Olli Autonomous Shuttle Revival: From Low-Level Embedded Architecture to High-Level Perception Integration

By

M V N S H Praneeth

mvnshpra@buffalo.edu

A supervised research project report
submitted to the Faculty of the Graduate School
The University at Buffalo, State University of New York
In partial fulfillment of the requirements for the degree of
Master of Science

Advisor: Dr. Chunming Qiao

Department of Computer Science and Engineering
Connected and Autonomous Vehicles Group (CAVAS)
May 2026

Acknowledgments

I would like to express my deepest gratitude to Professor Chunming Qiao and Steven Korzelius for their invaluable advice, guidance, and continuous support throughout the duration of this project. Their academic and technical insights were instrumental in navigating the complex hardware and software challenges presented by the Olli platform. Their persistent encouragement allowed us to push through complex systems integration barriers.

Special thanks must be given to Gustavo Fortmann. His provision of the comprehensive System Interface Definitions, alongside the vital CAN DBC files (for the Battery, Paravan, and DC-DC converter), served as the foundational blueprint. Without these documents, reverse-engineering the rigid activation states of the high-voltage systems would have been nearly impossible.

Finally, I would like to sincerely thank Marco Bianco, who worked alongside me on this project. His dedicated collaboration, technical assistance, and problem-solving skills were vital to the successful completion of this revitalization effort. Rebuilding a vehicle architecture from the ground up requires extreme coordination, and his partnership was invaluable.

Olli - Autonomous Shuttle

Contents

Abstract **1**

1. Introduction **2**

1.1 Platform Overview and Hardware Specifications 2

1.2 Motivation: A Ground-Up Revitalization 2

1.3 Foundational Guidance: The System Interface Definitions 4

2. Low-Level System Architecture and Methodology **4**

2.1 The Read-Process-Write Paradigm 4

2.2 Dual-Rate Triggered Subsystem Execution 6

2.3 Global Variable Management and Data Packing 7

3. Power Management State Machines **8**

3.1 The Power-Up Sequence (Key ON) 8

3.2 The Controlled Power-Down Sequence (Key OFF) 8

4. Drive-by-Wire and Safety Systems **11**

4.1 Paravan CAN Actuation and CRC Logic 11

4.2 Smoothing Actuation: The Ramp Function 11

4.3 The Mode Arbiter and Emergency Handbrake 12

5. Remote Procedure Call (RPC) Integration **13**

5.1 Dataspeed Controller Architecture 13

5.2 DBW Node Logic and Custom RPC Mapping 14

6. High-Level Perception and Localization Integration **15**

6.1 Network Discovery via Wireshark 15

6.2 ROS 2 WebSocket Bridge Implementation 15

6.3 Sensor Calibration and URDF Preparation 15

7. Conclusion and Future Work **17**

7.1 Project Summary 17

7.2 Future Roadmap 17

Abstract

This technical project report details the comprehensive revitalization of the Olli 3.0 autonomous shuttle platform. The primary objective of this research was to completely erase all legacy, undocumented software and rebuild the vehicle's embedded architecture from the ground up. This bottom-up engineering approach transformed a highly capable but inoperable chassis back into a fully functional drive-by-wire platform, primed to support modern high-level autonomous navigation stacks such as Autoware.

The project is structurally divided into embedded systems integration and high-level perception mapping. Utilizing the MATLAB/Simulink environment, a custom, dual-rate Vehicle Control Unit (VCU) was engineered. This VCU governs the rigid state machines required for the vehicle's Power-Up handshake, Battery Management System (BMS) monitoring, and a software-latched Power-Down sequence. Following power stabilization, the Paravan Space Drive II drive-by-wire system was actively reverse-engineered. CRC-16 checksums and synchronized message counters were developed in MATLAB to satisfy watchdog requirements, allowing precise CAN-based control over steering, braking, and throttle.

A robust safety framework was established through an embedded mode arbiter, which strictly prioritizes physical human interventions (via joystick or physical switches) over automated commands. Furthermore, a Remote Procedure Call (RPC) integration layer was developed using C++ ROS nodes and a Dataspeed controller, translating high-level trajectory requests into low-level Paravan CAN directives.

The final phase of the project focused on recovering the "eyes and ears" of the bus. Through extensive network sniffing and packet analysis, communication was restored with the vehicle's three Velodyne LiDARs, six Axis F44 cameras, and a proprietary Robotic Research (RR-N-140) GPS/IMU navigation module. A custom WebSocket bridge was authored to bypass a locked web interface, translating legacy navigation data into native ROS 2 topics. This report chronologically documents the algorithms, workflows, and logic implementations that successfully restored the Olli platform to an autonomy-ready state.

1. Introduction

1.1 Platform Overview and Hardware Specifications

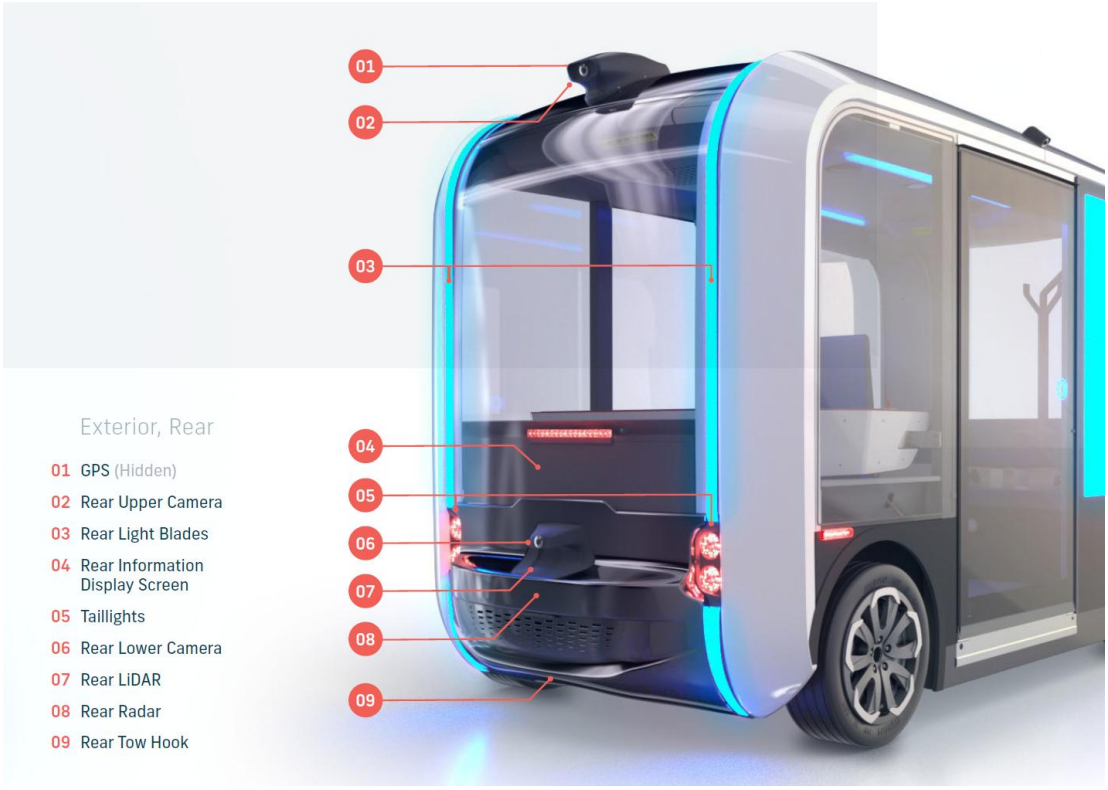
The Olli 1.0 shuttle is a heavily equipped autonomous platform designed for localized, low-speed urban transit. Functioning as a technological sandbox, the vehicle is outfitted with a comprehensive suite of hardware designed to achieve Level 4 autonomy. The primary perception hardware includes an array of three Velodyne LiDAR sensors strategically mounted to capture the forward, upward, and rear topographies, alongside six Axis F44 multi-sensor camera mainframes providing 360-degree high-definition visual coverage.

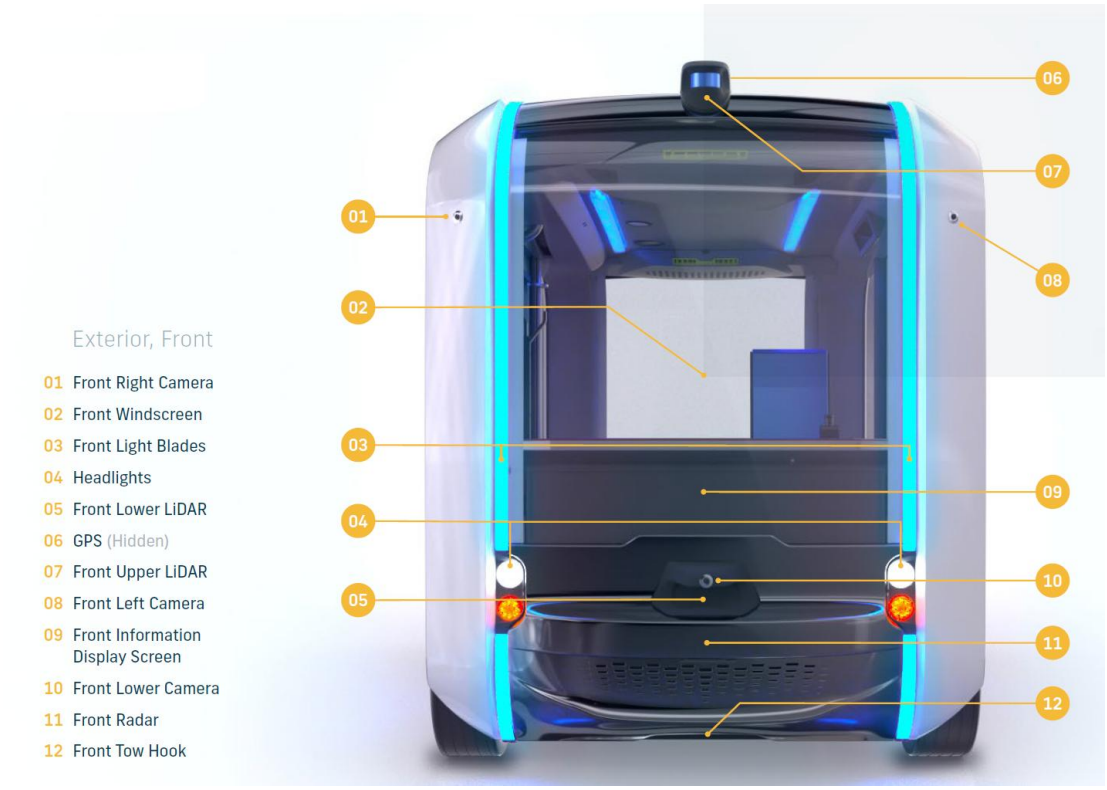
Locomotion and physical actuation are governed by the Paravan Space Drive II system. Paravan is an industry-standard drive-by-wire modular unit that sits between the vehicle's computational brain and its physical steering racks and inverter systems. It mandates strict communication protocols over Controller Area Network (CAN) buses to prevent unauthorized or unsafe vehicle maneuvers. To manage these complex protocols, a New Eagle GCM196 Vehicle Control Unit (VCU) was installed to store and execute the entirety of the low-level logic, while the three onboard Nuvo embedded computers (featuring Intel i7 processors and 16GB of RAM) are reserved strictly for the high-level autonomy stack.

1.2 Motivation: A Ground-Up Revitalization

Prior to the commencement of this project, the Olli bus was rendered inoperable. The vehicle was initially purchased by the university from Local Motors. However, after the company closed down, the proprietary software IP and legacy code remained entirely inaccessible. Consequently, our team made the strategic decision to install a new New Eagle GCM196 Vehicle Control Unit (VCU) and build the entire low-level software architecture from scratch. This bottom-up approach ensured complete engineering sovereignty over the vehicle.

Consequently, the central motivation of this project was engineering sovereignty. Our team made the decision to completely erase the legacy codebase and revive the bus from a blank slate. By starting from absolute zero, we ensured that every sub-system was seamlessly integrated.





Exterior Sensor Positions of the Bus

1.3 Foundational Guidance: The System Interface Definitions

Rebuilding an electric vehicle from scratch requires exact knowledge of its electrical limits and digital handshakes. This critical knowledge gap was bridged by Gustavo Fortmann, a former CTO involved in the original engineering of the bus. We were supplied with a highly technical "System Interface Definitions" PDF, alongside standard CAN DBC (Database CAN) files corresponding to the vehicle's Battery Management System (BMS), Paravan module, and DC-DC converter.

This documentation provided comprehensive flowcharts, CAN message structures, and logic guidelines required to govern the vehicle's core states. Specifically, the PDF detailed the rigid sequences for the Low Voltage (LV) power-up, the High Voltage (HV) request and Battery Management System (BMS) handshakes, the safe power-down sequences, and the exact heartbeat formats needed to actuate the Paravan drive-by-wire and Remote Procedure Call (RPC) systems. The primary challenge of the initial phase of this project was translating his theoretical flowcharts and rules into robust, executable MATLAB/Simulink logic blocks capable of running on the embedded New Eagle Vehicle Control Unit (VCU).

2. Low-Level System Architecture and Methodology

To control the physical hardware of the bus safely, a robust real-time operating environment was required. We utilized the MATLAB/Simulink environment, augmented with Raptor Dev tools, to visually construct and compile the embedded logic. This architecture ensures determinism—meaning the code executes at strictly defined intervals, a mandatory requirement for drive-by-wire safety.

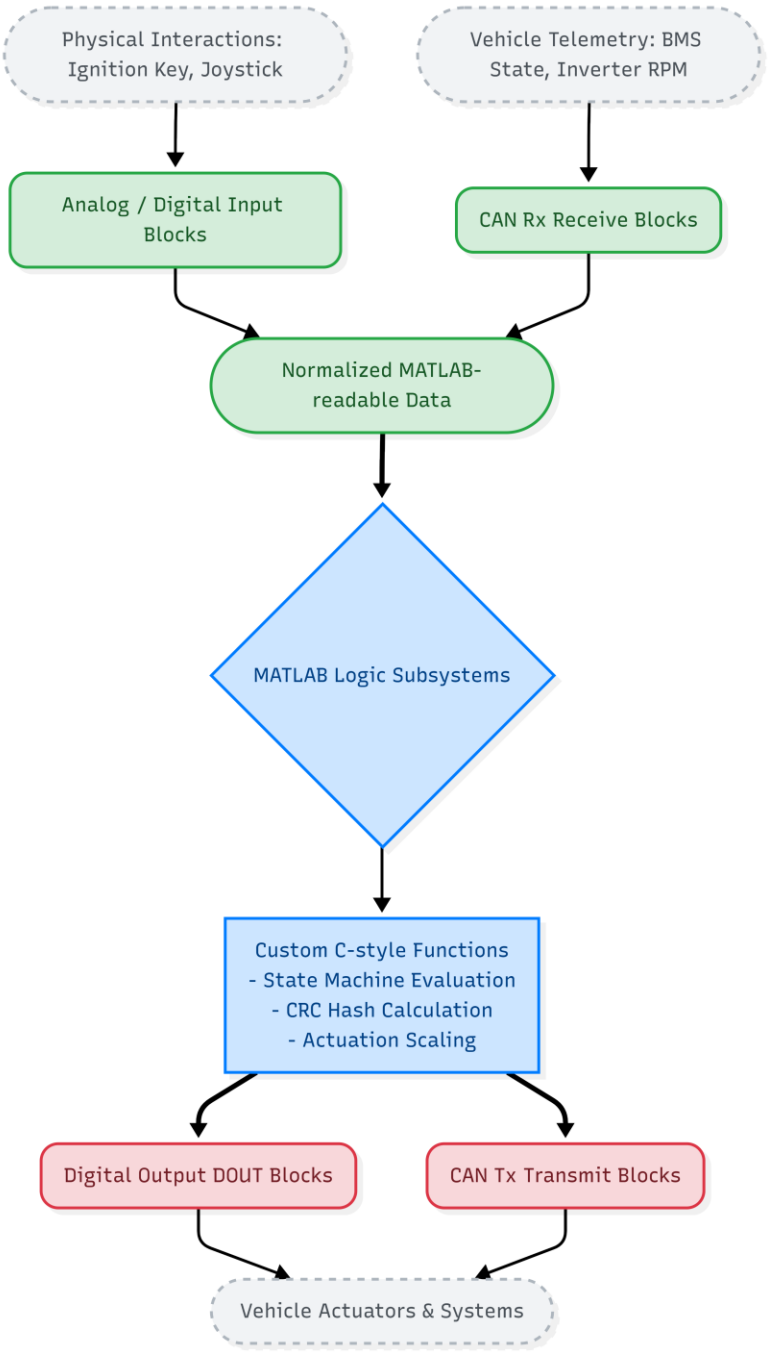
2.1 The Read-Process-Write Paradigm

The VCU architecture was designed around a strict read-process-write loop.

1. Read: Physical interactions (e.g., turning the ignition key, toggling the joystick) and vehicle telemetry (e.g., BMS state, inverter RPM) are ingested through specific Analog Input, Digital Input, and CAN Rx (Receive) blocks. These raw electrical signals and CAN payloads are normalized into MATLAB-readable integers and floats.

2. Process: The normalized signals are routed into complex MATLAB Logic Subsystems. Inside these blocks, custom C-style MATLAB functions evaluate the inputs against our state machines, calculating the necessary responses, CRC hashes, and scaling factors.

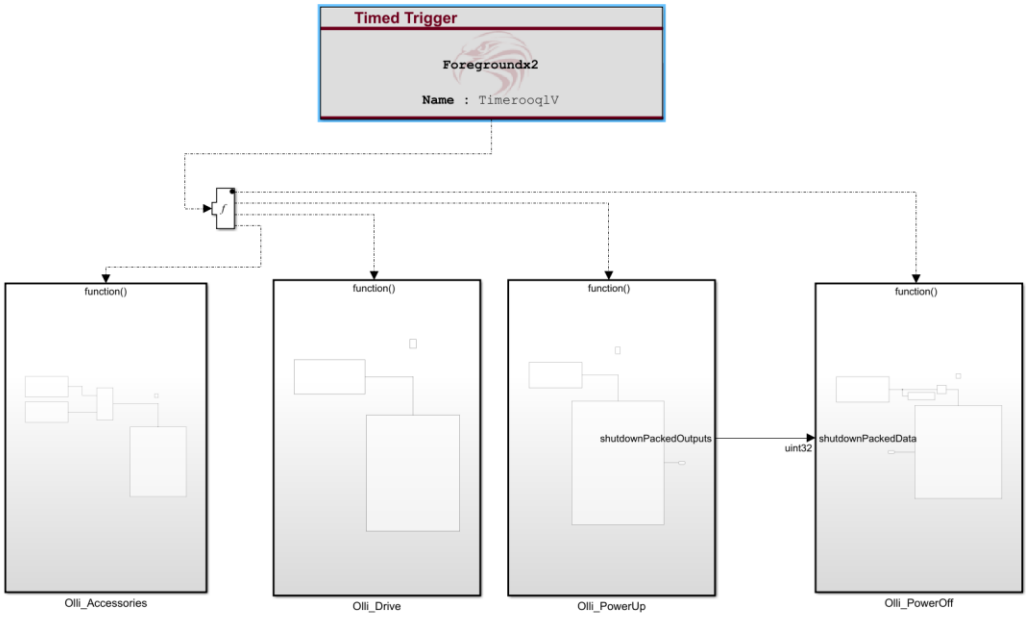
3. Write: The calculated state decisions and drive commands are dispatched back to the vehicle hardware via Digital Output (DOUT) and CAN Tx (Transmit) blocks.



2.2 Dual-Rate Triggered Subsystem Execution

A critical challenge discovered during early testing was CAN bus flooding. Initially, the Simulink model was configured with a fixed-step solver running at 0.005 seconds (200 Hz). While this provided rapid logic execution, it overwhelmed the Paravan drive-by-wire ECU. The Paravan watchdog, expecting a heartbeat and command updates at approximately 40ms intervals (as indicated by the system interface PDF), registered the 200 Hz transmission rate as a "flood" and subsequently triggered critical communication faults, halting the vehicle.

To address this and optimize processing overhead, the architecture was restructured using a dual-rate execution model via Enabled Subsystems and Function-Call Split blocks. When the vehicle key is in the ON position, all primary modules—Power-Up, Arbiter, and Drive Controllers—run on a timed foreground trigger of 100 Hz (a 10ms fixed-step size). This rate represents the industry standard for automotive steering and braking control; it provided precise responsiveness without triggering the Paravan flood safeguards. Conversely, the specialized Power-Off submodule was designed to operate as a background process running at a slower 50 Hz rate. This background task exclusively enables when the physical key is turned OFF, taking over the system resources to manage the slow, controlled de-energization of the bus.



Four Main Submodules Connected to 100Hz Timed Trigger

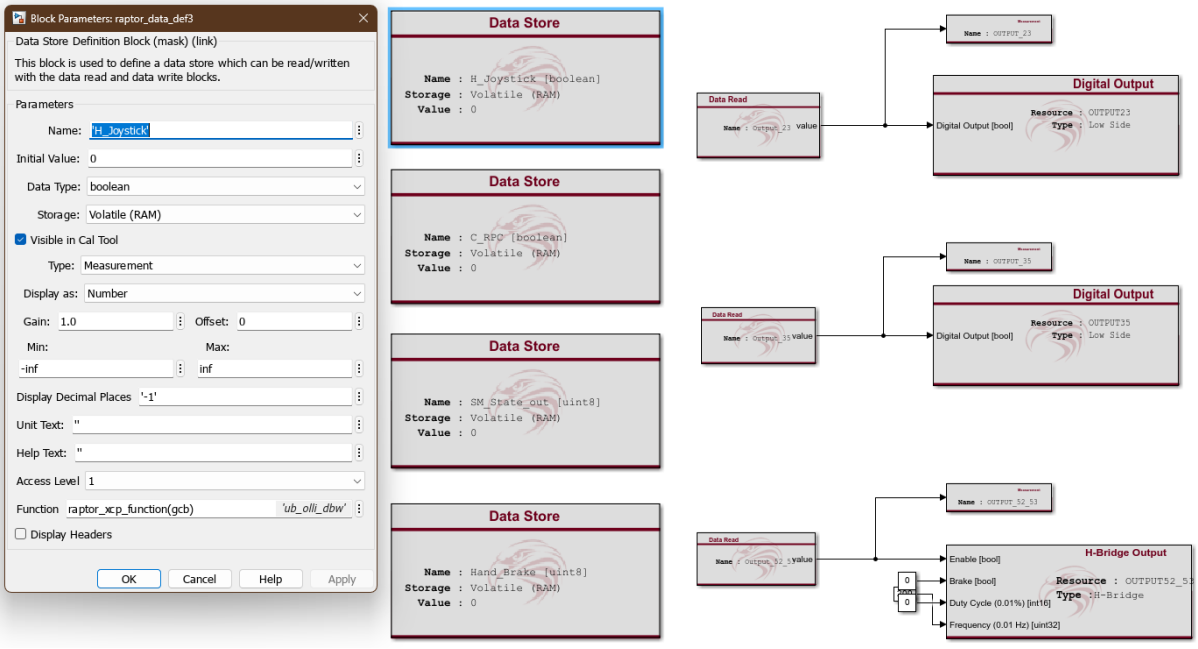
2.3 Global Variable Management and Data Packing

As the VCU logic grew to encompass the BMS, DC-DC converter, steering, braking, and fault monitoring, drawing individual signal lines between every block resulted in visual and architectural chaos. To maintain a scalable, clean workspace, "Data Store Read" and "Data Store Write" blocks were heavily utilized. These blocks act as global memory allocations within the embedded RAM. They allow critical parameters—such as the `KeepModuleOn` software latch and target RPMs—to be securely shared across completely separate subsystems without direct wiring.

Furthermore, for the transition between the 100 Hz Active state and the 50 Hz Shutdown state, a bit-packing methodology was employed. To ensure the Shutdown module knew the exact state of the bus the millisecond the key was turned off, the active module packs all boolean logic states into a single 32-bit unsigned integer (`uint32`) before passing it globally:

```
digital_inputs_packed = bitset(digital_inputs_packed, 1, dout4);
digital_inputs_packed = bitset(digital_inputs_packed, 2, dout6);
```

This single integer is passed to the background process, which dynamically unpacks it to freeze the last known state of the outputs before safely shutting them down sequentially.



Data Store and Data read Blocks

3. Power Management State Machines

Modern electric and autonomous vehicles cannot simply be turned on like a traditional combustion engine. High-voltage (HV) relays require pre-charging to prevent electrical arcing, and sensitive computational payloads (like LiDARs and embedded PCs) require graceful shutdowns to prevent data corruption. Chapters 3 details the strict Finite State Machines (FSMs) developed to manage these critical power transitions.

3.1 The Power-Up Sequence (Key ON)

Following the flowcharts defined in Gustavo Fortmann's system interface documentation, the power-up sequence was implemented as an FSM residing within the `Olli_PowerUp` MATLAB function. The states are rigidly defined as follows:

```
HVREADY   = uint8(0);
HVREQUEST = uint8(1);
LVPULSE   = uint8(2);
IDLE      = uint8(3);
```

Upon detecting the key ON signal, the system immediately asserts a software latch (`KeepModuleOn = true`). This is a critical safety feature: it ensures that even if the physical ignition key is violently toggled or fails, the VCU maintains electrical power over itself to execute logic.

The logic then transitions to `LVPULSE`, sending a low-voltage wake-up signal to the auxiliary systems. Once awake, the VCU transitions to `HVREQUEST`. During this phase, it actively monitors the Battery Management System over CAN. It checks the `state_BMS` and verifies that the `minSOC_BMS` (Minimum State of Charge) is above safe threshold levels. If the BMS reports healthy isolation and cell temperatures, the high-voltage contactors close. Finally, the logic asserts the `dcdc_enable_out` signal, firing the DC-DC Converter to step down the high voltage to the 12V/24V required to fully power the onboard computers and Paravan systems.

3.2 The Controlled Power-Down Sequence (Key OFF)

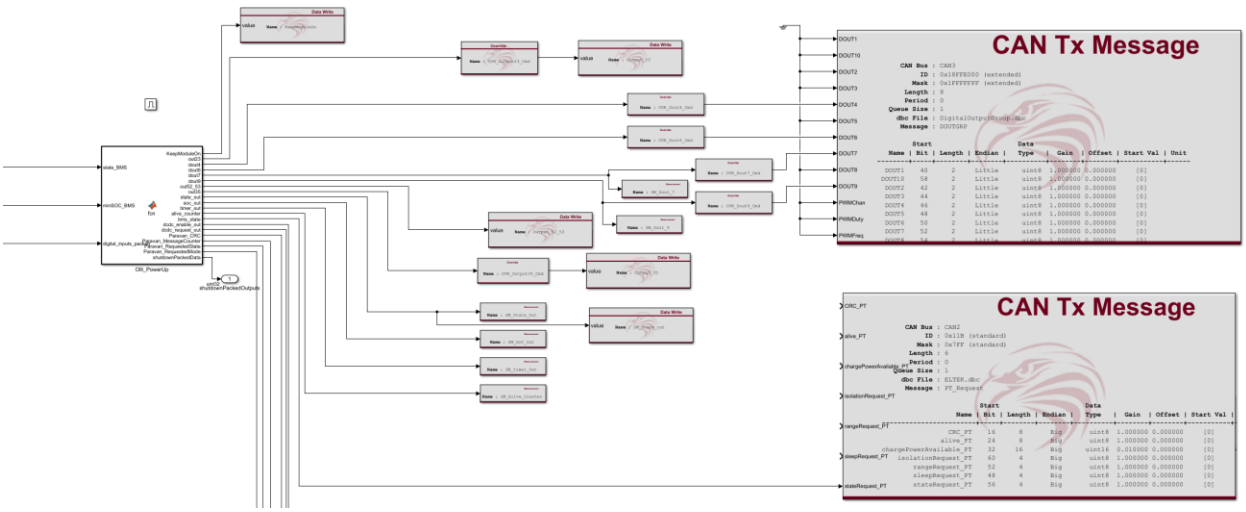
Shutting down the bus abruptly by severing the 12V line would disrupt the GCM196 VCU's logic states and cause severe physical damage to the spinning Velodyne LiDAR motors. Therefore, a controlled power-down sequence was engineered.

When the physical key is turned OFF, the primary 100 Hz logic suspends, and the 50 Hz Shutdown background process activates. Because the `KeepModuleOn` latch is still true, the VCU remains alive. The shutdown script first unpacks the `shutdownPackedData` to understand which relays are currently active.

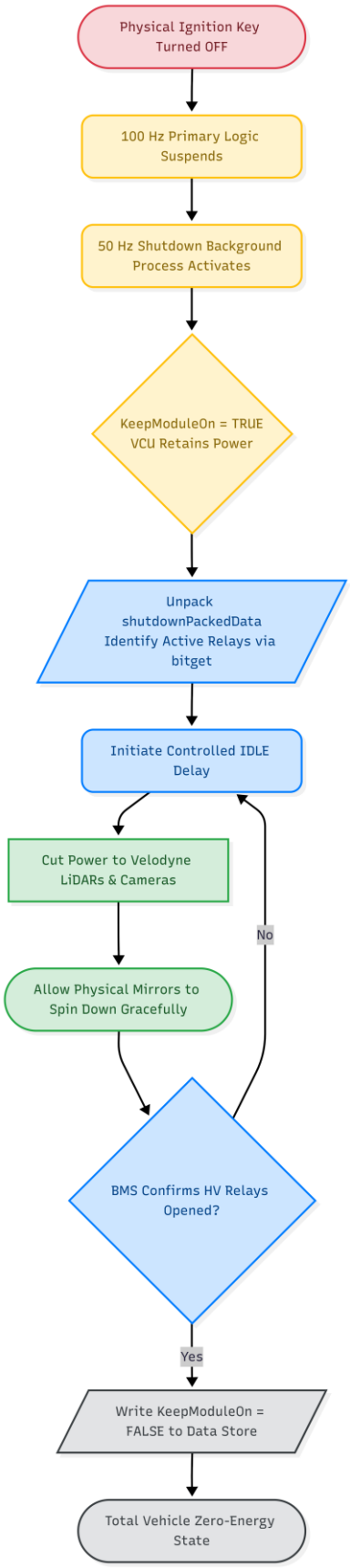
```
shutdownPackedData = uint32(shutdownPackedData);
outs.dout4  = uint8(bitget(shutdownPackedData, 1));
outs.dout6  = uint8(bitget(shutdownPackedData, 2));
outs.dout7  = uint8(bitget(shutdownPackedData, 3));
```

Once unpacked, the system initiates a controlled IDLE delay. During this window, the GCM196 VCU actively maintains its own power via the ‘KeepModuleOn’ software latch. Simultaneously, power is cut to the LiDAR and camera sensors, allowing the physical LiDAR mirrors to spin down gracefully before the VCU finally severs its own power relay

The VCU confirms that the BMS has safely opened the high-voltage relays, does the shutdown script execute its final command: writing `false` to the `KeepModuleOn` Data Store. This drops the primary relay, achieving total vehicle zero-energy state.



Simulink Power-up Logic Blocks


Physical Ignition Key Turned OFF
100 Hz Primary Logic Suspends
50 Hz Shutdown Background Process Activates
KeepModuleOn = TRUE VCU Retains Power
Unpack shutdownPackedData Identify Active Relays via bitget
Initiate Controlled IDLE Delay
Cut Power to Velodyne LiDARs & Cameras
Allow Physical Mirrors to Spin Down Gracefully
BMS Confirms HV Relays Opened?
No
Yes
Write KeepModuleOn = FALSE to Data Store
Total Vehicle Zero-Energy State

4. Drive-by-Wire and Safety Systems

With the chassis safely energized, the next phase required interfacing with the Paravan Space Drive system. Paravan acts as the muscular system of the bus, translating digital CAN messages into physical hydraulic and electrical actuations for steering, throttle, and braking.

4.1 Paravan CAN Actuation and CRC Logic

The Paravan module is highly secure. It will ignore any CAN message that does not contain a synchronized message counter and a valid CRC-16 (Cyclic Redundancy Check) hash. This prevents rogue nodes or corrupted packets from randomly turning the steering wheel.

Within the `Master_Drive_Controller` MATLAB function, a dedicated `build_can_msg` subroutine was created to dynamically generate these packets at 100 Hz:

```
function [crc, msg_cnt] = build_can_msg(pos_u16, spd_u16, counter)
    packet = uint8([bitand(pos_u16, 255), bitshift(pos_u16, -8), ...
                    bitand(spd_u16, 255), bitshift(spd_u16, -8)]);
    crc = calculate_crc16(packet);
    msg_cnt = counter;
end
```

One major implementation hurdle involved resolving critical CAN communication errors during standstill. When sending a command of '0' for steering and braking, the Paravan system would throw a critical fault. Through analysis of the interface documentation, it was discovered that the Paravan system does not recognize '0' as neutral. Instead, a strict scaling offset of +16384 must be applied to the center values for both steering and braking parameters. Once this constant offset was injected into the MATLAB bit-shifter, the standstill communication faults were entirely eliminated.

4.2 Smoothing Actuation: The Ramp Function

Autonomous trajectory algorithms can sometimes output sudden, jagged requests (e.g., commanding a full brake lock instantly). Passing these raw values directly to Paravan would result in violent jerking. To mitigate this, a mathematical ramping function was implemented within the drive controller:

```
function new_val = calculate_ramp(current, target, req_rate, dt)
    if req_rate <= 0, req_rate = 1000.0; end
    step = req_rate * dt;
    if abs(target - current) <= step
        new_val = target;
    else
        new_val = current + (sign(target - current) * step);
    end
end
```

This function evaluates the `target` value requested by the software against the `current` physical position of the actuator. By multiplying a maximum allowed `req_rate` by the delta time `dt` (0.01s), it limits the physical step size per cycle, ensuring smooth, human-like acceleration and braking regardless of the autonomy stack's input.

4.3 The Mode Arbiter and Emergency Handbrake

Safety during testing is paramount. A Mode Arbiter submodule was developed to serve as the ultimate gatekeeper for actuation commands. The logic continuously polls `DOUT7_State` (Joystick Human Active Flag), `gas_brake_counts`, and the `forward_sw/reverse_sw` physical toggles.

The arbiter enforces strict priority: any physical input detected from the human operator immediately supersedes and overwrites autonomous CAN commands. If an autonomous command is executing, and the safety operator touches the joystick, the VCU instantly ignores the software and executes the manual joystick vector.

Furthermore, an emergency handbrake override logic was implemented. The VCU continuously monitors a remote RPC heartbeat. If the automated connection drops, times out, or reports a `CommsLoss` fault, the system triggers the `calculate_ramp` function to safely but rapidly clamp the braking system, dropping the vehicle into a safe IDLE state.

5. Remote Procedure Call (RPC) Integration

To bridge the gap between the low-level Simulink VCU and the high-level Ubuntu autonomy stack, a Remote Procedure Call (RPC) translation layer was engineered. This layer allows ROS-based software to issue high-level commands (like "drive forward at 2 m/s") that are subsequently translated into the raw CAN signals Paravan expects.

5.1 Dataspeed Controller Architecture

To facilitate automated manual control and safety interventions, a Dataspeed controller was mapped to the system. This physical gamepad features dual joysticks, proportional throttle/brake triggers, and discrete buttons for Drive, Park, and Neutral modes.

Dataspeed Controller

On the primary testing machine, an HP Omen laptop running Ubuntu, dedicated C++ ROS nodes were utilized to interface with this controller. The architecture is split into two logical tasks. The first task polls the USB interface, capturing raw joystick voltage values and normalizing them into standardized ROS sensor_msgs/Joy floats. The second task encodes these inputs into our custom RPC-readable CAN payloads, packaging them onto the CAN bus using explicit IDs: 300 (Steering), 200 (Braking), and 100 (Throttle/Shift)

5.2 DBW Node Logic and Custom RPC Mapping

Instead of writing the ROS translation layer completely from scratch, an open-source ROS node from Dataspeed's GitHub repository was utilized as the foundational framework. The core engineering task involved heavily modifying and tweaking this source code to bridge the gap between the standard controller inputs and our proprietary VCU logic.

Specifically, the node was adapted to translate the generic sensor_msgs/Joy outputs into the exact CAN message formats required by our Simulink RPC block. By stripping out unnecessary legacy logic and injecting our custom encoding schemes, the C++ node successfully maps physical joystick vectors and button presses to CAN IDs 100, 200, and 300. This ensures that the high-level ROS environment speaks the exact same digital language as the low-level Paravan drive controllers executing on the embedded GCM196.

Simultaneously, on the embedded side, dedicated CAN Rx blocks inside the Simulink model listen exclusively for IDs 100, 200, and 300. When valid payloads arrive, the VCU extracts the target values, routes them through the safety arbiter described in Chapter 4, and feeds them into the `calculate_ramp` functions. This completes the full Drive-by-Wire loop:

Physical interaction -> C++ ROS Node -> RPC CAN Message -> Simulink Arbiter -> Paravan Actuation.

6. High-Level Perception and Localization Integration

With the chassis fully driveable by code, the project shifted focus upward. Autonomous navigation requires a steady stream of highly accurate environmental data. The objective was to reconnect the vehicle's proprietary sensor suite to a modern ROS 2 Humble environment.

6.1 Network Discovery via Wireshark

Because the legacy network topology was completely undocumented, packet sniffing via Wireshark was necessary to map the local network on the `NAV-ETH` port. By filtering traffic, several key assets were discovered:

1. Axis F44 Cameras: Static IP addresses (e.g., 192.168.0.100) were identified, allowing us to reset and access the administrative interfaces of the 6 camera modules.

2. Velodyne LiDARs: Heavy UDP traffic directed to port 9347 confirmed the presence of the LiDAR units streaming point cloud data (originating from IPs such as 192.168.1.201).

3. RR-N-140 Navigation Module: The proprietary Robotic Research navigation module, housing both a NovAtel GNSS unit and a NavChip IMU, was detected on the network. However, its direct web interface was securely locked.

Axis F44 Cameras

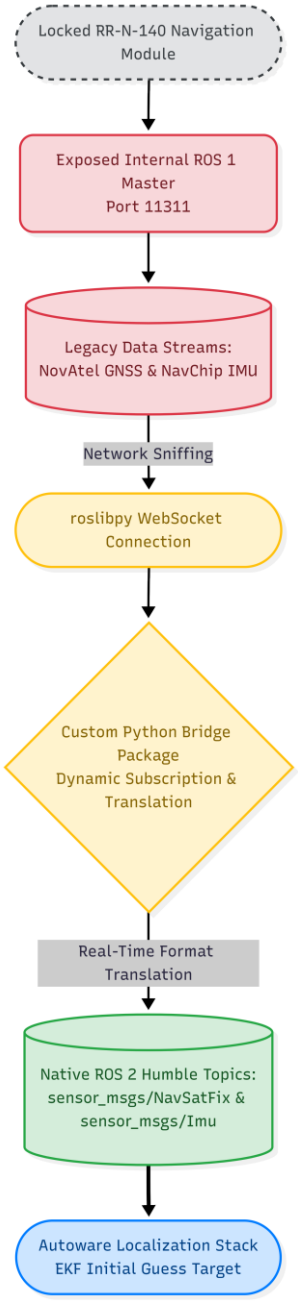
Velodyne Lidar

RR-N-140 Module

6.2 ROS 2 WebSocket Bridge Implementation

Extracting high-precision localization data from the locked RR-N-140 unit proved to be a significant challenge. However, further network sniffing revealed an exposed internal ROS 1 master operating on standard port 11311 inside the module.

To bypass the locked UI and extract the telemetry, a custom Python bridging package was developed utilizing `roslibpy` (a WebSocket library). This script successfully tapped into the internal NovAtel and IMU topics. It dynamically subscribes to the legacy ROS 1 messages and translates them, in real-time, into native, modern ROS 2 `sensor_msgs/NavSatFix` and `sensor_msgs/Imu` formats. This bridge is a critical achievement, as it provides the essential "Initial Guess" required by the Extended Kalman Filter (EKF) in the Autoware localization stack.



7. Conclusion and Future Work

7.1 Project Summary

This project represents a complete, successful ground-up restoration of the Olli 1.0 autonomous platform. By rigorously interpreting system interfaces and overcoming severe legacy software hurdles, the vehicle was transformed from a non-responsive shell into a highly secure, drive-by-wire platform. The dual-rate MATLAB/Simulink architecture ensures absolute stability in the high-voltage and Paravan handshakes, while the custom C++ RPC nodes and WebSocket bridges successfully integrate modern ROS 2 frameworks with proprietary hardware.

7.2 Future Roadmap

With the low-level chassis fully operational and perception data actively broadcasting, the physical infrastructure is prepared for full autonomy deployment. Immediate future work will focus on three key pillars:

1. RTK-GPS Integration: Leveraging the university's NTRIP local network base station to provide real-time differential corrections to the Nav module, upgrading standard GPS accuracy to centimeter-level precision.

2. Multi-LiDAR Point Cloud Merging: Utilizing the recovered extrinsic matrices to merge the forward, top, and rear Velodyne datastreams into a single, cohesive 360-degree point cloud representation in ROS 2.

3. Autoware Deployment: With the URDF complete and sensor data localized, the final step involves launching the Autoware environment to enable dynamic path planning, obstacle avoidance, and fully autonomous waypoint navigation.

The revitalization of the Olli platform establishes a robust, highly documented baseline for all future connected and autonomous vehicle research at the university.