From CAN to ROS: A Monitoring and Data Recording Bridge

Safwan Elmadani
safwanelmadani@email.arizona.edu
University of Arizona
Tucson, AZ, USA

Matthew Nice
matthew.nice@vanderbilt.edu
Vanderbilt University
Nashville, Tennessee, USA

Matthew Bunting
mosfet@email.arizona.edu
University of Arizona
Tucson, Arizona, USA

Jonathan Sprinkle
sprinkle@acm.org
University of Arizona
Tucson, AZ, USA

Rahul Bhadani
rahulbhadani@email.arizona.edu
University of Arizona
Tucson, AZ, USA

Abstract

The Controller Area Network (CAN) bus protocol is used in modern vehicles for sharing messages between several control units within a vehicle. CAN bus messages are encoded with unknown scheme and decoding these messages provide unlimited access to valuable information that is used in many autonomous vehicles applications . This paper proposes a ROS based package (CAN-to-ROS) for monitoring, recording, and real-time and offline decoding of CAN bus messages. The package is developed in the ROS framework to add modularity and ease of integration with other software, and it is written in C++ to guarantee speed of the execution during run-time. For realtime decoding of CAN bus data, CAN-to-ROS package used in conjunction with other library called Libpanda that provide access to CAN bus message from a vehicle. The package was evaluated and tested on a Raspberry Pi with real CAN bus data from a Toyota RAV4. The results confirm the capabilities of CAN-to-ROS package and resulted in using the package in other research projects.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Keywords: CAN-to-ROS, Libpanda, CAN Coach, ROS

ACM Reference Format:
Safwan Elmadani, Matthew Nice, Matthew Bunting, Jonathan Sprinkle, and Rahul Bhadani. 2021. From CAN to ROS: A Monitoring and Data Recording Bridge. In *The Workshop on Data-Driven and Intelligent Cyber-Physical Systems (DICPS '21), May 18, 2021, Nashville, TN, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3459609.3460531

1 Introduction

The Electronic Control Units (ECUs) are embedded systems in vehicles that are designed for controlling different functions. The ECUs are mainly used to control systems such as fuel injection, airbags, air conditioning, power steering control, etc [6]. A vehicle's ECUs are interconnected via the Controller Area Network (CAN) bus [8] so that information can flow between individual ECUs. The CAN bus of a vehicle provides access to data flowing between the ECUs which, to some people, could be valuable for controller design, drive-by-wire design, training and testing machine learning model, etc [9]. However, in its raw state, CAN bus data is not in human readable format and it has to be decoded before it can be used. Furthermore, similar to other network protocols, the CAN protocol does not define encoding scheme for the data, and vehicle manufacturers develop their own proprietary implementation which makes decoding CAN data challenging and very time consuming.

To decode CAN data, a CAN database file (DBC) needs to be provided to identify packets sent over the CAN bus network. The DBC file is an ASCII based translation file that specifies how a CAN data frame is translated to human readable data by defining information such as names, scaling, bit order, offsets, and signed/unsigned. The DBC files are generated by the community through hacking and reverse engineering of CAN bus. The current approaches for decoding CAN bus messages are implemented in Python [2]. Cantools is the most widely used Python package that handles encoding and decoding CAN bus messages. However, the Python code suffers from performance limitations when decoding CAN data in real-time, and integrating the code with off the shelf libraries written in C/C++ to improve the performance is difficult and increases program's complexity.

In this paper, we take a different approach to tackle some of the challenges with decoding CAN bus messages. Our approach is to use ROS as framework to build a package for decoding CAN data in real-time and offline. ROS is a language agnostic, and ROS packages are written mainly in Python and C++. Our approach is advantageous because for heavy tasks and real time processing we can utilize C++ language to achieve fast execution and use Python for simpler tasks and post-processing. In addition, ROS framework has a built-in network system that allows packages written in different languages to communicate with one another. The CAN-to-ROS package has been used in research work (CAN Coach) that requires access to decoded CAN bus messages from a Toyota RAV4 in real-time. The modularity of ROS framework makes the process of integrating the software needed for CAN Coach system feasible. The CAN-to-ROS package was integrated with C++ library i.e. Libpanda [3]that interfaces with the car to provide the CAN Coach system with a decoded stream of sensor data.

The remainder of this paper is organized as follows: section 2 provides an overview of the ROS system, section 3 presents an overview of the CAN-to-ROS package and its components, section 4 discuses visualization of CAN data in ROS, section 5 describes the CAN Coach system, section 6 introduces some limitations. Finally, the conclusion is drawn in section 7.

2 ROS

The Robotic Operating System (ROS) is an open source operating system for robotics [7]. It is a framework that provides the tools and libraries to facilitate the creation of robotic applications [4]. ROS supports a variety of programming languages (C++, Python, and Lisp) and it has a network system that handles communication between processes. ROS applications are deployed in one or more ROS packages. A ROS package is the organizational unit of ROS code and each package contains the source code, libraries and script. ROS runs executable as a process called node. Nodes are able to communicate with each other by exchanging messages through a publisher/subscriber scheme. If a node has data to share, it publishes to the applicable topic and when a node requires some data, it subscribes to the relevant topic. This scheme eases software deployment and integration because many of the low level interactions are handled by the ROS framework. The data in a running ROS system can be easily recorded with a ROS tool and saved into a bag file. The bag file can be used to play back the recorded data to produce a similar environment in a running ROS system.

3 CAN-to-ROS overview

This section discusses the Libpanda library, CAN data and DBC files, and CAN-to-ROS package workflow.

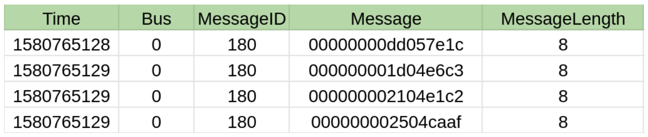
3.1 Libpanda

Libpanda is a low-level C++ library intended to act as an interface with the Comma.ai Panda. The efforts of libpanda stemmed from a lack of data-collection performance from minimal Python-based code on low-cost hardware, resulting in a language shift as well as different design structures. The Python version running on a Raspberry Pi 4 resulted in 92% CPU usage while also dropping 40% of the CAN packets. Through multi threading, observer design patterns, and asynchronous USB calls, libpanda has been able to reduce CPU usage to 35% while not dropping any packets. In addition to this performance, libpanda is able to record GPS data and send control commands to the car.

Libpanda utilizes a thread to establish and maintain a connection with a Panda device using libusb-1.0. The use of the observer design pattern allows classes to be designed in a highly cohesive manner and subscribe to notifications of lower-level handlers. Libpanda provides two observers to USB data, used for GPS and CAN data. These classes can be attached and perform data polling and regular time intervals, then parse notifications of newly incoming data for further observers. In the case of the CAN-to-ROS framework, a ROS node subscribes to the CAN parser provided by libpanda.

3.2 CAN data and DBC files

Libpanda is capable of saving CAN bus data in a CSV file as timeseries data for offline processing. As seen in Figure 1, the CAN messages do not give much information if not decoded. The CAN-to-ROS offline mode takes as input a CSV file with same format shown in Figure 1 and uses it to generate a CAN bus data stream in ROS. The most important columns in the file are Time, MessageID, and Messages. The Time column is used to infer the frequency at which each message is received. The MessageID column links the a message with its DBC file definition and the Message column has the actual sensor data that is yet to be decoded. It is useful to note that a single CAN bus message does not mean a single sensor readings because a message could have multiple signals from different sensors combined depending on the message length.



Time	Bus	MessageID	Message	MessageLength
1580765128	0	180	00000000dd057e1c	8
1580765129	0	180	000000001d04e6c3	8
1580765129	0	180	000000002104e1c2	8
1580765129	0	180	000000002504caaf	8

Figure 1. CAN bus data saved in a CSV file format by Libpanda

In order to decode CAN messages correctly, a DBC file that corresponds to a specific vehicle model is needed to add the proper functions for each message ID. Figure 2 provides the DBC definition of the steering angle for Toyota RAV4 2019. The steering angle messages contains 3 signals

STEER_ANGLE, STEER_FACTION, and STEER_RATE. The following are the key observations :

- The Message name is STEER_ANGLE_SENSOR, message ID is 37, and the length of the message is 8 bytes.
- The signal STEER_ANGLE has the following specifications:
  - **3|12**: the signal starts at bit position 3 and the size is 12 bits.
  - **@0-**: the 0 indicates that the signal is big-endian (1 for little-endian), and the - sign indicates that the signal is signed value (+ for unsigned value).
  - **(1.5,0)**: A scalar to multiply by the signal and offset value.
  - **[-500|500]**: the min and max values of the signal.
  - **"deg"**: units (degrees).

Every CAN bus message must have a definition in the DBC similar to the one described above in order to decode it. However, CAN bus documentation of a vehicle is proprietary and some of the CAN messages' definitions are unknown.

```
BO_ 37 STEER_ANGLE_SENSOR: 8 XXX
 SG_ STEER_ANGLE : 3|12@0- (1.5,0) [-500|500] "deg" XXX
 SG_ STEER_FRACTION : 39|4@0- (0.1,0) [-0.7|0.7] "deg" XXX
 SG_ STEER_RATE : 35|12@0- (1,0) [-2000|2000] "deg/s" XXX
```

Figure 2. DBC definition of the steering angle for Toyota RAV4 2019

3.3 CAN-to-ROS package workflow

The CAN-to-ROS consists of three primary nodes that handle the decoding of CAN bus messages. The nodes are written in C++ to provide fast performance. The package has a core library that nodes can invoke to perform decoding processes. In Figure 3, the realtime_pub and offline_pub nodes are the two nodes that handle preparing the data before decoding. The only difference between them is that realtime_pub node uses CAN parser provided by Libpanda to publish CAN bus messages from the vehicle to a ROS topic named realtime_can. On the other hand, the offline_pub node reads CAN bus data from a file that is passed as argument to it and it publishes the CAN bus messages to the offline_can topic. Finally, the decode_subs node subscribes to both topics i.e. realtime_can and offline_can and it has multiple callback functions that get invoked only when a new message arrives on either topics. When a new message arrives on realtime_can or offline_can topics, a callback function gets invoked that utilizes the core library to pass the message and its ID to a function to decode the message. After the function call inside the callback function returns the decoded message, the callback function proceeds with execution and finishes by publishing the data to its respective topic. This process repeats for every new message that arrives. Figure 3 shows some of the topics that decode_subs node publishes to.

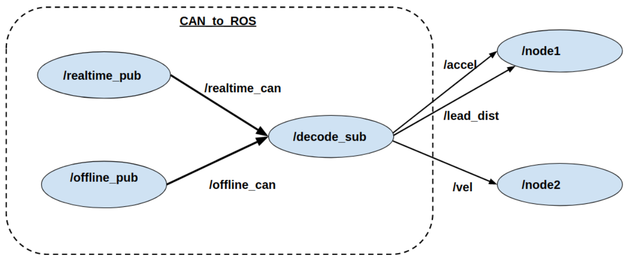


Figure 3. Nodes and topics used in CAN-to-ROS package

4 Data visualization in ROS

ROS framework has great visualization tool e.g. rqt_plot and rviz for visualizing robots and sensor data. Using the CAN-to-ROS package, a bridge between the CAN bus and ROS framework can be created, and sensor data from a vehicle is easily accessible within the ROS environment. A demonstration of such integration is presented in Figure 5 where radar traces from Toyota RAV4 CAN bus are visualized in ROS using a 3D visualization tool (rviz). This is done by publishing the radar traces of the RAV4 as visualization_msgs/Marker messages to ROS topic of the same type, and once rviz is launched the topic is easily added from a drop down menu in the UI. To give a more accurate representation of the vehicle's location with respect to the radar traces, a vehicle model that is used in CAT Vehicle Testbed simulator [1] is imported and used to represent the RAV4's body. In Figure 4, an image of the Toyota RAV4 with a lead vehicle is shown. Comparing Figure 4 and Figure 5, we see that the radar sensor is capturing some of the objects in front of the vehicle and on the sides. The two red dots in the front of the vehicle represent the lead vehicle and the closest red dot on the right side of the car represents the electricity pole.

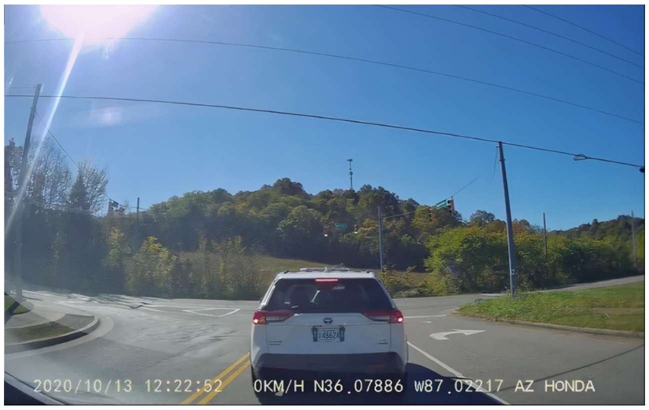


Figure 4. An image of Toyota RAV4 behind another vehicle

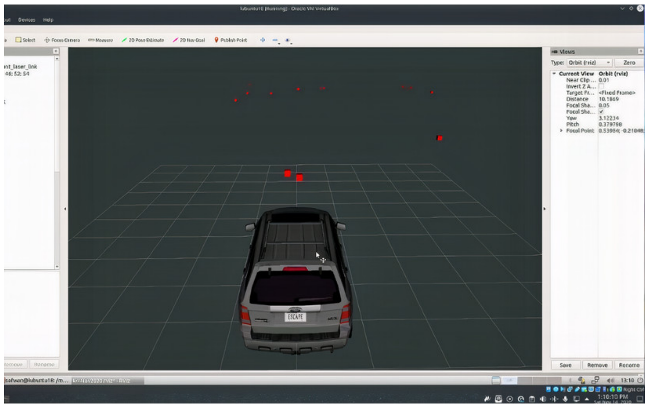


Figure 5. Visualizing radar traces of Toyota RAV4 using rviz and CAN-to-ROS

5 CAN Coach

CAN-to-ROS was instrumental in the human-in-the-loop experiments conducted in [5]. CAN-to-ROS allowed for the smooth transition of critical data from the lower-level CAN data bus to the ROS software layer where there is freedom to process, record, and distribute the information reliably and at high frequency.

The CAN Coach is a system that puts the human into the loop with vehicle sensors, with audio feedback derived from CAN data in real time. The CAN Coach subscribes to velocity, relative distance, and relative velocity data obtained from the CAN bus via CAN-to-ROS.

In field tests, a Raspberry Pi 4 decodes the messages, transforms them into Robotic Operating System (ROS) messages, executes the CAN Coach to generate an auditory feedback for the human-in-the-loop, and records all data from the car and from all ROS nodes. Having CAN-to-ROS allowed for critical data playback from ROS during testing and development, as well as easy analysis.

6 Limitations

During the CAN Coach experiment, we have noticed some inconsistencies between the data recorded with Libpanda Figure 6 versus the same data recorded with ROS tools Figure 7. Later, it was discovered that due to the limited resources of the Raspberry Pi running the ROS system, some of the packets were dropped and new packets received after a couple of seconds. Figure 7 shows the spots of the dropped packets. When running the CAN coach experiment, all CAN bus messages are published to the ROS network at a frequency of approximately 2500 Hz which the ROS communication system could not handle. The complication was resolved by narrowing down the CAN bus messages that were being published to ROS and only publishing messages needed for the system. That reduced the publish rate to approximately 500 Hz.

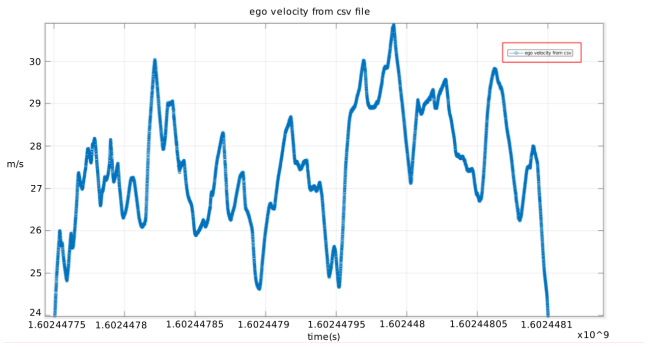


Figure 6. Data recorded with Libpanda

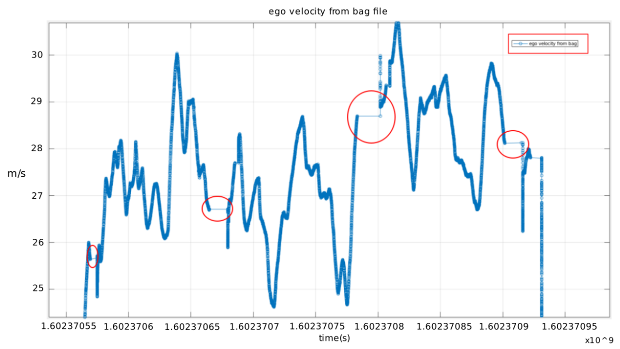


Figure 7. Data recorded with ROSbag

7 Conclusion

Our work has demonstrated the ability to bridge the interface gap between the CAN bus and ROS, opening the door to the use for component-based software additions through ROS middleware. Our results include a demonstration that the acquired data can be analyzed in real-time and feedback given to a human-in-the-loop. We have also demonstrated that the computational burden is such that it is possible to implement the CAN-to-ROS bridge on inexpensive computing hardware such as a Raspberry Pi.

Acknowledgments

This work is supported by the National Science Foundation under awards 1544395 and 1837652. This material is based upon work supported by the U.S. Department of Energy's Office of Energy Efficiency and Renewable Energy (EERE) under the Vehicle Technologies Office award number CID DE-EE0008872. The views expressed herein do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

[1] Rahul Bhadani, Jonathan Sprinkle, and Matthew Bunting. 2018. The CAT Vehicle Testbed: A Simulator with Hardware in the Loop for Autonomous Vehicle Applications. *Proceedings of 2nd International Workshop on Safe Control of Autonomous Vehicles (SCAV 2018), Porto, Portugal,*

10th April 2018, Electronic Proceedings in Theoretical Computer Science 269, pp. 32–47 (2018).

[2] Bhadani, Rahul and Sprinkle, Jonathan. 2020. *Strym: A data-analytic tool for CAN-bus messages*. Department of Electrical & Computer Engineering, The University of Arizona. https://jmscslgroup.github.io/strym/ 0.3.1.

[3] Matthew Bunting, Rahul Bhadani, Safwan Elmadani, and Jonathan Sprinkle. 2020. *Libpanda: A software library and utilities for interfacing with vehicle hardware systems*. The University of Arizona. https://jmscslgroup.github.io/libpanda/

[4] David Come, Julien Brunel, and David Doose. 2018. Improving code quality in ROS packages using a temporal extension of first-order logic. *Encyclopedia with Semantic Computing and Robotic Intelligence* 2, 01 (2018), 1850003.

[5] Matthew Nice, Safwan Elmadani, Rahul Bhadani, Matt Bunting, Jonathan Sprinkle, and Dan. Work. 2021. CAN Coach: Vehicular Control through Human Cyber-Physical Systems. *12th ACM/IEEE International Conference on Cyber-Physical Systems* (2021).

[6] Jayshri Sudhir Potdar and Yashwant B Mane. 2018. Hardware Design and Development of Engine Control Unit for Four Cylinder Engine. In *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*. IEEE, 1–5.

[7] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.

[8] ISO Standard. 2003. Road vehicles–Controller area network (CAN)–Part 1: Data link layer and physical signalling. *ISO* 11898 (2003), 1.

[9] Leng Yi, Li Qingxia, Liu Sheng, and Dong Tianlin. 2008. Direct tire pressure monitoring system based on wireless sensor and CAN bus. *Chinese Journal of Scientific Instrument* 29, 4 (2008), 711.