ERP-Connected Edge Telemetry with Qt/C++ on Firefly: Serial COM Aggregation, SQLite Caching, and Cloud Control

Alperen Kayhan
alperen-1393@hotmail.com
https://github.com/AlperenKayhan

***Abstract*—This paper presents an edge telemetry system where a Firefly EK-3566 single-board computer runs a Qt/C++ application that ingests serial (COM) streams from ESP32 sensor nodes, persists measurements locally in SQLite, and synchronizes with an *ERP* (IoT/Industrial Remote Platform) backend over secure WebSocket. The client performs session initialization via HTTPS, heartbeat-based liveness with command handling, hot-plug detection for serial ports, and a simulation mode for testing. The design balances local resilience (offline-first with a durable cache) and cloud control (remote commands, parameter queries, and log uploads), and can be deployed as a portable executable on devices without a full Qt SDK.**

I. Introduction

Modern field systems often require real-time sensing with robust connectivity to a cloud platform while tolerating intermittent networks. We implement an edge gateway that *locally* buffers sensor values and *remotely* participates in command-and-control. The gateway runs on a Firefly EK-3566 and aggregates multiple serial feeds from ESP32-based sensor nodes. A Qt/C++ application stores readings in SQLite and mirrors events to an ERP backend over a WebSocket. The result is a maintainable, cross-platform solution with a simple operator UI.

II. What is ERP and How We Use It

In this work, **ERP** denotes our IoT/Industrial Remote Platform: a backend that provides device session management, telemetry ingestion, and remote commands. The client first acquires or resumes a session token over HTTPS, then upgrades to a TLS WebSocket (Socket.IO compatible) and registers the device identity. LikeWise;

- `send_logs`: upload local logs (rows from the SQLite cache).
- `get_d_parameters`: echo device/network parameters (IP, MAC, IDs).
- `reboot` / `refresh`: reset state or clear local DB.
- `changed_parameters`, `ping`: toggle simulation or heartbeat control.

Our client acknowledges heartbeats (*ping/pong*), applies commands, and on each heartbeat—if "Sensor Reading" is enabled—commits a computed record to SQLite (either from real serial data or from a controlled simulation mode).

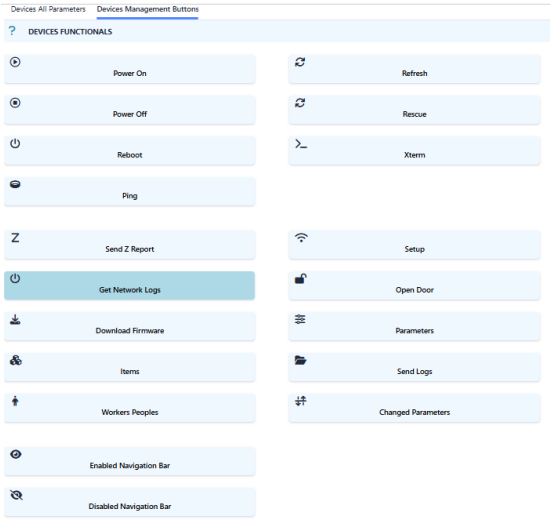

Fig. 1: DeviceManagementButtons

III. Hardware Overview

A. Firefly EK-3566 Edge Host

The Firefly EK-3566 (RK3566 SoC) runs the Qt application. It connects to one or more USB-to-UART dongles that expose sensor nodes as COM devices. The host provides:

- Sufficient CPU/GPU for Qt GUI rendering.
- Stable USB host ports for multiple serial adapters.
- Ethernet/Wi-Fi for HTTPS and WSS connectivity to ERP.

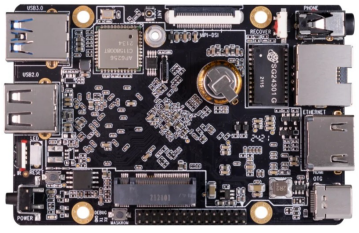

Fig. 2: Firefly ROC-RK3566-PC

Manuscript prepared August 2025.

B. ESP32 Sensor Nodes (Serial Senders)

Each ESP32 reads a sensor (e.g., ultrasonic distance) and periodically outputs a single ASCII float in centimeters followed by newline, This way we also, practice the fundamental process of raw data, due to separation between new line and read.

```
0.37\n
0.09\n
...
```

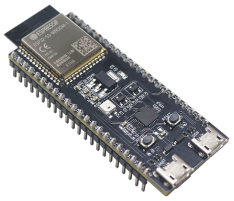

Fig. 3: ESP32-S3-WROOM-1

The Firefly interprets each line as one measurement. This simple, line-delimited protocol is robust and tooling-friendly. Moreover, we utilize the HC-SR04 Ultrasonic Sensor to make our readings to obtain our distance value between the fan blade and the sensor.

Fig. 4: HC-SR04 Ultrasonic Sensor

IV. Local Data Store: SQLite

We use SQLite as a durable, file-based cache. The schema is intentionally small:

```
CREATE TABLE IF NOT EXISTS warnings (
  id       INTEGER PRIMARY KEY AUTOINCREMENT,
  timestamp TEXT NOT NULL,
  level    TEXT  NOT NULL,
  distance REAL NOT NULL,
  xn       REAL  NOT NULL
);
```

For each accepted heartbeat, the client inserts a row with the latest distance and a derived state variable X_n (Section VI). Then, we process that data to generate our console log, which we send to our ERP system. Thus, our log results will recorded as ".json" file and it look like this;

V. What is Qt and Why We Use It

Qt is a mature C++ framework for cross-platform applications. Key features we leverage:

- **Signals/Slots** for decoupled event handling across threads.

Fig. 5: Sample Log .json Result

- **QtSerialPort** for COM I/O, **QtNetwork** for HTTP-S/WSS, and **QtSql** for SQLite.
- **QtWidgets** for a lightweight operator UI (table, buttons, 3D scatter).

Qt's portability lets us develop on Windows and deploy to ARM Linux on the Firefly with minimal changes.

VI. Software Architecture and Qt Implementation

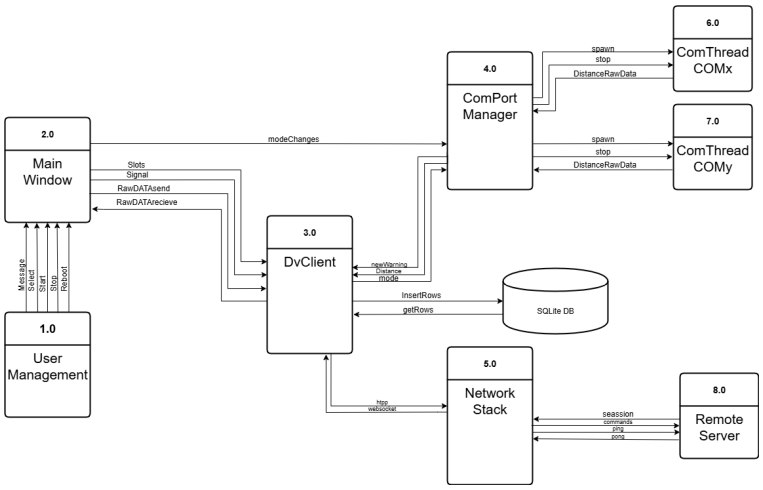

Fig. 6: Data Flow Diagram level-o

A. Modules

DvClient (main-thread):

- HTTPS session (`DvOp`) and WSS connection (Socket.IO frame).
- Heartbeat handler: on `pong`, if "Sensor Reading" is enabled, commit a record from either real serial input (d) or simulated value.
- Command handlers: `send_logs`, `get_d_parameters`, `reboot`, `changed_parameters`, `refresh`, `ping`.

- SQLite operations: schema init, insert, and log export (JSON upload).

ComPortManager (main-thread):

- Modes: *Idle*, *SimulationOnly*, *SinglePort*, *AllPorts*.
- Starts/stops `ComThreads`; rescans on "Reboot"; emits *anyPortOpened/allPortsClosed*.
- Auto-stop: if all ports close (e.g., unplug), turns off "Sensor Reading".

ComThread (worker thread per COM):

- Owns a `QSerialPort` *moved into the worker thread*.
- Blocking loop with `waitForReadyRead(100)`; accumulates bytes, splits by `\n`, parses to `float`.
- `errorOccurred` handler stops the loop on unplug/resource error; clean close.

MainWindow (UI):

- Port dropdown: *Select Port* (Idle), *Simulation*, *All Ports*, or a specific COM.
- Buttons: Start/Stop Sensor Reading, Reset DB, Send Logs, Get Parameters, Reboot (rescan).
- Table view bound to SQLite and a 3D scatter for quick visualization.

B. COM and Thread Logic (Detailed)

Threading model. Qt's main/GUI thread hosts the UI, `DvClient`, and `ComPortManager`. Each active serial port runs in its own `ComThread` (a `QThread` subclass). Signals from workers are auto-queued to the main thread.

Opening and switching ports. When the user picks a different COM, `ComPortManager`:

1) Calls `clearAll()`: `stop()` each worker, `wait()` for exit, delete threads.
2) Starts new thread(s) for the selected mode.

This guarantees the old port is closed before the new one opens.

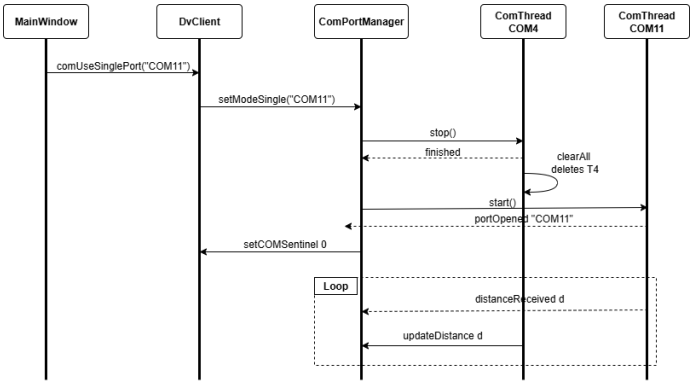

Fig. 7: Sequence Diagram of COMs

Unplug behavior. In the worker, `QSerialPort::errorOccurred` sets a running flag false on *ResourceError/ReadError/DeviceNotFound*, causing the loop to exit within $\leq$100 ms. The manager observes all workers closed and auto-disables "Sensor Reading" in non-simulation modes.

Stale reading guard. The main thread records a timestamp on every `distanceReceived`. If no fresh serial value arrives for a short window, heartbeats in non-simulation mode skip inserts. This avoids re-using the last distance after unplug.

C. Deriving the Warning Level

Given the most recent distance d (cm), we compute

$$X_n = \mathrm{fmod}(7 \cdot (10d) + 3,\, 4)\,,$$

then bucket into levels:

`WARNING-1` $(\mathbf{X}_n \leq 1.5)$, `WARNING-2` $(X_n \leq 2.1)$, `WARNING-3` $(X_n \leq 3.1)$, `WARNING-4` otherwise. Each heartbeat (when enabled) persists $(\texttt{timestamp}, \texttt{level}, d, X_n)$ to SQLite and emits a UI update.

VII. ESP32 Message Format and Rate

Each node outputs an ASCII float with newline; example:

```
0.37\n
0.09\n
```

Recommended rate: 5–20 Hz per port. Keep lines $<$32 bytes and end with `\n` to align with the Qt parser.

VIII. Operator Workflow

1) Launch app; dropdown shows *Select Port* (idle).
2) Choose *Simulation*, *All Ports*, or a specific COM.
3) Press *Start Sensor Reading* to begin committing rows on heartbeat.
4) *Reboot* rescans ports (e.g., when plugging a new device).
5) Unplugging all ports auto-stops reading in non-sim modes.

IX. Deployment Notes

A. Windows

Build Release and run `windeployqt`:

```
windeployqt --release --compiler-runtime QtAlp.exe
```

Place OpenSSL 3 DLLs (`libssl-3-x64.dll`, `libcrypto-3-x64.dll`) beside the EXE for HTTPS/WSS.

B. Linux on Firefly EK-3566

Deploy the executable with Qt 6 runtime libraries and plugins (platforms, SQL/SQLite). Tools like `linuxdeploy-qt` can bundle the dependencies, or install Qt runtime packages on the device. Ensure serial devices (e.g., `/dev/ttyUSB*`) are accessible by the user (`udev` rules or dialout group).

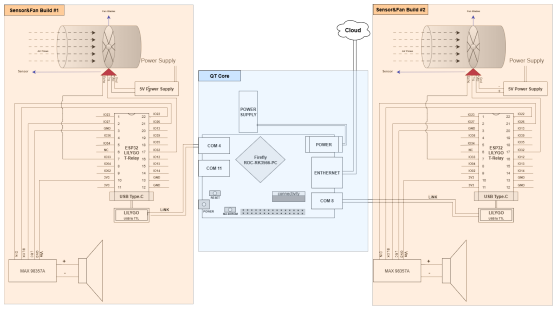

Fig. 8: Sample Log .json Result

X. Design

XI. Conclusion

The presented system combines a Qt-based edge client with serial aggregation, an offline-capable SQLite cache, and an ERP backend channel for control and telemetry. The architecture supports hot-plugging, safe thread management, and a simulation path for testing, making it practical for field deployments that need both resilience and remote control.

XII. Sample Run

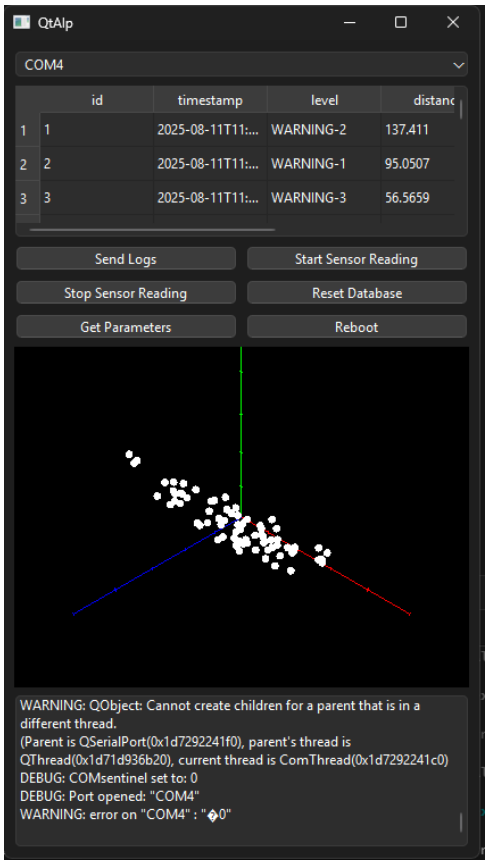

Fig. 9: Sample Log .json Result

Video-Link: https://www.youtube.com/shorts/tWgTr-uwxCM

Acknowledgment

Thanks to the IoT Backend Systems team for support and feedback.