ChatGPT

Design of a RedByte-Based HIL Telemetry Platform for a VSM Inverter

Introduction and Project Overview

This paper presents the design of a **hardware-in-the-loop (HIL) telemetry and simulation software platform** for a senior design project at Gannon University. The project involves a three-phase grid-forming inverter operating as a **Virtual Synchronous Machine (VSM)** – an inverter controlled to emulate the inertial and voltage/frequency response of a traditional synchronous generator [1]. The interdisciplinary team includes Connor Angiel (Cyber Engineering student, software lead) and two Electrical Engineering students, Hannah and Antreas, who developed the inverter hardware and HIL setup. The software platform, derived from Gannon's RedByte software stack, is a desktop application that interfaces with the real-time inverter system to monitor, record, and replay telemetry data, supporting both live experiments and simulated scenarios.

The motivation for this platform stems from the need to **validate and analyze the VSM inverter's performance under realistic conditions** without risking physical equipment. Traditional synchronous generators inherently stabilize grid transients through their rotating inertia [2]. In contrast, inverter-based distributed resources lack physical inertia; a VSM control algorithm addresses this by **replicating inertia and damping characteristics via software control**, improving microgrid stability [1]. Our HIL setup places the physical inverter controller (a microcontroller-based unit) in loop with a real-time simulator (such as an OPAL-RT system) that emulates the grid and load environment. The microcontroller exchanges signals with the simulator in real time, allowing safe testing of grid transients and control strategies. **Figure 1** illustrates the project context: the microcontroller (running VSM control firmware) interfaces with both a simulated power environment and the new telemetry software.

(Figure 1: System Context – A microcontroller running VSM control connects to an OPAL-RT real-time grid simulator and sends telemetry via UART to the RedByte-based desktop application for monitoring and analysis.)

The **software requirements** were defined to address the team's needs for data visualization, experiment repeatability, and system verification. Key capabilities include:

- **Serial Telemetry Ingestion:** Reading real-time data from the microcontroller's UART output (JSON-formatted telemetry frames) at the inverter's control loop rate.
- **Real-Time Monitoring UI:** Displaying key signals (voltages, currents, frequency, etc.) on a live dashboard akin to an oscilloscope or SCADA panel, for immediate feedback during tests (similar to Typhoon HIL's SCADA interface for VSM models [3]).
- **Deterministic Session Recording:** Logging all incoming data with precise timestamps and system state, such that a test session can be recorded in full.
- **Replay & Comparison Tools:** Replaying recorded sessions through the software to compare against live or expected results. The platform should overlay original vs. replayed data for verification, enabling "time-travel" debugging of inverter behavior [4] [5].

- **Exportable Data Capsules:** Saving recorded sessions and associated meta-data to portable files (e.g. JSON or CSV) that encapsulate the test conditions and results, for offline analysis or sharing.
- **Simulation Scenario Control:** Providing an interface to configure and run predefined HIL simulation scenarios (e.g. load step changes, grid faults) in coordination with the real-time simulator (e.g. via OPAL-RT APIs). This allows repeatable experiments where the software triggers certain events and captures the inverter's response.
- **Modularity & Extensibility:** The application is built with a **desktop-first** approach but structured like a SaaS platform in modular components. This means it can run standalone on a lab PC, yet its architecture (separated data ingestion, processing, UI modules) could scale to a distributed or cloud-based solution in the future.
- **Non-Intrusive Integration:** The platform augments existing team tools without replacing them. The microcontroller currently uses **Blynk**, a low-code IoT dashboard, to display some telemetry on a mobile app [6], and logs data to CSV files and physical LCD displays. The new software listens passively to the same telemetry stream (UART JSON) so that Blynk and other tools remain operational in parallel. It acts as an additional layer of insight and control for the engineers.

In summary, the project's goal is to deliver a **unified HIL software environment** that leverages the robust features of RedByte (multi-window GUI, real-time simulation control, recording/replay, etc.) and tailors them to the domain of power electronics and grid simulation. By building on an existing stack, the team accelerates development while ensuring the solution meets the **ECE 358 Senior Design** course requirements for technical rigor, testing, and professional practice. The remainder of this paper details the technical design, functional decomposition, validation plans, and how the platform fulfills academic and engineering standards.

Technical Background

Virtual Synchronous Machine (VSM) and HIL Simulation

A **Virtual Synchronous Machine** refers to an inverter control strategy that emulates the dynamics of a synchronous generator. In essence, VSM-controlled inverters are **power electronic converters with control algorithms that mimic inertia and damping**—responding to grid frequency changes and voltage disturbances as a real machine would [1]. This approach improves stability in grids with high inverter penetration by providing virtual inertial response and droop control for frequency and voltage support [7] [8]. Our project implements a VSM on a three-phase inverter: the microcontroller adjusts the inverter's output phase and magnitude according to swing equations and droop characteristics, making the inverter "act" like a generator with configurable inertia.

For development and testing, a **Hardware-in-the-Loop simulation** is utilized. HIL simulation involves connecting the real controller hardware (our microcontroller running VSM firmware) to a real-time plant model. In our setup, an OPAL-RT simulator (or similar real-time system) runs a model of the grid and inverter power stage. The microcontroller reads simulated sensor values (e.g. grid frequency, voltages, currents) from the HIL system and sends back control signals (PWM commands) to drive the simulated inverter. This allows testing scenarios such as grid frequency drops, load steps, or faults in a safe, controlled environment. OPAL-RT's real-time platform is well-suited for this purpose – it can execute complex power system models with high fidelity and speed, enabling closed-loop operation with actual control hardware [9]. By using HIL, the team can validate the VSM algorithm's performance (stability, transient response) before any real power hardware is energized.

The telemetry of interest from the microcontroller includes internal states (e.g. estimated rotor angle, frequency, voltage setpoints) and outputs (three-phase currents, voltages, etc.). These are packaged as JSON messages and sent over UART at regular intervals (synchronized with the control loop). The **Blynk** IoT platform is used in parallel for remote monitoring; Blynk allows a smartphone or web dashboard to display selected variables in real-time, by sending the telemetry to a cloud server [6]. Blynk's convenience is valuable for quick checks, but it is limited in data depth and offline analysis. Our platform complements this by capturing **high-resolution telemetry locally**, providing advanced analysis (like oscilloscope traces and automated comparisons), and facilitating reproducible testing which Blynk cannot do.

RedByte Software Stack and OS Metaphor

The foundation of our software design is inspired by **RedByte**, an interactive digital logic simulation platform originally developed at Gannon. RedByte runs in a web browser and presents itself as a mini operating system (RedByte OS) for managing multiple simulation "apps" [10] [11]. Though RedByte was built for educational digital logic circuits, its architecture offers a proven template for building complex interactive simulation tools. Key aspects of RedByte we leverage are:

- **Desktop Environment & Windowing:** RedByte OS provides a familiar desktop-like interface with resizable windows, a taskbar, and multiple apps running concurrently [12] [13]. This metaphor will reduce users' learning curve and allow our team to open several tools at once (e.g. a live monitor window, a replay analysis window, a settings console). By repurposing RedByte's React-based desktop shell and window manager, we gain a robust multi-window UI framework out of the box [14] [15].
- **Logic Playground -> Inverter Playground:** In RedByte, the main app is the **Logic Playground** for circuit design and simulation [16]. We will create an analogous core app, tentatively called **Inverter Lab**, tailored to power electronics. Instead of logic gates on a schematic, this app's workspace will show live waveforms, system diagrams, and controls for the VSM inverter. The underlying principles remain: an interactive canvas where the user can start/stop simulation, inject events, and probe signals. The RedByte Files app (virtual file system) and Settings app can also be adapted to manage saving session records and adjusting global options (like telemetry rate or themes) [17] [18].
- **Real-Time Simulation Loop:** RedByte's simulation engine uses a tick-based deterministic loop for updating logic circuits state every tick [19] [20]. In our context, "ticks" correspond to real-time steps (e.g. control loop cycles or milliseconds). We ensure the software can keep up with the microcontroller's data stream in real time, and we borrow RedByte's concept of a **controllable clock** – the ability to pause, resume, or step through simulation time. This is essential for replay and analysis: during a replay, we want to step through each recorded sample similarly to how RedByte steps through circuit ticks.
- **Probes and Oscilloscope:** RedByte allows users to attach probes to signals and observe their history on an Oscilloscope view [21] [22]. We implement a comparable feature: all key telemetry variables (e.g. grid frequency, inverter current, etc.) are treated like probe signals that automatically feed into time-series plots. The **oscilloscope panel** in our app scrolls in real-time, plotting values vs. time, enabling pattern recognition and timing analysis of the inverter's response [23] [24]. This continuous view of data (as opposed to single-point indicators like LEDs or numeric displays) is crucial for debugging control issues and understanding system dynamics.
- **Run Recorder and Replay:** One of RedByte's most powerful features is its deterministic **Run Recorder**, which captures an entire simulation run (initial state, inputs, and all probed signals each tick) for exact replay and verification [25] [26]. We adopt this concept to record HIL test sessions. The

recorder will log the time-series of all telemetry signals, as well as any external events (e.g. a user triggering a scenario or the simulator injecting a fault at a known time). A saved **Session Record** contains the "ground truth" of that test: initial conditions, all input stimuli (e.g. a change in load), and the resulting output trajectories. Upon replay, the software resets the system state and replays the inputs step by step, while monitoring if the outputs match the recorded traces [4] [5]. Under deterministic conditions (same controller firmware and scenario), the replayed traces should exactly overlay the originals [27]. Any discrepancy would indicate a non-deterministic factor or a system change, which is flagged to the user (similar to RedByte's verification highlighting mismatched signals in red [28]). This **replay & verify mechanism** will be invaluable for verifying that code changes in the microcontroller haven't altered its behavior unexpectedly, and for confirming bug fixes (a time-aligned comparison pinpoints the tick where behavior diverged, as RedByte demonstrates [29] [30]).

- **Data Export and Sharing:** RedByte stores projects and run records in a JSON-based format and allows exporting for external use [31] [32]. Our platform similarly will support exporting session data – e.g. a **"data capsule"** JSON containing the telemetry log and meta-data of a test, or conversion to CSV for analysis in tools like MATLAB/Excel. The ability to **share a recorded scenario** with colleagues (who can replay it on their own instance of the software) is a major advantage for collaboration and for compliance with the course's documentation standards.

Using the RedByteOS as a starting point significantly accelerated the design. However, we avoid a one-size-fits-all approach: the system is **highly domain-specific to power electronics**. All applications, nomenclature, and visualizations are tailored to the VSM inverter context. For example, while RedByte's general logic scope shows binary digital waveforms, our scope plots analog values (volts, amps, Hz) with appropriate scaling and units. The file system may store different artifact types (session recordings, configuration profiles, etc.), and the "Terminal" app (if included) might provide a console for interacting with the microcontroller or simulator (e.g. send manual commands), rather than a generic shell. By basing the architecture on a tested framework, we ensure consistency and reliability, but every module is customized or rewritten to serve the specific needs of this project.

Development Platform and Tools

The software is implemented as a **desktop application** using *Python* for backend logic and *PyQt* for the graphical user interface. PyQt was chosen for its ability to create professional, cross-platform desktop GUIs and its rich widget set for plotting and layouts. *PyQt is a Python library for creating GUI applications using the Qt toolkit* [33], which means we can design modern interfaces (with windows, dialogs, menus, etc.) analogous to what RedByte's web UI provides. The use of Python aligns with the team's expertise and allows rapid development, especially leveraging libraries for serial communication (e.g. `pyserial` for UART), data handling (`json`, NumPy/Pandas for processing telemetry), and plotting (PyQtGraph or Matplotlib for the oscilloscope traces).

While RedByte's original implementation is in TypeScript/React (web-based), reimplementing in Python/PyQt offers a **desktop-first experience** out of the box, without requiring a browser or server. The architecture, however, remains modular: we separate the concerns into a **frontend (PyQt GUI)** and a **backend (data ingestion and processing)**. These could potentially communicate via an internal API or signals/slots, which mimics a client-server separation like a SaaS product. This modular design means that, for instance, the data ingestion and logging component could later be replaced with a networked service without changing the UI code – supporting scalability to a cloud or multi-user system if needed.

Other tools integrated into the development include:

- **UART Communication:** We configure a dedicated thread or asynchronous loop to continuously read from the serial port to which the microcontroller is connected. The data format is JSON strings terminated by newline, so the parser simply accumulates a line, parses the JSON, and emits it to the rest of the system. The microcontroller protocol can be extended as needed (for example, to send a "scenario start" event or a sync timestamp). Ensuring thread-safe or signal-slot communication between this reader and the GUI is crucial so that data visualization is smooth.
- **Data Visualization:** For plotting real-time data, we evaluate using **PyQtGraph**, which is a Python graphics library optimized for real-time plotting (capable of handling high update rates and large datasets) [34]. PyQtGraph integrates with PyQt and will allow us to create the oscilloscope-like widgets with multiple overlaid traces, zoom/pan controls, cursors, etc. Alternatively, Matplotlib in interactive mode could be used for simpler plots, though PyQtGraph offers a more interactive experience (important for examining long data records).
- **Opal-RT Interface:** If an Opal-RT simulator is used, it typically provides APIs or interfaces (such as Ethernet-based protocols or shared memory) for control and data access. Our software will not perform the heavy real-time simulation itself but may include a module to send commands to the Opal-RT (like switching a scenario or adjusting a parameter) and possibly subscribe to simulator measurements. For example, an Opal-RT can publish signals to a UDP socket or a shared memory, which we could read to cross-verify the microcontroller's own telemetry. Given OPAL-RT's prominence in HIL testing (used to test and refine complex control systems in real time [9]), we design our system to be compatible, though detailed integration may be part of future extensibility rather than the MVP.
- **Blynk Compatibility:** Since the microcontroller might already be transmitting data to Blynk's cloud (via WiFi or Ethernet), we ensure our serial telemetry does not interfere. Blynk typically uses its own library and protocols; our platform simply reads the microcontroller's USB/UART output, which is separate. In case we wanted to fetch data from Blynk (e.g. to incorporate remote sensor readings or user commands issued from the phone app), Blynk's REST API or MQTT integration could be used. However, this is not in the initial scope. We primarily ensure that **the microcontroller can output telemetry concurrently to UART and Blynk** (for instance, by duplicating writes in code) so that the development team gets the rich local data without losing the convenience of the mobile dashboard. Blynk is known as a *low-code IoT platform that lets users monitor telemetry and control devices from a unified dashboard* [35]; our solution plays a complementary role by focusing on engineering analysis and offline data, whereas Blynk serves remote accessibility.

By combining these tools and frameworks, the development of the platform remains manageable and aligned with the team's skill set. Python/PyQt offers rapid prototyping and a large ecosystem of libraries, which is advantageous given the project's time constraints in an academic setting. Furthermore, a Python-based approach is conducive to feeding into an AI code assistant (such as Anthropic's Claude, as mentioned) to generate boilerplate or repetitive code segments – something we plan to leverage for speed. The detailed design outlined in this paper provides a blueprint that such an AI can follow to produce the initial code for the repository, which the team will then refine and customize.

System Architecture and Functional Decomposition

To meet the diverse requirements, the software platform is organized into a set of functional components, each responsible for a specific aspect of the system's operation. **Figure 2** shows the high-level architecture,

highlighting how data flows from the physical HIL setup into the software and then to various modules (and eventually out to storage or the user interface).

(Figure 2: Software Architecture – Telemetry from the microcontroller (UART JSON) is ingested by the Data Ingestion module, passed to Real-Time Visualization (oscilloscope UI) and recorded in the Data Logger. The Session Recorder manages recording control and interfaces with the Replay Engine for deterministic playback. A Scenario Controller communicates with the HIL simulator (Opal-RT) to coordinate test events. The GUI Shell (RedByte OS core via PyQt) hosts multiple App Windows: Live Monitor, Session Manager, Analysis/Comparison, etc.)

The major functional components are:

- **Telemetry Ingestion Service:** A background service (threaded or asynchronous) that handles all communication with the microcontroller. It opens the serial port, parses incoming JSON lines, and converts them into internal data structures (e.g. a dictionary of signal name -> value). This module time-stamps each data sample as it arrives (to align with simulation time) and immediately forwards the data to two places: (1) the **Real-Time Monitor** for display, and (2) the **Data Logger** for recording. It also monitors the connection health; if data stops or an error is detected (bad JSON or checksum failure), it will notify the UI so the user can take corrective action. This robust ingestion ensures no data is lost and that the rest of the system can treat the telemetry as a live stream of structured data.

- **Real-Time Monitor (Oscilloscope UI):** This is the front-end component that provides live visualization of telemetry. Implemented as an app window (within the RedByte-like desktop), it subscribes to incoming data updates. It contains an **Oscilloscope Widget** that plots selected signals vs. time, updating continuously as new data comes in (with a sliding time window, e.g. last N seconds of data). The user can choose which signals to "probe" (similar to adding probes in RedByte [36]): for instance, one might select grid frequency, inverter output voltage, and battery current to display. Each chosen signal gets an assigned color and trace on the graph. The monitor also includes numeric displays for instantaneous values, and possibly gauge widgets for values like power or temperature if needed. Controls on this window allow pausing the live update (to scroll back and examine a waveform segment) and adjusting the time base or scale. By default, it operates in a **Follow mode** where the timeline scrolls as data comes (like RedByte's follow mode [37]), but users can switch to manual mode to inspect history. This component essentially gives the team the same insight as using an oscilloscope or high-speed data logger, but fully synchronized with the digital system's internal states.

- **Data Logger and Session Recorder:** This component handles the recording of telemetry for replay and analysis. When the user initiates a record (via a GUI control, e.g. a "Record" button on a Session Manager app), the logger begins saving all incoming telemetry into a structured log in memory (or stream to disk for long runs). It records not only the raw signal values at each time step, but also the context metadata of the test – similar to RedByte's Run Record containing initial state and events [38] [39]. In our case, metadata includes the test start time, any scenario ID or description, the firmware version of the microcontroller, and possibly the HIL simulator model version. The logger ensures determinism by capturing **all inputs that the controller sees**: for example, if the HIL simulator injects a disturbance at time t, an entry is logged for that event; if the user pressed a "trip" button in the GUI at time t2, that is logged as well. This way, the session record is a self-contained timeline of "stimulus and response". The Data Logger is tightly integrated with the **Session Manager** app, which provides controls to arm recording, stop, save, or load sessions (akin to RedByte's recorder panel [40]

[41]). The saved session (e.g. as a `.json` file) serves as our **data capsule** concept – it can be reloaded to review the test or fed into the replay engine.

- **Replay Engine and Comparison Tool:** The Replay Engine allows any recorded session to be played back through the visualization and analysis tools. When a user loads a past session and hits "Replay", the engine will feed the data samples (timestamped values) into the system at the same rate they originally occurred. Essentially, it **mimics the microcontroller by sending the logged data through the same pathways** as live telemetry. The Real-Time Monitor doesn't know the difference – it will display the data as if it were happening live. This is crucial for side-by-side comparison: the user can have the system connected to a live run (or a second recorded run) and overlay the signals. For instance, one could replay yesterday's test while the hardware runs today's test, and compare machine speed or output power traces on the same graph. The comparison tool will highlight differences and compute statistics (e.g. the error between two runs). Following RedByte's verification concept [42] [43], if an ideal replay of the same scenario yields mismatches, the tool will flag the time and value of divergence. In practice, due to real-world noise, perfect match isn't expected unless it's a pure simulation replay, but this tool greatly helps quantify variations and detect anomalous behavior. The replay engine also supports stepping through a record tick by tick (or sample by sample) on user command, enabling detailed post-mortem analysis – one can pause at the exact moment something went wrong and inspect all signal values.

- **Scenario Controller:** This module interfaces with the HIL simulator or any external systems needed to control test scenarios. It can issue commands to start or stop the real-time simulation, load different models, or impose events (like "apply a 50% load increase at t = 5s" or "disconnect phase A at t = 2s"). In the absence of an API, this might simply coordinate with the team's procedure (e.g. prompt the user to flip a physical switch). But ideally, through OPAL-RT's API or a custom microcontroller command, scenarios can be orchestrated programmatically. The Scenario Controller ensures that scenario events are also logged via the Data Logger (as they are part of the input stimuli). It might have a small UI of its own – for example, a **Scenario Runner app** where the user selects from predefined scenarios (voltage sag, frequency drop, etc.) and triggers them. The scenario definitions would include expected outcomes or tolerances, which the software could later use to automatically judge if the test passed (this can be part of integration testing, see Test Plan section).

- **RedByteOS Shell / GUI Framework:** This encompasses the windowing system, menu bar, taskbar, and overall user experience of the application. We use a PyQt implementation that replicates key behavior of the RedByte OS environment [12] [13]. There will be a main application window that contains a **desktop area** (perhaps just an MDI – multiple document interface area – where child windows can float). The taskbar can be a Qt toolbar showing open "apps" (Live Monitor, Session Manager, etc.), allowing switching between them. The shell also manages global services like the **File Manager** (opening/saving session files, which might map to a real file system directory or a virtual space), and **Settings** (application-wide settings like communication port selection, UI themes, data directory, etc.). This layer is largely about user convenience – providing a cohesive, familiar environment. The RedByte metaphor of an OS is maintained because it proved intuitive: users can minimize a window, switch tasks, etc., rather than being stuck in one monolithic interface. We will have at least the following "apps" within the shell:

- *Inverter Monitor App:* the live telemetry and control dashboard (oscilloscope and gauges).
- *Session Manager App:* for recording/replaying sessions, managing saved records.

- *Analysis App:* possibly combined with Session Manager, used for comparing runs or computing performance metrics from a log (e.g. calculating frequency nadir, overshoot, settling time from a disturbance).
- *Scenario Editor/Runner App:* optional in MVP, but for selecting and executing test scenarios.
- *Files App:* an interface to browse saved sessions and any other project files (could simply open a folder on disk).
- *Settings App:* to configure user preferences, connection settings, etc.
- *Terminal App:* optional, could provide a direct serial console to the microcontroller (for sending manual commands) or a Python console for advanced users to script interactions.

Each app runs in a window with standard controls (close, maximize, etc.) provided by the PyQt MDI framework. The inter-app communication is via shared backend services or signals. For example, if a user hits "Record" in Session Manager, that triggers the Data Logger (back-end) and also notifies the Monitor app (which might visually indicate recording is on). This architecture ensures **loose coupling** – apps are decoupled from each other's internal logic, talking through defined interfaces, which is beneficial for modularity and future expansion.

- **Data Storage & Export Module:** Responsible for saving session records to disk and exporting data in user-requested formats. When commanded by Session Manager to save, it will take the in-memory log and write it as a JSON file (likely following a schema similar to RedByte's `.json` project format, containing all signal traces and events [31] [44]). It can also convert a session into a CSV (columns for time and each signal) for analysis in Excel, or possibly into MATLAB `.mat` file if needed by the team. For documentation and report purposes, this module might also generate summary reports of a test (for example, computing key statistics and packaging plots), which can aid in weekly logs or the final report. Because data can get large, the module will implement a rolling buffer or compression as needed (especially if recording long runs at high frequency). However, given typical control loop frequencies (on the order of kHz or less) and test durations (seconds to minutes), the data sizes are manageable with modern PCs.

Each of these components is clearly delineated in code (e.g. separate classes or threads) and in the UI. This functional decomposition not only makes development and debugging easier (each part can be developed and tested somewhat independently), but it also aligns with **ECE 358 course expectations for system hierarchy**. The project has been broken down into subsystems, roughly corresponding to "Level 0" (the entire software), "Level 1" (major modules like ingestion, visualization, recording, etc.), and "Level 2" (specific sub-functions like parsing, plotting, file I/O within those modules). Such decomposition facilitates targeted testing (as described in the Test Plan section) and ensures we can reason about the design's completeness and reliability.

Minimum Viable Product (MVP) Scope

Given time constraints of the academic year, we define a clear **MVP scope** that delivers the core value and satisfies the immediate needs of the senior design project, while deferring some non-essential features to future work. The MVP focuses on **essential telemetry monitoring and recording**:

1. **Basic Telemetry Dashboard:** Real-time plotting of at least 4 key signals (e.g. grid frequency, inverter output voltage, inverter current, DC link voltage) with a time window of ~30 seconds. Numeric displays of these values updated in real time. Ability to start/stop the data stream (or pause the

plotting) and clear the traces. This covers the fundamental need for real-time observation of the VSM behavior.

2. **Serial Connection Manager:** A simple UI to select the serial port and baud rate, connect/disconnect, and display connection status. It will attempt to auto-reconnect if connection is lost. This is crucial for usability in the lab environment (where unplugging or resetting the microcontroller can otherwise crash the software).

3. **Recording and Saving Data:** A one-click record button that logs telemetry to memory, and on stop allows the user to save the session to a file. MVP will include saving to our JSON format (with proper structure for later replay). Possibly also a quick export to CSV for immediate use. Deterministic replay verification logic can be simplified in MVP: initially, we ensure we *can* replay data visually, but automated difference checking can be added later if needed.

4. **Session Playback (Offline Mode):** The ability to load a saved session file and replay it on the dashboard as if live. The user should perceive the graphs moving exactly as they did during the original capture. This satisfies the need for post-run analysis without the hardware. MVP may not have dual-run comparison yet, but at least you can view one run's data after the fact.

5. **User Interface Shell:** A basic multi-window interface where the main window is the telemetry dashboard, and a secondary window (or modal dialog) is the session manager (with record, save, load controls). Full OS-like multitasking can be simplified: for MVP, it's acceptable if only one main window is active at a time with some dialogs. The RedByte OS concept can be incrementally implemented – the key is having a clean, navigable UI. A menu bar could provide access to "File > Open Session…", "View > Live Monitor", etc., instead of a fancy taskbar in MVP.

6. **Test Scenario Handling:** For MVP, we might not integrate fully with Opal-RT or automatic scenario scripts. Instead, we ensure the software can at least log an annotation or timestamp when an event is manually triggered. For example, the user could press a "Mark Event" button when they apply a step change externally, logging a marker in the data. This will help during analysis to align with what happened. Full automation (the software commanding the simulator) can be planned for later.

7. **Resilience & Basic FMEA Considerations:** Even in MVP, the software should handle common failure modes gracefully. For instance, if the serial data is malformed (e.g. microcontroller sends partial JSON), the ingestion should skip or request resend rather than crash. If the data rate exceeds what the UI can plot in real-time, it may start decimating or buffering data to maintain responsiveness (ensuring the GUI thread isn't overrun). These align with some FMEA-identified risks (discussed later). MVP will prioritize stable, correct function of core features over exhaustive feature set.

This MVP delivers a working tool that the team can use in their current testing of the inverter. It directly supports the **ECE 358 requirement of prototype demonstration and testing** – by the end of the project timeline, we will have a software prototype that can be shown live, capturing the inverter's behavior under various conditions and helping validate the design.

Crucially, the MVP addresses the **course's IEEE competition expectations** by providing a polished interface and solid functionality that can be presented to judges or in a paper. In fact, many IEEE Student Hardware Competition entries benefit from having a custom monitoring tool to showcase real-time data – our

platform fulfills that role, demonstrating professional-level software integration in an undergraduate project.

Extensibility and Future Work

While the MVP is sufficient for immediate needs, the architecture is deliberately designed for **extensibility**. Future iterations or follow-on teams could enhance the platform in numerous ways:

- **Advanced Analysis Modules:** We envision adding dedicated analysis tools, for example: a *Harmonic Analysis* module that computes FFT of the inverter output to analyze power quality, or a *Stability Analyzer* that automatically computes the damping ratio or frequency nadir from a disturbance response. These could be separate apps or integrated into the analysis view, and they would use the recorded data to output engineering metrics. Given that RedByte's environment supports adding new apps easily [45] [18], our platform could similarly accept plugins.

- **Automated Test Sequences:** Building on the Scenario Controller, a future version could allow scripting a sequence of actions. For instance, a user could write a script (in Python or a simple sequence language) to: set initial condition, run for 5 seconds, introduce a fault at 5s, wait till 10s, then stop. The software would execute this on the HIL and possibly on a purely software simulation for comparison. This moves the tool closer to a full testing framework, where a batch of tests could run overnight and results be collected.

- **Integration with Version Control and Continuous Integration:** For Cyber Engineering and robust design, one might integrate this platform with the firmware's version control. Each test record could automatically tag the microcontroller firmware version (from Git commit, for example). If a CI pipeline is in place, after each firmware build, the system could run a standard set of HIL tests via our software and report if anything regressed. This is an ambitious extension but very much aligned with professional practices (treating the HIL tests like unit tests for the integrated system).

- **Security Features (Cyber Engineering Focus):** As a Cyber Engineering student's project, one could extend the platform to also monitor for cyber-physical security anomalies. For example, include a module that watches for unusual patterns (maybe comparing expected vs actual behavior in real-time) that could indicate a fault or intrusion. If a malicious actor tried to tamper with sensor signals, the software might detect the deviation. This goes slightly beyond our initial scope, but it's a nod to the cybersecurity aspect of a Cyber Engineering curriculum. The modular design allows insertion of such monitoring without overhauling the core.

- **SaaS Deployment and Collaboration:** Although currently desktop-only, the SaaS-like modularity means we could deploy components on a server. A future cloud version could allow multiple team members to view the telemetry remotely in a web browser (essentially running the UI as a web app and streaming data over WebSockets). Additionally, data capsules could be uploaded to a team server for centralized storage and comparison. This would be useful for an extended project or if multiple teams worked on similar hardware – they could share and compare results easily. Since the UI is built in Qt, an alternate front-end (web) could mimic it by using a similar model-view separation in the code.

- **Hardware Control Extensions:** Right now, the platform mainly listens and visualizes. In future, it could actively control hardware: for example, allow sending commands to the microcontroller (tuning a parameter on the fly, or switching modes). A control panel could let an operator change the VSM's virtual inertia or damping coefficient in real-time to see the effect, which would be great for experimentation. This would involve defining a protocol (perhaps extending the UART JSON to also accept commands) and building UI elements for control knobs/sliders.

All these potential extensions would benefit from the strong foundation laid by the current design. The use of RedByteOS principles means adding a new feature is often a matter of adding a new app or module without disturbing existing ones – illustrating **open-closed principle** in design (open for extension, closed for modification). This future-proofing is important not just technically, but also for the project's **competition and publication potential**. An IEEE paper or competition demo could highlight how the system is scalable and not a dead-end prototype. It shows the team considered maintainability and future needs, which is a mark of engineering professionalism.

Test Plan and Validation Strategy

To ensure the platform meets its requirements and functions reliably, we developed a comprehensive **test plan** covering unit tests for each subsystem and integration tests for the whole system. This approach aligns with ECE 358's emphasis on defining explicit, quantitative test cases for all subsystem levels [46] . Key elements of the test plan include:

- **Unit Testing of Subsystems:** Each major module identified in the functional decomposition is unit-tested in isolation using simulated inputs:
- *Telemetry Parser Test:* We feed the Telemetry Ingestion module a series of known JSON strings (including edge cases like malformed JSON, boundary values, very rapid bursts) and verify it correctly outputs parsed data or error flags. For example, an input `{"freq":60.0,"Vabc":[120,119,121]}` should result in internal structures with freq=60.0 and the three-phase voltages as given. We also test behavior at extreme baud rates and ensure no memory leaks when data flows continuously.
- *Real-Time Plotter Test:* Using a test harness, we simulate a stream of values into the plotting widget (bypassing the serial) to ensure it can render at the required rate (e.g. 100 samples per second) without lag. We measure the update latency and confirm that the last 500 samples are retained (the oscilloscope's circular buffer length, similar to RedByte's default [47]). We also test that pause and resume functionality works – when paused, the plot stops scrolling and the user can inspect old data; on resume it jumps to live (like the "Follow Now" button effect [48]).
- *Recorder/Playback Test:* We programmatically generate a fake "session" – for instance a simple waveform like a 1 Hz sine wave for 10 seconds – and run it through the Data Logger as if recording. Then feed the recorded data into the Replay Engine and check that the values output match the original within an acceptable tolerance (accounting for any floating-point or timing minor differences). Ideally, for a deterministic stream, the comparison should show exact match (which would be indicated by our tool with a success message [49]). We intentionally introduce a difference (e.g. alter one sample in the record) to test that the verification flags a mismatch at the correct time index.
- *File I/O Test:* Save a known small session to file, then load it back and confirm the data integrity. Also test exporting to CSV yields correct formatting. Additionally, ensure that if the application tries to load a corrupted or incompatible file, it handles it gracefully (error message, not crash).

- *GUI/OS Shell Test:* Verify that multiple windows can be opened, moved, minimized without error. Ensure menu actions trigger the right responses (e.g. “Open Session” opens a file dialog). These tests are partly manual (UI interaction testing) and partly automated with Qt’s test functions or simply careful code review, since GUI logic can be hard to fully automate.

- **Integration Testing:** These tests involve running the software end-to-end in various scenarios:

- *Live Data Integration Test:* Connect the software to the actual microcontroller (or a simulator that generates similar telemetry) and run a simple scenario. For example, start the inverter in a steady state and then cause a frequency step change in the HIL. We expect to see the frequency trace in the UI jump and recover. We will compare the values shown on our software vs. an independent reference: the Blynk dashboard or the Opal-RT console reading the same values. They should match closely (within sensor tolerances and data rate differences). This validates that our ingestion and parsing pipeline works in the real environment.
- *Timing and Performance Test:* Measure the end-to-end latency from when the microcontroller sends a data packet to when it’s plotted on screen. This can be done by toggling a digital output on the microcontroller whenever it sends a packet and simultaneously logging a timestamp in the software when the packet is processed, then using an oscilloscope or logic analyzer to measure the difference. Our goal is to keep this latency low (ideally a few milliseconds, definitely under 100ms) so that “real-time” feedback feels instantaneous. We also test the system under maximum data load – e.g. if microcontroller increases telemetry rate or sends many signals (to approach the worst-case of our design). The software should cope without crashes; if necessary it might drop frames but should recover and never hang the UI thread.
- *Scenario Replay Consistency Test:* Conduct a back-to-back experiment: run a particular test scenario on the hardware twice and record both runs. Then use the software’s comparison tool to overlay the two runs. Since they are two separate executions of the physical system, they won’t be identical, but we expect them to be **consistent within certain error bounds**. For instance, in two runs of a load step, the frequency dip might differ by <0.1 Hz. Our comparison tool can automatically compute the difference and confirm it’s within an expected tolerance. This not only tests our software but also validates the repeatability of the physical experiment – a requirement in the **test plan for statistical validation of subsystem functionality** for EEs [50] . In other words, this platform helps fulfill the EE majors’ requirement of analyzing experimental variance by providing them the data to compute average, standard deviation etc., and our comparison outputs can directly assist with those calculations.
- *Failure Mode Injection Test:* As part of ensuring reliability, we simulate or induce certain failure modes to see how the software responds (in line with a design FMEA approach). For example, disconnect the serial cable during operation: the software should detect loss of data and alert the user, then attempt reconnection (and not freeze). Or feed an incorrect JSON key (simulate a firmware bug sending a wrong field): the software might ignore unrecognized fields but continue running. Another example: run the disk out of space while recording – ensure the software at least warns and stops gracefully rather than corrupting the whole file. These tests are derived from our FMEA analysis, where we identified potential failures and mitigations. For instance, a potential failure “Data Overflow causes memory crash” is mitigated by using bounded buffers and will be tested by long-duration runs.

The test plan is documented in a tabular form per ECE 358 guidelines, with each test case specifying the subsystem under test, the specific input conditions, and the expected output (including **numerical tolerances for min/typical/max results** as applicable [51] [52]). For example, a test case for the Telemetry

Parser might specify: *Input:* JSON string with 10 fields at 115200 baud; *Expected Output:* All 10 fields parsed correctly (typical), at most 1% CPU usage (max), no missed messages (min requirement 0 lost). By quantifying expectations, we make it clear what constitutes a pass or fail.

One particular course requirement is the inclusion of **Cyber Engineering-specific test cases**, which means setting up experiments to validate system or subsystem functionality [53]. In our context, an example could be an experiment to validate the **determinism of the replay feature** (which is essentially a software subsystem test, but executed as an experiment). We might also craft a cyber-physical security test: for instance, intentionally spoof one of the sensor readings in the HIL (simulate a bad actor or sensor fault) and see if the software can log and perhaps flag the inconsistency (this ties into future security features). While security is not the main focus of this project, demonstrating that we considered and tested for abnormal conditions reflects the CyEng perspective of ensuring system trustworthiness.

All tests and results are being logged in the team's **weekly progress reports**, which is another ECE 358 expectation. Each week, the team documents what tests were run, their outcomes, and issues found (if any). This practice of maintaining **weekly logs** demonstrates professional project management and communication: it ensures transparency with advisors and helps the team reflect on progress. By the end of the project, these logs show the evolution of the system's reliability and feature set. For example, Week 6 log might note: "Integration Test #3 (live data) passed: frequency response captured correctly on software, minor timing offset of 20 ms observed – will calibrate timestamps next week." This continuous logging and refinement loop is a concrete application of professional engineering practice taught in the course (aligning with learning outcomes on effective communication and teamwork).

Finally, a **Design Failure Modes and Effects Analysis (FMEA)** was conducted early in the design phase to preemptively address reliability concerns. The FMEA table (based on the provided template [54] [55]) lists potential failure modes, their causes, and effects for both the software and its interaction with hardware: - One example is *"Telemetry data loss or corruption"* which could be caused by buffer overflow or serial noise. The effect would be inaccurate analysis or missed events. Our mitigation (and design choice) was to use line-based protocol with checksum and to implement buffering with flow control (so the microcontroller can slow down if the PC is overwhelmed). The RPN (Risk Priority Number) for this was initially moderate, but with mitigations, we reduced likelihood significantly. We plan tests specifically to verify these mitigations (as discussed above). - Another example: *"UI freeze during long run"* due to main thread blocking (cause: heavy file I/O or plotting on UI thread). Effect: user loses visualization, potential data loss. Mitigation: move recording and heavy processing to background threads and ensure thread-safe design. Testing involved simulating long runs to confirm UI remains responsive. - *"Replay mismatch despite identical inputs"* could indicate a hidden nondeterministic behavior (perhaps a race condition in code). This would undermine trust in the tool. Mitigation: rigorous debugging and using the verification feature to catch any divergence as a bug to fix. Test: replay back-to-back known deterministic input and ensure exact match (which we have in the plan).

By addressing these in design and validating through tests, we increase confidence that the platform will function smoothly during the critical demos (e.g., final presentation or IEEE contest). The FMEA process also shows our commitment to **risk management**, a topic emphasized in senior design. We have documented how each high-severity risk is handled, which will be included in the final project documentation.

Course Requirements and Professional Practice

Throughout the project, we kept alignment with the **ECE 358 Senior Design Laboratory** course requirements, ensuring that our work is not only technically sound but also well-communicated and professionally executed:

- **Test Plan Development:** As detailed, we produced a thorough test plan, iteratively refined with instructor feedback. This satisfies the requirement of a final test plan that facilitates unit and integration testing of all subsystems [56] [57]. By including quantified conditions and expected outputs, and addressing the Cyber Engineering experiment criteria, our test plan is positioned to score full points in all categories (comprehensiveness, specificity, expected results, and CyEng special requirements) [58] [50].

- **Weekly Logs and Communication:** We maintained a project journal with entries for each week's meetings and progress. These logs include what each team member (software vs. hardware) accomplished, any challenges, and plans. This practice not only keeps the team synchronized (promoting teamwork and leadership skills as per course outcomes [59]) but also serves as a communication tool to advisors. We often included screenshots from our software in these reports to visually communicate our progress. Additionally, as the software lead, Connor ensured that complex software concepts were explained in accessible terms for the electrical teammates, honing the skill of effective peer communication (an outcome of the Professional Communication curriculum [60]). For instance, explaining how the "time-travel debugging" works in our tool in a way that a non-software person appreciates its value.

- **IEEE Competition Preparation:** The course outline explicitly mentions preparation for an IEEE technical presentation and student paper contest [61] [62]. We incorporated this by writing our documentation (including this paper) in a format adhering to IEEE conference standards – clear sectioning, formal tone, and citing relevant work. The software platform itself gives us a strong demonstration piece for competitions: its polished GUI and real-time capabilities make the project stand out beyond a typical hardware prototype. We plan to use it during the final presentation to show live data and possibly run a mock demonstration of a microgrid event. This level of integration and polish addresses the IEEE review criteria of innovation, technical excellence, and quality of presentation.

- **Professional Ethics and Responsibilities:** Although less directly applicable to the software design, we touched on ethical considerations like data integrity (ensuring our tool does not misrepresent data) and acknowledging limitations (for example, if our software had a bug, being transparent about it in logs and working to fix it). These relate to the course's learning outcomes about applying ethical principles and professional behavior [63] [64]. In practical terms, this meant careful testing before using the tool to make engineering decisions, and not over-relying on unverified outputs.

- **Interdisciplinary Collaboration:** This project required close collaboration between the cyber (software) side and the electrical (hardware) side. We made use of professional communication practices, holding weekly meetings with set agendas, using shared documents for requirements and test plans, and version control (GitHub) for managing the software code. The use of a common platform (the HIL tool we built) actually acted as a communication bridge – for instance, when Hannah and Antreas wanted to convey an electrical issue (like a noise in current signal), seeing it on

the software's plot helped Connor understand and respond from the software angle (maybe adding a filter or noting it as a hardware artifact). This synergy exemplifies the "competence with communication skills needed to engage in diverse personal, social, and civic relationships" outcome that the course as part of the LS core aims to fulfill [60] (interpreting here the diverse team roles as part of that outcome).

- **Documentation (Final Report & FMEA):** We used guidance from Gannon's provided materials, such as the FMEA lecture [65] [66] and template, to produce a Design FMEA document that will be included in our final report appendix. Moreover, this research paper itself is written to double as a section of our final design document, covering software design and integration. By structuring it like a research-grade technical paper, we ensure it meets the expectation of a "complete design document" with professional language and thoroughness [67]. This will be reviewed by faculty and possibly industry judges, so we have aimed for clarity, coherence, and technical depth.

In conclusion, the development of the RedByte-derived HIL telemetry platform has been an educational capstone experience in itself – not only have we engineered a complex system, but we have also practiced the full engineering project cycle from requirements, design, and implementation to testing, documentation, and presentation. The resulting software is poised to significantly aid the validation of our VSM inverter project, providing immediate insight into system behavior and a means to rigorously test and demonstrate our design. We are confident that this platform meets the needs of our senior design project and exemplifies the integration of hardware and software in modern power engineering solutions. By leveraging an existing innovative software stack and tailoring it to a new domain, we achieved a robust solution efficiently – a valuable lesson in reusing and adapting technology. We look forward to deploying the tool in our final testing phase and showcasing it during our final presentation and any competitions, where it will underscore the professionalism and innovation of our team's work.

References:

- RedByte User Manual excerpts for OS metaphor, oscilloscope, and recorder features [14] [4] [21] [5].
- Gannon University ECE 358 course documents for test plan and FMEA guidelines [46] [53].
- MDPI Engineering article defining Virtual Synchronous Machines [1].
- Typhoon HIL application note on VSG demonstrating HIL SCADA interface [3].
- OPAL-RT real-time simulator description [9].
- Blynk IoT platform description [6].
- PyQt library description (Python Qt GUI toolkit) [33].

---

[1] [7] [8] Control Algorithm for an Inverter-Based Virtual Synchronous Generator with Adjustable Inertia
https://www.mdpi.com/2673-4117/6/9/231

[2] [3] Grid-connected inverter with virtual synchronous machine
https://www.typhoon-hil.com/documentation/typhoon-hil-application-notes/References/virtual_sync_machine_inverter.html

[4] [5] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [47] [48] [49] REDBYTE_USER_MANUAL.md
https://github.com/swaggyp52/redbyte-ui-genesis/blob/b2ebd0d151efa9d8c6d6cd05b134eb7d41d4233e/REDBYTE_USER_MANUAL.md

6 Blynk: Low-Code IoT Platform | Apps, Cloud & Device Management
https://blynk.io/

9 OPAL-RT | Trusted Real-Time Simulation Solutions | Discover OPAL-RT Technologies for Innovation | Explore OPAL-RT Simulator Systems | OPAL-RT
https://www.opal-rt.com/

33 PyQt5 Tutorial 2026, Create Python GUIs with Qt
https://www.pythonguis.com/pyqt5-tutorial/

34 PyQtGraph - Scientific Graphics and GUI Library for Python
https://www.pyqtgraph.org/

35 IoT Web Console | Manage Devices, Data & Users - Blynk
https://blynk.io/platform/web-console

46 50 51 52 53 56 57 58 HW05_FinalTestPlan.docx
file://file_000000001dcc7230b0553307e52d1bed

54 55 65 66 LEC01_FMEA.pdf
file://file_00000000f21071fdb3267d786fadb4dd

59 60 61 62 63 64 67 ECE358_26SP_LS Core_Zhao.docx
file://file_00000000384471fd864ec398914e8c8f