flags to indi and lqr logs	13/11/2019	Willem Melching
Add fault status to health	14/11/2019	Riccardo Biasini
add front frame	14/11/2019	Comma Device
deprecate irpwrr	14/11/2019	ZwX1616
disengage	14/11/2019	Harald Schafer
noOutput safety mode is now called silent	20/11/2019	Riccardo Biasini
added power save state to health packet	21/11/2019	Riccardo Biasini
Add uptime to health	22/11/2019	Riccardo Biasini
Added radar time step to car params	22/11/2019	Riccardo Biasini
20Hz for radar time step is very standard	22/11/2019	Riccardo Biasini
log Panda fault types	27/11/2019	Riccardo Biasini
Added communityFeature bit detection to CarParams	03/12/2019	Riccardo Biasini
Added communityFeatureDisallowed event	03/12/2019	Riccardo Biasini
log mem available and CPU perc in thermald	05/12/2019	Riccardo Biasini
adding low memory event	05/12/2019	Riccardo Biasini
for legacy-testing reasons, better to define the used percent instead of avail	05/12/2019	Riccardo Biasini
log stock AEB events	06/12/2019	Riccardo Biasini
Remove plusFrame socket in favor of UiLayoutState	10/12/2019	Andy Haden
Add ldw alert	10/12/2019	Riccardo Biasini
no l/r distinction for LDW	10/12/2019	Riccardo Biasini
Add stock Fcw to carState	11/12/2019	Riccardo Biasini

Commit summary	Date	Contributor
add pa0 temp to ThermalData	12/12/2019	Willem Melching
steerLimitTimer should be car dependent	12/12/2019	Riccardo Biasini
Add carUnrecognized event	13/12/2019	Riccardo Biasini
add more stuff to fw log in CarParams	17/12/2019	Willem Melching
Add canRxErrs to health	20/12/2019	Riccardo Biasini
add longitudinal	20/12/2019	Harald Schafer
better name	21/12/2019	Harald Schafer
separate Honda safety models between Bosch Giraffe and Bosch Nidec	21/12/2019	Riccardo Biasini
Reserve safety #21 for VAG PQ35/PQ46/NMS	29/12/2019	Jason Young
Add radar comm issue error	04/01/2020	Riccardo Biasini
add can error counter to controlsState	14/01/2020	Willem Melching
add face stds	14/01/2020	ZwX1616
add uncertainty event name	16/01/2020	ZwX1616
LateralParams	19/01/2020	Greg Hogan
standalone monitorstate (#23)	28/01/2020	ZwX1616
better put	28/01/2020	ZwX1616
add fingerprintSource to carParams	30/01/2020	Willem Melching
unconfusing and accessible to all	31/01/2020	ZwX1616
add networkType to thermal	31/01/2020	valish
Added ublox hw status	04/02/2020	Robbe
Added offroad power usage	08/02/2020	Robbe
add fixed fingerprintSource	12/02/2020	Willem Melching
add Honda ecus (#29)	18/02/2020	jpancotti
add espDisabled to carState (#30)	25/02/2020	Willem Melching
rigour	26/02/2020	Harald Schafer
or rigor in American	26/02/2020	Harald Schafer
improvements	26/02/2020	Harald Schafer
already exists	26/02/2020	Harald Schafer
deprecate	26/02/2020	Harald Schafer
Add Subaru pre-Global safety mode Biasini	27/02/2020	Riccardo Biasini
pulse desire and e2e	29/02/2020	Harald Schafer
Add blindspot cereal values (#26)	04/03/2020	Willem Melching
fix duplicate ordinals	04/03/2020	Willem Melching
val valid is confusing	05/03/2020	Harald Schafer
support for end of log sentinel (#34)	05/03/2020	George Hotz
not everyone likes gpstime	05/03/2020	Harald Schafer
add networkStrength to thermal (#36)	07/03/2020	Andrew Valish
6log focus state	12/03/2020	ZwX1616
solve by renaming event name instead of service	21/03/2020	Willem Melching
Add invalid lkas setting alert	28/03/2020	Willem Melching
add speedTooHigh alert	31/03/2020	Willem Melching
Additional car params auto-detection in support of VW (#38)	31/03/2020	Jason Young
Add the laneChangeBlocked Event (#40)	06/04/2020	Arne Schwarck
UiLayoutState: add 'none' app	08/04/2020	andyh2
UiLayoutState: add mockEngaged for onboarding	08/04/2020	andyh2

Commit summary	Date	Contributor
Add all panda fault types to health	09/04/2020	Willem Melching
Add comment to faulttype	09/04/2020	Willem Melching
add preview driver flag	15/04/2020	ZwX1616
Add relayMalfunction alert	16/04/2020	Willem Melching
add sharpness metric	20/04/2020	ZwX1616
add repeated warning2	22/04/2020	ZwX1616
add carState.steerWarning and carState.steerError	01/05/2020	Adeeb Shihadeh
don't duplicate ordinals	01/05/2020	Adeeb Shihadeh
gasPressed event	04/05/2020	Adeeb Shihadeh
stockFcw event	07/05/2020	Adeeb Shihadeh
add alert event type	11/05/2020	Adeeb Shihadeh
move remaining alerts to car events	12/05/2020	Adeeb Shihadeh
remove unnecessary new event type	13/05/2020	Adeeb Shihadeh
mark unused car events as deprecated	14/05/2020	Adeeb Shihadeh
add white panda deprecation events	15/05/2020	Willem Melching
add OK flags to locationd output	20/05/2020	Willem Melching
add default values for backwards compat	20/05/2020	Willem Melching
add gpsOK flag to liveLocationKalman	28/05/2020	Willem Melching
add canErrorPersistent event	29/05/2020	Adeeb Shihadeh
add belowEngageSpeed event	03/06/2020	Adeeb Shihadeh
add validLen to PathData	03/06/2020	George Hotz
add recover state	12/06/2020	ZwX1616
deprecate canErrorPersistent	13/06/2020	Adeeb Shihadeh
add noGps event (#55)	13/06/2020	Willem Melching
add to enum	13/06/2020	ZwX1616
add Hyundai legacy safety mode	13/06/2020	Adeeb Shihadeh
add wrongCruiseMode event	15/06/2020	Adeeb Shihadeh
add adaptive cruise flag to cruiseState	15/06/2020	Adeeb Shihadeh
laneChangeBlocked	16/06/2020	Adeeb Shihadeh
add calibrated orientation	16/06/2020	Harald Schafer
add neosUpdateRequired event	16/06/2020	Adeeb Shihadeh
increment ordinal	16/06/2020	Adeeb Shihadeh
add sensorsOK field to LiveLocationKalman	22/06/2020	Willem Melching
Dos	29/06/2020	Robbe Derkx
add sunglasses prob	02/07/2020	ZwX1616
add modelDlagging event	06/07/2020	Adeeb Shihadeh
k-line 5 init fault type (#61)	06/07/2020	Greg Hogan
add frameAge to ModelData	10/07/2020	Adeeb Shihadeh
add frameDropPerc (#63)	14/07/2020	Adeeb Shihadeh
add hyundaiCommunity	27/07/2020	Adeeb Shihadeh
add deviceStable	03/08/2020	Adeeb Shihadeh
add deviceFalling event	04/08/2020	Adeeb Shihadeh
add calib spread metric	06/08/2020	Harald Schafer
add validBlocks for calibration	06/08/2020	Adeeb Shihadeh
Added car battery capacity (#79)	17/08/2020	robbederkx
more fault types (#80)	17/08/2020	robbederkx
wideFrame	22/08/2020	ZwX1616

Commit summary	Date	Contributor
ThermalData cleanup (#81)	24/08/2020	Willem Melching
add sensor source	25/08/2020	Willem Melching
deprecate rhdChecked field	29/08/2020	Adeeb Shihadeh
add light sensor source	10/09/2020	Willem Melching
add temperature to sensor packet	21/09/2020	Willem Melching
new model packet (#86)	22/09/2020	Harald Schafer
cleanup deprecated car events	22/09/2020	Adeeb Shihadeh
add fanMalfunction event	05/10/2020	Adeeb Shihadeh
alert cleanup (#94)	15/10/2020	Adeeb Shihadeh
Cereal: Typo fix (#96)	17/10/2020	vanillagorillaa
add launch script log field to boot log	21/10/2020	Adeeb Shihadeh
add cameraMalfunction event	22/10/2020	Adeeb Shihadeh
fix duplicate ordinal	22/10/2020	Adeeb Shihadeh
events for grey panda deprecation	24/10/2020	Adeeb Shihadeh
just white for now	27/10/2020	Adeeb Shihadeh
giraffe are extinct	30/10/2020	Adeeb Shihadeh
deprecate deprecation fields	03/11/2020	Adeeb Shihadeh
add legacy stds (#99)	11/11/2020	Harald Schafer
front/wide encodeIdx	17/11/2020	ZwX1616
add model execution time	19/11/2020	Harald Schafer
add timestamps	19/11/2020	ZwX1616
make clear	20/11/2020	Harald Schafer
add execution time to driverState	20/11/2020	Adeeb Shihadeh
modelLagWarning	23/11/2020	Adeeb Shihadeh
add raw predictions	23/11/2020	ZwX1616
deprecate model lag warning	24/11/2020	George Hotz
comm issue warning	27/11/2020	Adeeb Shihadeh
Add HW types (#102)	03/12/2020	Willem Melching
Added minSpeedCan to CarParams to parametrize MIN_CAN_SPEED to interfaces (#103)	03/12/2020	Igor
log DSP execution time	09/12/2020	Adeeb Shihadeh
Added stoppingBrakeRate to CarParams to parametrize STOPPING_BRAKE_RATE to interfaces (#104)	11/12/2020	Igor
eon deprecation event	11/12/2020	Adeeb Shihadeh
add GPU execution time	12/12/2020	Adeeb Shihadeh
Parametrize startingBrakeRate (#106)	16/12/2020	Igor
remove commIssueWarning	17/12/2020	Adeeb Shihadeh
deprecate commIssueWarning	17/12/2020	Adeeb Shihadeh
deprecate internet connectivity needed event	22/12/2020	Adeeb Shihadeh
Adding breakpoints to INDI lateral tuning (#108)	07/01/2021	Igor
add maxSteerAngle to car.capnp (#110)	14/01/2021	Greg Hogan
add GPS malfunction event	14/01/2021	Adeeb Shihadeh
update pathPlan message	19/01/2021	Harald Schafer
add dm is_active	19/01/2021	ZwX1616
add e2e dm states	25/01/2021	ZwX1616
fix indexs	25/01/2021	ZwX1616
add managerState (#111)	26/01/2021	Willem Melching

Commit summary	Date	Contributor
Add MON_HW2 ublox message	29/01/2021	Robbe Derks
split out deprecated structs into separate schema file (#113)	02/02/2021	Adeeb Shihadeh
cleanup + comments (#116)	04/02/2021	Adeeb Shihadeh
this is a percent	12/02/2021	Adeeb Shihadeh
Best practice (#107)	17/02/2021	Harald Schafer
Change params value to take Data instead of Text.	26/02/2021	Maksym Sobolyev
deprecate gpsLocation	01/03/2021	Adeeb Shihadeh
Planner outputs curvature (#123)	12/03/2021	Willem Melching
gpsLocationDEPRECATED -> gpsLocationDEPRECATED	12/03/2021	Greg Hogan
Improve comments on CarParams from manual	15/03/2021	qadmus
add unfiltered curvatures (#127)	16/03/2021	Harald Schafer
add safetyParam to PandaState	17/03/2021	Willem Melching
deprecate UiLayoutState	25/03/2021	Adeeb Shihadeh
add event for an always-on dashcam alert	26/03/2021	Adeeb Shihadeh
deprecate oneplus event	26/03/2021	Adeeb Shihadeh
4222bc91 - Events cleanup	07/04/2021	Adeeb Shihadeh
Fuzzy FW match alert and carParams (#134)	20/04/2021	Willem Melching
add steerTempUnavailableUserOverride event	21/04/2021	Adeeb Shihadeh
add bsm to car params	25/04/2021	Adeeb Shihadeh
add controls initializing event	27/04/2021	Comma Device
turn pstore into map (#140)	12/05/2021	Willem Melching
add flags (#143)	14/05/2021	Harald Schafer
add usbError event (#145)	15/05/2021	Adeeb Shihadeh
add networkInfo struct to deviceState (#146)	17/05/2021	Willem Melching
add filter age (#148)	18/05/2021	Harald Schafer
add extra field to NetworkInfo	18/05/2021	Willem Melching
rename to deprecate (#151)	19/05/2021	Shane Smiskol
add last athena ping time to deviceState	19/05/2021	Adeeb Shihadeh
add cameraError event: (#149)	20/05/2021	Willem Melching
New model outputs (#155)	20/05/2021	Mitchell Goff
add NetworkInfo.state	20/05/2021	Willem Melching
add osVersion to initData	20/05/2021	Adeeb Shihadeh
split up camera events	21/05/2021	Adeeb Shihadeh
add signal to sentinel (#159)	25/05/2021	Willem Melching
add harness status to panda state (#161)	31/05/2021	robbederks
add cvt transmission type (#162)	03/06/2021	Shane Smiskol
Add stock LKAS/LDW camera detection flag	03/06/2021	Jason Young
Check resets (#163)	03/06/2021	Willem Melching
Permanent joystick debug alert (#158)	03/06/2021	Shane Smiskol
add heartbeatLost field to pandaState	05/06/2021	Adeeb Shihadeh
add no fw startup event	07/06/2021	Adeeb Shihadeh
next gen outputs	09/06/2021	ZwX1616
new lateral log for debug mode (#166)	10/06/2021	Shane Smiskol
add support for trajectory packet (#168)	18/06/2021	Harald Schafer
add ethernet network type	23/06/2021	Adeeb Shihadeh

Commit summary	Date	Contributor
add gpuUsagePercent to DeviceState (#170)	29/06/2021	Willem Melching
Lateral refactor (#169)	01/07/2021	Harald Schafer
deprecate CarParams.enableCamera	07/07/2021	Adeeb Shihadeh
deprecated (#175)	08/07/2021	Harald Schafer
refactor (#176)	08/07/2021	Harald Schafer
Refactor camerad logging (#174)	08/07/2021	Willem Melching
make qlogs small again (#178)	08/07/2021	Adeeb Shihadeh
rename steerTempUnavailableUserOverride -> steerTempUnavailableSilent	10/07/2021	Adeeb Shihadeh
rename carState.enableCruise -> carState.pcmCruise	10/07/2021	Adeeb Shihadeh
add lsm6ds3trc sensor source	12/07/2021	Willem Melching
add mmc5603nj sensor source	12/07/2021	Willem Melching
add uploaderState (#179)	14/07/2021	Adeeb Shihadeh
panda: rename fault interruptRateTim9 -> interruptRateTick (#180)	14/07/2021	Igor
add jerks (#181)	14/07/2021	Harald Schafer
Add RP to log.capnp (#182)	22/07/2021	Igor Biletskyy
change name for red panda in log.capnp (#185)	03/08/2021	Igor Biletskyy
consistent name	04/08/2021	Harald Schafer
add accel	12/08/2021	Harald Schafer
comment	12/08/2021	Harald Schafer

Table 72 - Commits adding new parameters to capnp files in cereal

Commit summary	Date	Contributor
OpenDBC updates	27/07/2017	George Hotz
OpenDBC updates	27/07/2017	George Hotz
Nidec radar: name change to RADAR_DIAGNOSTIC msg	22/09/2017	Riccardo Biasini
Acura RDX 2018 (#53)	09/12/2017	vanillagorillaa
Parking brake light (#54)	13/12/2017	vanillagorillaa
Adds rdx (#56)	18/12/2017	Ted Slesinski
Define Cruise Buttons (#62)	23/12/2017	vanillagorillaa
move acura rdx to generator folder	29/01/2018	Willem Melching
fix acura rdx dbc, import was missing	31/01/2018	Willem Melching
acura rdx remove double defined message	31/01/2018	Willem Melching
Add 2020 Acura RDX (#290)	11/10/2020	Chris Souers

Table 73 - Contributions to the DBC files of Acura cars

Commit summary	Date	Contributor
BMW 2008-2013 (#230)	02/04/2020	dzid26

Table 74 - Contributions to the DBC files of BMW cars

Commit summary	Date	Contributor
initial signals for Chrysler Pacifica 2017 hybrid	08/03/2018	Drew Hintz

Commit summary	Date	Contributor
add gear status PRNDL	08/03/2018	Drew Hintz
tighten up speed bits. brake pressure max comment.	08/03/2018	Drew Hintz
value table for gear status	08/03/2018	Drew Hintz
speed of right vs left side of car	08/03/2018	Drew Hintz
units for speed_right	08/03/2018	Drew Hintz
Set packet lengths, adding steering rate, adjusted speed	08/03/2018	Ted Slesinski
Speed, braking, and distance signals	08/03/2018	Ted Slesinski
turn signals	08/03/2018	Drew Hintz
turn signal lights (and thus hazard lights)	08/03/2018	Drew Hintz
high beams for genericToggle	08/03/2018	Drew Hintz
high beams also. likely dashboard message.	08/03/2018	Drew Hintz
acceleration pedal for gasPressed	08/03/2018	Drew Hintz
more Chrysler Pacifica signals (#84)	27/07/2018	Drew Hintz
Add Chrysler ACC cancel button (#127)	11/12/2018	Drew Hintz
Chrysler: seems more correct for torque	29/12/2018	Riccardo Biasini
Chrysler: naming consistency	29/12/2018	Riccardo Biasini
Chrysler: no big endian	29/12/2018	Riccardo Biasini
Chrysler: less big endian in dbc	29/12/2018	Riccardo Biasini
Chrysler: no more big endian	29/12/2018	Riccardo Biasini
added torque driver and torque motor	29/12/2018	Riccardo Biasini
Chrysler: added lkas icon color signal	31/12/2018	Riccardo Biasini
Chrysler: better names and LKAS_HUD message understood	31/12/2018	Riccardo Biasini
Chrysler: minor message renaming	31/12/2018	Riccardo Biasini
Chrysler: fixed torque motor understanding	03/01/2019	Riccardo Biasini
Chrysler message to play an audible beep & ACC cancel (#133)	07/01/2019	Drew Hintz
Chrysler L gear (#139)	31/01/2019	Drew Hintz
Chrysler car model in LKAS message so we can use CAN packer (#140)	21/02/2019	Drew Hintz
change Chrysler radar to all big endian to avoid OP can parser bug (#141)	28/02/2019	Drew Hintz
add Chrysler ACC resume button (#161)	03/05/2019	Drew Hintz
Chrysler: increase size of ACCEL_134 (#174)	23/07/2019	Drew Hintz
Parking Assist Messages (#183)	06/09/2019	TK211X
Chrysler commonize gear VALs	16/02/2020	Riccardo Biasini
Chrysler: add counter to 514	24/02/2020	Riccardo Biasini
Chrysler: Speed msg is 5 bytes	24/02/2020	Riccardo Biasini
reverting changes to Chrysler: speed message seems different from car to car	24/02/2020	Riccardo Biasini
Chrysler: calculate checksum in can packer/parser	30/04/2020	Adeeb Shihadeh
Fix wrong message size in Chrysler	09/05/2020	Adeeb Shihadeh
Pacifica: Add cruise state indicator (#332)	04/01/2021	vanillagorillaa

Table 75 - Contributions to the DBC files of Chrysler cars

Commit summary	Date	Contributor
ford cgea 1.2 2011 (#32)	05/10/2017	jessrussell

Commit summary	Date	Contributor
ford fusion initial dbc file	16/04/2018	Commaremote
Ford Fusion: dbc corrections	16/04/2018	Commaremote
Ford Fusion: added steering stalk buttons	17/04/2018	Commaremote
Ford Fusion: added cruise states	17/04/2018	Commaremote
Ford Fusion: fixed bits for wheel speeds	17/04/2018	Commaremote
Ford Fusion: corrected wheel speed factor	17/04/2018	Riccardo Biasini
Ford Fusion: draft for LKAS message	17/04/2018	Riccardo Biasini
Ford Fusion: more info about LKAS control	18/04/2018	Riccardo Biasini
Ford Fusion: more discoveries on LKAS msg	18/04/2018	Riccardo Biasini
Ford Fusion: added LKAS ui masg	19/04/2018	Commaremote
Ford Fusion: LKAS msg update	19/04/2018	Riccardo Biasini
Ford Fusion: added lkas state fbck	19/04/2018	Commaremote
Ford Fusion: bit 7 is not steering angle	20/04/2018	Riccardo Biasini
Ford Fusion: fixed typo	21/04/2018	Commaremote
Ford Fusion: added radar dbc file	25/04/2018	Riccardo Biasini
Ford Fusion: fixed stere conversion	26/04/2018	Riccardo Biasini
Ford Fusion: added accel pedal pos	27/04/2018	Commaremote
Ford Fusion: adjusted pedal msbw	27/04/2018	Commaremote
Ford Fusion: added VAL and CM regarding Lkas_Action signal	03/05/2018	Riccardo Biasini
Ford Fusion: added brake and doors info	03/05/2018	Commaremote
Ford: Add new base DBC. (#287)	09/09/2020	roxasthenobody98
Create FORD_CADSdbc (#351)	12/03/2021	ReFil

Table 76 - Contributions to the DBC files of Ford cars

Commit summary	Date	Contributor
Create gm_global_a_lowspeed_1818125dbc	04/10/2017	jessrussell
Update gm_global_a_lowspeed_1818125dbc (#34)	10/10/2017	Mutley
GM: reordered msg addrs so they are monotonic	04/05/2018	Riccardo Biasini
GM: typo	13/06/2018	Commaremote
GM: added cruise main on status	14/06/2018	Commaremote
GM: added TC status	14/06/2018	Commaremote
GM: EPB applied bit	15/06/2018	Riccardo Biasini
GM: fixed epb msg addr	15/06/2018	Commaremote
GM: typoin epb msg	15/06/2018	Commaremote
GM: cruise status	15/06/2018	Commaremote
GM: fixed cruise values	15/06/2018	Commaremote
GM: typo fixes	16/06/2018	Riccardo Biasini
GM: copy radar header from cadillac (#116)	15/10/2018	Vasily Tarasov
Add GM signals for ESP/Mode/Highbeams/Intellibeam	30/10/2018	srpape
Add GM FCW Alert (Take 2) (#125)	28/11/2018	Jamezz
GM Object Front Cam Signals (#128)	21/12/2018	Kylan
new GM powertrain signals (#136)	15/01/2019	Kylan
GM: use common gear VALs	16/02/2020	Riccardo Biasini
Fix GM message signal sizes	09/05/2020	Adeeb Shihadeh
Fix non-standard units in GM global A lowspeed (#327)	20/12/2020	Ryan Rowe

Commit summary	Date	Contributor
GM steering rate scale fix (#362)	12/03/2021	qadmus

Table 77 - Contributions to the DBC files of GM cars

Commit summary	Date	Contributor
Volt gen 2 support	04/07/2017	Vasily Tarasov
Fix Chevy Volt actuator signal parsing (#66)	16/01/2018	Vasily Tarasov
Volt's gas pedal only and combined gas/acc (#76)	20/02/2018	Vasily Tarasov
Volt: switch to parsing ACC buttons from powertrain CAN (#74)	20/02/2018	Vasily Tarasov
Volt doors and belts status (#70)	26/02/2018	Vasily Tarasov
Chevy Volt tweaks (#83)	16/03/2018	Vasily Tarasov
Added High Voltage Management to powertrain file.	03/02/2020	Sean Murphy
Removed non high voltage items. Added units.	03/02/2020	Sean Murphy
Fixed up cell voltage readings and added more commands (#220)	27/05/2020	streber42
Updated Chevrolet Volt HV management messages (#345)	23/06/2021	streber42

Table 78 - Contributions to the DBC files of Chevrolet cars

Commit summary	Date	Contributor
added Cadillac dbc, starting from Volt	15/05/2018	Riccardo Biasini
Cadillac CT6: added LKAS cmd msg. Thanks mutley	17/05/2018	Commaremove
Cadillac CT6: fixed LKAS msg	17/05/2018	Commaremove
Cadillac: forgot to update old references to msg 384	17/05/2018	Commaremove
Cadillac: updated vehicle speed factor	17/05/2018	Commaremove
Cadillac: added ASCM-b lkas control msg	17/05/2018	Commaremove
Cadillac: lkas mode added	17/05/2018	Riccardo Biasini
some adjustments in Cadillac dbc	18/05/2018	Riccardo Biasini
checksum seems only 10 bits	17/05/2018	Commaremove
Cadillac: lkas clarification	18/05/2018	Riccardo Biasini
Cadillac: add object bus dbc	24/05/2018	Commaremove
Cadillac: fixed lkas status msg	25/05/2018	Commaremove
Cadillac: fixed eps messages	30/05/2018	Commaremove
Cadillac: fixed dbc VAL	30/05/2018	Riccardo Biasini
Cadillac: added chassis dbc, for now simple copy from gm	31/05/2018	Riccardo Biasini
Cadillac: add lkas cmd to chassis bus as well	31/05/2018	Riccardo Biasini
Cadillac: fixed counter size	01/06/2018	Commaremove
Cadillac: fixed lkas torque delivered	01/06/2018	Riccardo Biasini
Cadillac: few things added to EPS status	01/06/2018	Riccardo Biasini
Cadillac: adjusted gas command	04/06/2018	Riccardo Biasini
Cadillac: bug fix in redundant steer command msg	09/06/2018	Riccardo Biasini
Cadillac: typo	13/06/2018	Commaremove

Table 79 - Contributions to the DBC files of Cadillac cars

Commit summary	Date	Contributor
Adds 2016 CR-V dbc	09/06/2017	Ted Slesinski
Merge pull request #1 from energie/crv	09/06/2017	George Hotz
push touring mod	26/06/2017	George Hotz
Syntax Error in accord dbc, no 0xE4 message	10/07/2017	Ted Slesinski
tweak crv format	26/07/2017	espes
Honda: changing 'BRAKE_LIGHTS_ON' into 'BRAKE_SWITCH', which is more appropriate and not confused with 'BRAKE_LIGHTS'	05/08/2017	Riccardo Biasini
Several ADAS updates to CR-V dbc	31/08/2017	Ted Slesinski
Renaming 17 crv dbc to include trim	31/08/2017	Ted Slesinski
Changing units, max, and factor on second transmission signal to match honda civic	31/08/2017	Ted Slesinski
Latest dbc updates, including chffr metric for engine temp	12/09/2017	Ted Slesinski
Adds Chinese Odyssey DBC (#110)	17/09/2018	Ted Slesinski
Honda/Acura: added acc speed offset to dbc files	21/09/2017	Riccardo Biasini
Honda/Acura: added radar status msg to detect radar malfunctions	21/09/2017	Riccardo Biasini
Honda/Acura: added name to msg 0x37c (892)	23/09/2017	Riccardo Biasini
Civic: added brake hold state to dbc	30/09/2017	Riccardo Biasini
Civic: clarified brake hold signals	30/09/2017	Riccardo Biasini
Added trunk open signal	01/10/2017	vanillagorillaa
Update honda_civic_touring_2016_candbc	01/10/2017	vanillagorillaa
Improved seat belt status (#27)	01/10/2017	vanillagorillaa
Add Odometer on 0x516 (#25)	01/10/2017	vanillagorillaa
Added counter and checksum to all defined messages (#29)	03/10/2017	vanillagorillaa
Civic: readded odometer and corrected a bug introduced by cabana	03/10/2017	Riccardo Biasini
More updates to support work on 2017 CR-V port (#33)	05/10/2017	Ted Slesinski
Civic Hatchback DBC (#35)	08/10/2017	Ted Slesinski
Door Locked/Unlocked on 0x309 (#36)	08/10/2017	vanillagorillaa
Reverse Lights on 0x326 (#37)	08/10/2017	vanillagorillaa
ECON Mode On 0x221 (#39)	08/10/2017	vanillagorillaa
Added gear 2 (#40)	10/10/2017	vanillagorillaa
Removed wrong gear signal (#42)	17/10/2017	vanillagorillaa
Cleanup and corrections to the accord DBC	14/10/2017	Ted Slesinski
Wipers (#43)	23/10/2017	vanillagorillaa
Headlight status (#46)	23/10/2017	vanillagorillaa
Passenger Airbag (#44)	23/10/2017	vanillagorillaa
Honda: fixed gas interceptor offset bug	24/10/2017	Riccardo Biasini
Add stalk definitions and checksum fixes (#51)	12/11/2017	Chris Souers
Honda Odyssey dbc (#52)	23/11/2017	Ted Slesinski
Honda Pilot Touring 2017 (#58)	18/12/2017	vanillagorillaa
2017 CR-V Update (#60)	20/12/2017	Ted Slesinski
Added EPB and Brake_Hold (#61)	21/12/2017	Joel Jacobs
Add Honda Clarity Hybrid (#65)	05/01/2018	vanillagorillaa

Commit summary	Date	Contributor
Cleanup duplicate message names	24/01/2018	Willem Melching
consistent can message names for supported Hondas	25/01/2018	Willem Melching
add odometer message to civic and odyssey	27/01/2018	Willem Melching
fix honda dometer scaling	27/01/2018	Willem Melching
readded gas interceptor to Hondas so it does not break regression tests	27/01/2018	Riccardo Biasini
fix crv steering_control message	27/01/2018	Willem Melching
rename generic honda and Toyota dbcs to include year	27/01/2018	Willem Melching
honda wheelspeed in kph to match Toyota	28/01/2018	Willem Melching
fix bugs in some counter/checksum definitions	29/01/2018	Willem Melching
move pilot touring to generated	29/01/2018	Willem Melching
fixed inconsistent factor for speed in Honda dbc files	13/02/2018	Riccardo Biasini
Add 2017 Honda Ridgeline (#77)	20/02/2018	Ted Slesinski
fix honda pcm gas message size	23/02/2018	Willem Melching
add set me to lkas hud honda	23/02/2018	Willem Melching
add interceptor to civic	09/03/2018	Willem Melching
add setme to honda ACC_HUD	11/03/2018	Willem Melching
Updating Bosch dbcs to use new format and bringing in new honda changes (#82)	18/03/2018	Ted Slesinski
Add 2019 CR-V Hybrid DBC (#148)	22/03/2019	Ted Slesinski
add back import file for honda pedal's (#94)	19/05/2018	vanillagorillaa
Honda: name change to make the brake pump request bit explicit	21/06/2018	Riccardo Biasini
Honda Accord: does not have wheels moving bit	07/07/2018	Riccardo Biasini
Add 2018 Honda Fit EX F-CAN dbc (#100)	17/07/2018	Ted Slesinski
Honda Nidec: added wrong config radar value	24/07/2018	Riccardo Biasini
Toyota: added chr hybrid. Honda: regenerated fit.	29/07/2018	Riccardo Biasini
Adds 1.5L Accord DBC (#107)	14/08/2018	Ted Slesinski
Honda-Bosch: fixed xmission speed unit	17/08/2018	Riccardo Biasini
Honda: forgot to generate dbc files	17/08/2018	Riccardo Biasini
add vals honda (#121)	04/11/2018	dekerr
Adds dbc for 2019 Honda Insight (#122)	19/11/2018	Ted Slesinski
Honda Nidec: VSA_STATUS msg is the same for all	07/02/2019	Riccardo Biasini
Honda: for simplicity all cars now have BRAKE_HOLD signal	07/02/2019	Riccardo Biasini
Honda: added signal with imperial unit bit	31/05/2019	Commaremove
Honda: fix bug due to little endianness	01/06/2019	Riccardo Biasini
Honda: added time gap setting signal	01/06/2019	Riccardo Biasini
Civic: HUD_SETTING is 5 bytes	01/06/2019	Riccardo Biasini
Honda: fix China model. Toyota: add STEERING_LTA message for nodsu cars	21/06/2019	Riccardo Biasini
Reverse engineer AEB in Honda	03/08/2019	Riccardo Biasini
Forgot to run generator	03/08/2019	Riccardo Biasini
Add DBC for JDM Honda Fit Hybrid 2018 (#178)	23/08/2019	Pramuditha Aravinda

Commit summary	Date	Contributor
Bosch AEB signals	23/08/2019	Adeeb Shihadeh
more honda Bosch AEB signals	23/08/2019	Adeeb Shihadeh
use generator for honda odyssey extreme	28/08/2019	Adeeb Shihadeh
generated odyssey extreme	28/08/2019	Adeeb Shihadeh
honda nidec AEB values	30/08/2019	Adeeb Shihadeh
honda Bosch longitudinal (#185)	08/09/2019	Greg Hogan
Fixed brake signal unit in Bosch Honda	10/09/2019	Riccardo Biasini
Honda: correct steering torque sensor sign to be consistent with standard convention (left+)	14/11/2019	Riccardo Biasini
Honda FCM: diagnostic signals	12/12/2019	Riccardo Biasini
Honda Nidec: add new ACC_HUD signals to all other cars other than the CIVIC	16/12/2019	Riccardo Biasini
update honda steering signals (#208)	20/12/2019	Greg Hogan
Fix honda dbc files after steer torque addition	20/12/2019	Riccardo Biasini
One more fix	20/12/2019	Riccardo Biasini
Adds 2016 Honda CR-V Executive	16/01/2020	Ted Slesinski
Add support for 2019 Civic Sedan Diesel. Split GAS_PEDAL_2 out to each car. (#215)	13/02/2020	Chris Souers
Add Honda-Bosch lane line detection signals. (#223)	03/03/2020	DisgracedPilot
fix: Replicate changes done on _honda_2017dbc (#225)	04/03/2020	Riccardo Biasini
Bosch: Adding minor changes to be compatible with 0.4.3.2+	11/04/2018	Ted Slesinski
Civic: added speed Unit	16/04/2019	Riccardo Biasini
Create DBC for HRV (#248)	20/04/2020	cowanhmoore
Add values for a static 0xe5 (honda Bosch) (#250)	28/04/2020	Chris Souers
Rename BYTE_ to SET_ME_X (#253)	29/04/2020	Chris Souers
Honda BSM body (#286)	21/07/2020	Greg Hogan
Use generator for Honda Clarity DBC (#289)	24/07/2020	Adeeb Shihadeh
Honda - steer down to zero (#317)	20/12/2020	Greg Hogan
new clarity brake_error bits (#367)	20/03/2021	vanillagorillaa
Honda HUD message (#371)	30/03/2021	vanillagorillaa
HRV correct GAS_PEDAL (#266)	02/06/2020	cowanhmoore
CIVIC_BOSCH needs an empty nidec brake command frame for long control. (#246)	05/10/2020	Chris Souers
Merge Accord DBCs (#400)	03/06/2021	Shane Smiskol
Fix steering rate signs for Hondas (#404)	23/06/2021	sshane
Honda Bosch: Add new LKAS HUG messages for 2021+ models (#372)	07/07/2021	Chris Souers

Table 80 - Contributions to the DBC files of Honda cars

Commit summary	Data	Contributor
add Hyundai 2015 (#63)	28/12/2017	jessrussell
add hyundai_i30_2014dbc	10/09/2017	Jess
This adds support for 8 Speed Auto Transmission (#104)	28/07/2018	Andrew Frahn
Correct Message ID on LKAS11 (#172)	01/08/2019	TK211X

Commit summary	Date	Contributor
Hyundai Santa Fe: first dbc commit	22/08/2018	Commaremote
added gear to dbc for Hyundai	22/08/2018	Commaremote
Santa Fe: for now unitless torque request	22/08/2018	Commaremote
Santa Fe: this signal seems 2 bits long	22/08/2018	Riccardo Biasini
Santa Fe: added lane icon color to dbc	22/08/2018	Riccardo Biasini
Santa Fe: how come the steer angle sign was wrong	25/08/2018	Commaremote
Santa Fe: dealing with steer torque integer is easier for now	25/08/2018	Commaremote
Hyundai: not sure why steer angle was unsigned... seems a bug	30/08/2018	Riccardo Biasini
Hyundai Cleanup (#130)	24/12/2018	Andrew Frahn
Update DBC for Hyundai Kona Support (#138)	29/01/2019	Andrew Frahn
Add FCA11 & SCC14 (#184)	06/09/2019	TK211X
add electrical gear and fix driver torque	19/01/2020	xx979xx
correct max value	19/01/2020	xx979xx
new Hyundai dbc	08/04/2020	George Hotz
Remove non ascii characters	08/04/2020	Willem Melching
Add LFAHDA message to Hyundai	16/04/2020	Willem Melching
Add gas/brake message for Hyundai EVs, from @TK211X	22/05/2020	Adeeb Shihadeh
fix endianness in signal from new Hyundai message	22/05/2020	Adeeb Shihadeh
Add DAW (#175)	27/05/2020	TK211X
Update SCC ECU Messages for OP Long Dev. (#267)	28/05/2020	TK211X
Hyundai: AEB and FCW signals	24/06/2020	Adeeb Shihadeh
Hyundai: update scc14 (#274)	27/07/2020	Alice Knag
Update hyundai_kia_genericdbc (#284)	03/08/2020	xps-genesis
hyundai esp12 checksum and counter were flipped (#320)	03/12/2020	Greg Hogan
fix hyundai 366_EMS (#319)	20/12/2020	Greg Hogan
fix LKAS12 CF_Lkas_Daw_USM (#318)	20/12/2020	Greg Hogan
hyundai: update LFAHDA_MFC for HDA (#338)	06/01/2021	Greg Hogan
hyundai: add P_STS counter and checksum (#333)	10/01/2021	Greg Hogan
hyundai: fix scc14 comfort band/jerk (#337)	12/01/2021	Greg Hogan
hyundai: better hda signal def (#342)	12/01/2021	Greg Hogan
Fix errors when using candools with hyundai_kia_genericdbc (#346)	31/01/2021	gsa88
Hyundai: add gas pedal position for hybrids (#401)	10/06/2021	Shane Smiskol
Define Hyundai gears in dbc (#405)	29/06/2021	Shane Smiskol

Table 81 - Contributions to the DBC files of Hyundai cars

Commit summary	Date	Contributor
Create luxgen_s5_2014dbc (#101)	16/10/2019	chinlin
fixed to luxgen dbc file	16/10/2019	Riccardo Biasini

Table 82 - Contributions to the DBC files of Luxgen cars

Commit summary	Date	Contributor
Add Mazda CX-5 2017 GT	05/03/2019	Jafar Al-Ghraibeh

Commit summary	Date	Contributor
Updates and new signals (#177)	22/08/2019	Jafar Al-Gharaibeh
Fix torque signal and add new CAN msgs (#181)	08/09/2019	Jafar Al-Gharaibeh
Speed Auto High Beam Traffic signs	28/03/2029	Jafar Al-Gharaibeh
traffic sign speed limit	28/03/2020	Jafar Al-Gharaibeh
Introduce the new Mazda 3 2019/2020 dbc	28/03/2020	Jafar Al-Gharaibeh
Tracking the steer angle with LKAS signal	28/03/2020	Jafar Al-Gharaibeh
Pedals/gear, gas pedal scale value	28/03/2020	Jafar Al-Gharaibeh
Speed limit signs	28/03/2020	Jafar Al-Gharaibeh
Rear Cross Traffic Alert	28/03/2020	Jafar Al-Gharaibeh
Raw angle signal data for easy checksum calc, and one less gear bit (#254)	01/05/2020	Jafar Al-Gharaibeh
Better GEAR signal tracking the gear stick rather than the gear box (#257)	03/05/2020	Jafar Al-Gharaibeh
Mazda: add missing static bits, tidy up endianness (#263)	27/05/2020	Jafar Al-Gharaibeh
Mazda: Traffic sign bits (#392)	10/05/2021	Jafar Al-Gharaibeh

Table 83 - Contributions to the DBC files of Mazda cars

Commit summary	Date	Contributor
create Mercedes e350 (2010) dbc (#47)	21/12/2017	quillford
Update Mercedes e350 dbc (#112)	17/09/2018	quillford

Table 84 - Contributions to the DBC files of Mercedes cars

Commit summary	Date	Contributor
Nissan: Added nissan_2017dbc (#173)	18/07/2019	Bugsy
Cleanup of Nissan DBC (#218)	25/02/2020	Andre Volmensky
Added messages. Cleaned up endianness (#226)	04/03/2020	Riccardo Biasini
Added ProPilot HUD messages (#231)	11/03/2020	Andre Volmensky
Fixed up sign on Driver Torque, added unit (#233)	12/03/2020	Andre Volmensky
Nissan x trail cleanup (#237)	25/03/2020	Willem Melching
Nissan leaf (#238)	28/03/2020	Willem Melching
Fixed signal unknown1 overlapping the button bits	29/03/2020	Andre Volmensky
Fix wrong message sizes in Nissan	09/05/2020	Adeeb Shihadeh
Update X-trail HUD message name, added SPEED MPH signal (#269)	01/06/2020	Andre Volmensky

Table 85 - Contributions to the DBC files of Nissan cars

Commit summary	Date	Contributor
Create subaru_outback_2016_eyesightdbc	12/07/2017	Jeff Palmer
OpenDBC updates	27/07/2017	Jeff Palmer
OpenDBC updates	27/07/2017	Jeff Palmer
Back	09/08/2017	Jeff Palmer
OpenDBC updates	09/08/2017	Jeff Palmer
OpenDBC updates	10/08/2017	Jeff Palmer

Commit summary	Date	Contributor
OpenDBC updates	10/08/2017	Jeff Palmer
Update subaru_outback_2016_eyesightdbc	10/08/2017	Jeff Palmer
Update subaru_outback_2016_eyesightdbc	10/08/2017	Jeff Palmer
Update subaru_outback_2016_eyesightdbc	10/08/2017	Jeff Palmer
Update subaru_outback_2016_eyesightdbc	10/08/2017	Jeff Palmer
Update subaru_outback_2016_eyesightdbc	10/08/2017	Jeff Palmer
Update subaru_outback_2016_eyesightdbc	10/08/2017	Jeff Palmer
OpenDBC updates	10/08/2017	Jeff Palmer
Update subaru_outback_2016_eyesightdbc	10/08/2017	Jeff Palmer
Update subaru_outback_2016_eyesightdbc	10/08/2017	Jeff Palmer
Update subaru_outback_2016_eyesightdbc	10/08/2017	Jeff Palmer
Update subaru_outback_2016_eyesightdbc	10/08/2017	Jeff Palmer
Update subaru_outback_2016_eyesightdbc	11/08/2017	Jeff Palmer
OpenDBC updates	11/08/2017	Jeff Palmer
Update subaru_outback_2016_eyesightdbc	11/08/2017	Jeff Palmer
Update subaru_outback_2016_eyesightdbc	11/08/2017	Jeff Palmer
Fixed a typo	13/08/2017	Jeff Palmer
Create subaru_outback_2015_eyesightdbc (#137)	27/01/2019	Bugsy
Subaru: added global dbc	01/03/2019	Riccardo Biasini
Subaru global dbc (#142)	01/03/2019	Bugsy
SUBARU LKAS: minus sign to steer command to match standard convention	07/03/2019	Riccardo Biasini
Subaru: left steer is positive	07/03/2019	Riccardo Biasini
fixed sign in steering angle	09/03/2019	Vehicle Researcher
Subaru Global: simplified Stalk Message	11/03/2019	Riccardo Biasini
Subaru Global: more endianness consistency. Still a long way to go	11/03/2019	Riccardo Biasini
Subaru: endianness consistency in wheel speeds	11/03/2019	Riccardo Biasini
Subaru: some cleanup to dbc	11/03/2019	Riccardo Biasini
Subaru: fixed DOOR_OPEN sgs	12/03/2019	Riccardo Biasini
Subaru: slightly touched wheel speed factor	15/03/2019	Riccardo Biasini
Subaru: update LKAS_State	20/03/2019	Vehicle Researcher
Subaru: set speed can be in kph and it needs 8 bits	22/03/2019	Vehicle Researcher
Subaru: added cruise buttons	14/04/2019	Riccardo Biasini
Subaru: minor pedal gas conversion fix	14/04/2019	Riccardo Biasini
Subaru: filled Cruise Buttons message	15/04/2019	Riccardo Biasini
Subaru: temporarily simplified msg 545 for dev reasons. Removed signals will be restored	15/04/2019	Riccardo Biasini
Subaru: filling ES_LKAS message	19/04/2019	Riccardo Biasini
Subaru: removed unknown signals from ES_LKAS_State	19/04/2019	Riccardo Biasini
Update subaru_outback_2015_eyesightdbc (#163)	09/05/2019	Bugsy
Subaru: added hud unit selection	12/05/2019	Riccardo Biasini
Subaru: added lane line visibility to ES_LKAS message	12/05/2019	Riccardo Biasini
Update metric value for Dash_Units (#164)	15/05/2019	martinl
Add Subaru global transmission msg with gear values (#168)	23/08/2019	martinl

Commit summary	Date	Contributor
remove obsolete Subaru dbc file	30/10/2019	Riccardo Biasini
Update subaru_outback_2015_eyesightdbc (#195)	07/11/2019	Bugsy
Fix outback endianness consistency (#196)	07/11/2019	Bugsy
Add steering error signal in Subaru global	03/02/2020	Bugsy
Update subaru_global_2017dbc	03/02/2020	Bugsy
Subaru: added counter and checksum to brake msg	07/03/2020	Riccardo Biasini
Fixed signals order and added new signals for Subaru global (#221)	06/04/2020	martinl
CANPacker: Subaru checksum support (#241)	14/04/2020	Adeeb
add checksum check to can parser for Subaru	30/04/2020	Adeeb Shihadeh
Subaru preglobal update (#260)	21/05/2020	martinl
Add BSD_RCTA to Subaru Global (#244)	24/05/2020	martinl
Subaru DBC update (#242)	28/05/2020	martinl
Subaru preglobal DBC update (#270)	11/06/2020	martinl
Subaru Outback 2019 (#278)	29/06/2020	martinl
Subaru DBC update (#277)	23/07/2020	martinl
Subaru: Pre-global signals unification + new messages (#395)	19/05/2021	martinl

Table 86 - Contributions to the DBC files of Subaru cars

Commit summary	Date	Contributor
Import the DBCs from Openpilot	31/05/2017	George Hotz
Lots of correction, thanks to cabana!	03/11/2017	jeankalud
Corrected MCU_clusterBacklightRequest (typo)	04/11/2017	jeankalud
Tesla: Add missing line break after VAL_ 69 WprSw6Posn (#109)	02/09/2018	Oscar Söderlund
DBC for the Bosch Radar for Tesla (#158)	02/05/2019	BogGyver
Update Tesla DBCs (#348)	19/02/2021	robbederks
Tesla driver braking (#360)	10/03/2021	robbederks
add tesla AP status (#365)	14/03/2021	robbederks

Table 87 - Contributions to the DBC files of Tesla cars

Commit summary	Date	Contributor
OpenDBC updates	27/07/2017	George Hotz
OpenDBC updates	27/07/2017	George Hotz
OpenDBC updates	27/07/2017	George Hotz
OpenDBC updates	27/07/2017	George Hotz
OpenDBC updates	27/07/2017	George Hotz
OpenDBC updates	27/07/2017	George Hotz
OpenDBC updates	27/07/2017	George Hotz
OpenDBC updates	27/07/2017	George Hotz
OpenDBC updates	27/07/2017	George Hotz
OpenDBC updates	27/07/2017	George Hotz

Commit summary	Date	Contributor
TOYOTA PRIUS: added pedal signals and created a new dbc file for the ADAS can bus with initial radar data	22/08/2017	Riccardo Biasini
Prius reverse engineering (#14)	24/08/2017	Riccardo Biasini
Toyota dbc (#15)	26/08/2017	Riccardo Biasini
Prius: added NEW_TRACK bit to radar's data	26/08/2017	Riccardo Biasini
adjusted factors in Toyota radar's dbc	26/08/2017	Riccardo Biasini
Toyota Prius: added driver steer torque to dbc (#16)	29/08/2017	Riccardo Biasini
Toyota Prius: fixed rel speed factor in radar tracks	29/08/2017	Riccardo Biasini
Toyota Prius: fixed STEER_FRACTION unit	30/08/2017	Riccardo Biasini
Toyota Prius: added part of the hud control	30/08/2017	Riccardo Biasini
Toyota Prius: correcting radar lateral distance factor	30/08/2017	Riccardo Biasini
Prius: added main acc switch to dbc	01/09/2017	Riccardo Biasini
Prius: added long acceleration command	13/09/2017	Riccardo Biasini
OpenDBC updates	13/09/2017	Riccardo Biasini
Prius: adjusted bit selection for lead distance and added LEAD_INFO msg	13/09/2017	Riccardo Biasini
Prius: fixed typo	13/09/2017	Riccardo Biasini
Prius: adjusted vehicle speed factor	13/09/2017	Riccardo Biasini
Prius: wheel speeds reordered	13/09/2017	Riccardo Biasini
Prius: added IPAS steering info	20/09/2017	Riccardo Biasini
OpenDBC updates	28/09/2017	Riccardo Biasini
OpenDBC updates	28/09/2017	Riccardo Biasini
Prius: minor signal name changes	28/09/2017	Riccardo Biasini
Prius: added TC disabled signal	29/09/2017	Riccardo Biasini
Prius: added gear vals and info about cruise state machine	29/09/2017	Riccardo Biasini
Prius: adding radar's info	03/10/2017	Riccardo Biasini
Prius: typo fix in radar dbc	03/10/2017	Riccardo Biasini
Prius: tuned radar longitudinal distance	03/10/2017	Riccardo Biasini
Prius: added gas released signal	04/10/2017	Riccardo Biasini
Prius: added radar validity bit	06/10/2017	Riccardo Biasini
Toyota: changed dbc names	10/10/2017	George Hotz
Toyota rav4: added dbc draft	10/10/2017	George Hotz
Toyota: fixed Rav4 VS Prius differences	11/10/2017	George Hotz
Toyota: increased factor for steer torque sensor	12/10/2017	Riccardo Biasini
Toyota: added fault indication for LKA	12/10/2017	Riccardo Biasini
Toyota: changed LKA state signal name	12/10/2017	Riccardo Biasini
Toyota: added UI set speed to dbc	18/10/2017	Riccardo Biasini
Toyota: added UI setting msg	18/10/2017	George Hotz
Toyota: added accel produced by cruise control	26/10/2017	Riccardo Biasini
Toyota: fixed steer motor offset	27/10/2017	Riccardo Biasini
Toyota: added checksums	29/10/2017	Riccardo Biasini
Toyota: bug fix in checksum bits	29/10/2017	Riccardo Biasini
Toyota: fixed 610 msg length error	30/10/2017	Riccardo Biasini
Toyota: added low speed lockout bit	30/10/2017	Riccardo Biasini
Toyota: added checksum for 467 and 467 msgs	30/10/2017	Riccardo Biasini
Toyota Prius: msg 610 is 8 bytes long	31/10/2017	Riccardo Biasini

Commit summary	Date	Contributor
Created Toyota iQ base plus reversed some signals (#48)	05/11/2017	alessbelli
Toyota: added standstill on signal	11/11/2017	Riccardo Biasini
Toyota: added brake lights when ACC commands decel	19/11/2017	Vehicle Researcher
rav4: fixed comments to BRAKE signals	19/11/2017	Riccardo Biasini
Toyota: added fcw	21/11/2017	Riccardo Biasini
Toyota rav4 hybrid: OP works, but some signals need to be verified	22/12/2017	Riccardo Biasini
Toyota: added auto high beam	12/01/2018	Carlaptop01
Toyota Corolla: added dbc file, seems the same as Rav4	25/01/2018	Riccardo Biasini
Toyota: different factor for STEER_TORQUE_EPS	27/01/2018	Riccardo Biasini
fix gas pedal message length rav4h	01/02/2018	Willem Melching
fix Toyota steering lka message length	01/02/2018	Willem Melching
Toyota: more vals for LKA_STATE	13/02/2018	Riccardo Biasini
Toyota: re-generated the files after cfbc9ae363f98ef	13/02/2018	Riccardo Biasini
Add 2018 Camry Hybrid DBC's (#73)	16/02/2018	vanillagorillaa
Add 2018 Toyota CHR dbc (#78)	22/02/2018	Ted Slesinski
update Toyota ACC_CONTROL fields	05/03/2018	Willem Melching
add set me fields to Toyota ACC_HUD	05/03/2018	Willem Melching
add set me to Toyota LKAS_HUD	05/03/2018	Willem Melching
Toyota IPAS: proper steer angle unit	09/03/2018	Riccardo Biasini
run generator for ipas scaling	09/03/2018	Willem Melching
add setme to Toyota STEERING_IPAS	09/03/2018	Willem Melching
Toyota missing ACC_CONTROL checksum	13/03/2018	Willem Melching
extra setme field Toyota LKAS_HUD	13/03/2018	Willem Melching
Toyota: change signal name in EPS_STATUS msg	14/03/2018	Commaremove
Fix Checksum errors for CH-R (#86)	18/03/2018	vanillagorillaa
Toyota: fixed LKA_STATE to be compatible with Corolla and properly generated CH-R dbc	27/03/2018	Riccardo Biasini
Toyota: added comma specific message that copies 0x266 to be able to control steer angle even if park assist ecu is plugged in	11/04/2018	Riccardo Biasini
Toyota Prius: added AUTOPARK_STATUS msg	11/04/2018	Riccardo Biasini
Toyota: forgot to add _commadbc	05/05/2018	Riccardo Biasini
Toyota Highlander and Avalon DBC (#93)	19/05/2018	vanillagorillaa
Toyota Pedal Support (#108)	24/08/2018	wocsor
Added Toyota Highlander Hybrid	02/09/2018	Riccardo Biasini
Toyota Highlander: fixed dbc file name	04/09/2018	Riccardo Biasini
Add Toyota radar SCORE field	17/10/2018	Willem Melching
add toyota_prius_2010_ptdbc (#50)	30/10/2018	ROBINSON MAS
Add Distance Lines and RSA (#118)	31/10/2018	arne182
Toyota: generated dnc files after latest change	31/10/2018	Riccardo Biasini
Toyota: added a better cruise active indicator	04/11/2018	Riccardo Biasini
Toyota: fixed typos	04/11/2018	Riccardo Biasini
Add 3rd RSA signal and cleanup (#120)	08/11/2018	arne182
Add more Sign recognitions (#126)	07/12/2018	arne182

Commit summary	Date	Contributor
Add DSU Speed (#134)	31/12/2018	arne182
Run generator again and added Toyota Sienna	10/01/2019	Riccardo Biasini
Toyota: added more info to long control message	11/01/2019	Riccardo Biasini
Toyota: clarified lane line VALs	15/01/2019	Riccardo Biasini
Added support to Toyota pedal	12/02/2019	Riccardo Biasini
Toyota pedal: added counter	01/03/2019	Riccardo Biasini
Toyota pedal: messages are now 7 bytes	02/03/2019	Commaremote
New camry steer message (#149)	26/03/2019	Bugsy
Add wet road symbol for RSA (#156)	08/04/2019	arne182
Added SPORT_ON message for Corolla (#155)	08/04/2019	Shane Smiskol
Rav4 2019 ADAS (#160)	30/04/2019	wocsor
Toyota ipas msgs: fix repeated signal name	03/06/2019	Riccardo Biasini
Toyota Camry: using the same conversion factor for STEER_TORQUE_EPS as in the CHR	07/06/2019	Riccardo Biasini
Toyota DSU-less: added better measurement of steer angle	10/06/2019	Riccardo Biasini
Toyota: better name for adas bdc files	10/06/2019	Riccardo Biasini
Toyota: better pt dbc file naming for all dsuless cars	10/06/2019	Riccardo Biasini
Toyota dsu-less: more precise steering angle conversion	13/06/2019	Riccardo Biasini
Toyota LTA: back to unit factor	21/06/2019	Riccardo Biasini
Toyota: STEERING_LTA actually has an angle interface	21/06/2019	Riccardo Biasini
Toyota: added 0x283 message description for PRE_COLLISION msg. Data from https://ioactive.com/pdfs/IOActive_Adventures_in_Automotive_Networks_and_Control_Units.pdf	30/07/2019	Riccardo Biasini
typo	30/07/2019	Riccardo Biasini
add another Toyota cancel request signal	01/08/2019	Adeeb Shihadeh
pre-collision signals	02/08/2019	Adeeb Shihadeh
Toyota pre-collision signals	02/08/2019	Vehicle Researcher
reference Toyota DBC	03/08/2019	Adeeb Shihadeh
fix typo	02/08/2019	Vehicle Researcher
Toyota pre collision	03/08/2019	Adeeb Shihadeh
rename	03/08/2019	Adeeb Shihadeh
VIN signal for Toyota	04/08/2019	Adeeb Shihadeh
better VIN msg name	05/08/2019	Adeeb Shihadeh
Toyota DSU cruise message	06/08/2019	Adeeb Shihadeh
restore original Toyota ref	28/08/2019	Adeeb Shihadeh
Toyota time signal (#187)	15/09/2019	quillford
Toyota no dsu: fix steer angle factor, it's 1% of a rad	19/09/2019	Riccardo Biasini
add units and a couple new signals for Toyota (#188)	24/09/2019	quillford
2019+ New Prius Steer Angle (#189)	24/09/2019	illumiN8i
Fix steer angle factor for Toyota	11/10/2019	Riccardo Biasini
Toyota Blind Spot Monitor (TSS2-only?) (#219)	21/02/2020	Nelson Chen
Add STEER_ANGLE to all STEER_TORQUE_SENSOR messages (#228)	06/03/2020	Willem Melching
Add RPM signal (#216)	02/04/2020	Arne Schwarck

Commit summary	Date	Contributor
Fix Toyota message size	11/05/2020	Adeeb Shihadeh
fix lta message (#262)	14/05/2020	Willem Melching
Toyota ACC_CONTROL PERMIT BRAKING and ACCEL_CMT_ALT speculated definition (#258)	28/05/2020	Nelson Chen
Tune 17 Corolla safetyParam (#298)	16/09/2020	Shane Smiskol
Date and Time (#306)	17/10/2020	TK211X
Added CRUISE_STATE value 11 description (timer_3sec) (#328)	24/12/2020	Igor
Add Toyota Odometer Reading to Toyota DBCs (#331)	04/01/2021	Nelson Chen
add more Toyota CRUISE_STATE values (#339)	08/01/2021	Willem Melching
add Toyota headlight signals (#373)	02/04/2021	cydia2020
Toyota hazard light (#375)	05/04/2021	cydia2020
move LTA to common DBC (#386)	25/04/2021	Adeeb Shihadeh
More TSS2 RSA symbols (#364)	25/04/2021	Kumar
Toyota combination meter dimmer signal (#398)	29/05/2021	cydia2020
Toyota light sensor (#393)	23/06/2021	cydia2020
Toyota: define ACC_TYPE signal (#409)	09/07/2021	sshane

Table 88 - Contributions to the DBC files of Toyota cars

Commit summary	Date	Contributor
Lexus: added first draft of RX dbc file	02/02/2018	Riccardo Biasini
Lexus: add is	04/11/2018	Riccardo Biasini
Adds dbc for 2017 lexus is300h (hybrid) (#146)	12/03/2019	eFini
Lexus IS: generated file was out of synch	05/04/2019	Riccardo Biasini
Add wet road symbol for RSA (#156)	08/04/2019	arne182
Lexus GS300h 2017 DBC (#159)	02/05/2019	wocsor
Lexus RX 350 DBC (#170)	12/07/2019	wocsor
properly generate Lexus 350 dbc file	12/07/2019	Riccardo Biasini
Add Lexus CT200h 2018 (#176)	03/08/2019	Thomas Pichard
Lexus CT200 needed one more run of generator	03/08/2019	Riccardo Biasini
merge lexus IS and SI hybrid	28/08/2019	Adeeb Shihadeh
Lexus CT200H seems to have the safetyParam 1 instead of 0.73	17/09/2019	Riccardo Biasini
Lexus is adjustment (#192)	11/10/2019	eFini
Add Lexus NX300H (#214)	04/02/2020	Patipat Susumpow
Lexus CTH fix: brake pressed is on bit 5 like corolla and rav4	05/03/2020	Riccardo Biasini
Add STEER_ANGLE to all STEER_TORUQE_SENSOR messages (#228)	06/03/2020	Willem Melching
Lexus NX300 (#313)	28/10/2020	Adeeb Shihadeh

Table 89 - Contributions to the DBC files of Lexus cars

Commit summary	Date	Contributor
Volvo DBC files for C1MCA and EUCD platform (#291)	29/09/2020	danielzmod

Table 90 - Contributions to the DBC files of Volvo cars

Commit summary	Date	Contributor
Create vw_golf_mk4dbc	22/09/2017	jessrussell
Create vw_mqb_2010dbc	01/10/2017	jessrussell
Volkswagen MQB platform DBC updates (#167)	29/05/2019	Jason Young
Updates for Volkswagen support (#191)	10/10/2019	Jason Young
Fixes and new message for VW MQB, fix for Accord Touring (#193)	17/10/2019	Jason Young
Added VW comment about ignition bit	23/10/2019	rbiasini
Fixes to vw dbc	30/10/2019	rbiasini
Add Motor_20 CRC support (#229)	06/03/2020	Jason Young
Add TSK_06 CRC validation for VW MQB (#234)	16/03/2020	Jason Young
Add SWA_01 message detail and CRC support for VW MQB (#236)	02/04/2020	Jason Young
VW MQB: Corrected CRC pad byte for ACC_10 (#353)	05/03/2021	Jason Young
VW MQB: Updated message and signal data, round 1 (#357)	11/03/2021	Jason Young
VW MQB: Updated message and signal data, round 2 (#358)	15/03/2021	Jason Young
VW PQ: New and updated CAN messaging (#380)	12/04/2021	Jason Young
VW MQB: EPS HCA status enum, comment cleanup (#396)	27/05/2021	Jason Young
VW MQB: Add detail to Blinkmodi_02 (#402)	14/06/2021	Jason Young

Table 91 - Contributions to the DBC files of Volkswagen car