Software Design Document

Telemetry Dashboard

Real-Time Vehicle Telemetry Visualization System

Version:	1.0
Date:	December 26, 2025
Organization:	ASU Racing Team
Platform:	Qt 6 / QML

Contents

Chapter 1

System Architecture

1.1 Overview

The Telemetry Dashboard is a real-time vehicle data visualization application built using the Qt 6 framework with QML for the user interface. The system receives telemetry data from a racing vehicle through multiple communication protocols and displays it in an intuitive dashboard interface.

1.2 Architectural Pattern

The application follows the **Model-View-ViewModel (MVVM)** architectural pattern, which is the natural fit for Qt/QML applications:

- **Model Layer**: C++ Controllers located in `src/Controllers/`
- **View Layer**: QML Components located in `src/UI/`
- **ViewModel Binding**: Qt's Q_PROPERTY system with signals and slots

1.3 High-Level Architecture

The system is designed with a clear separation between data acquisition (Controllers) and data presentation (UI). The `CommunicationManager` acts as a facade, providing a unified interface to the QML layer while abstracting the complexity of multiple communication protocols.

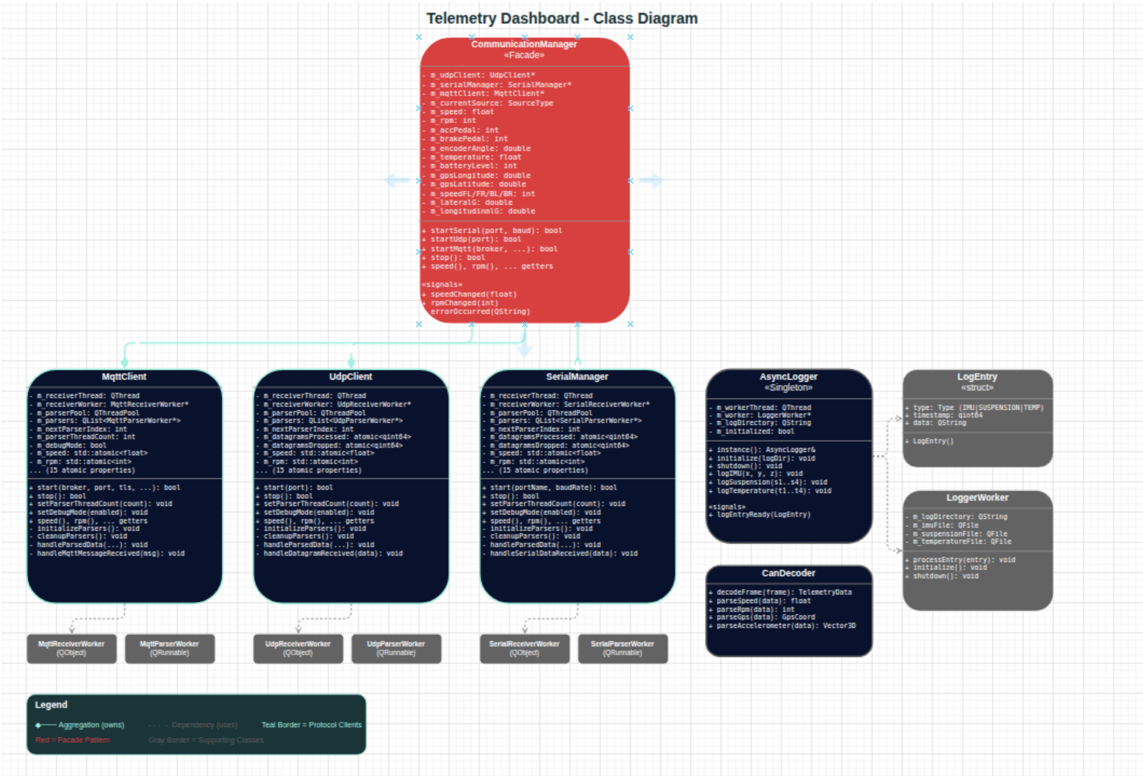

Figure 1.1: System Architecture Overview - Class Diagram

1.4 Data Flow

1. **Data Acquisition**: Raw telemetry data is received via UDP, Serial, or MQTT protocols
2. **Parsing**: Dedicated worker threads parse the incoming data streams
3. **Aggregation**: `CommunicationManager` aggregates data from all sources
4. **Property Binding**: Q_PROPERTY notifications trigger automatic UI updates
5. **Visualization**: QML components render the telemetry data in real-time

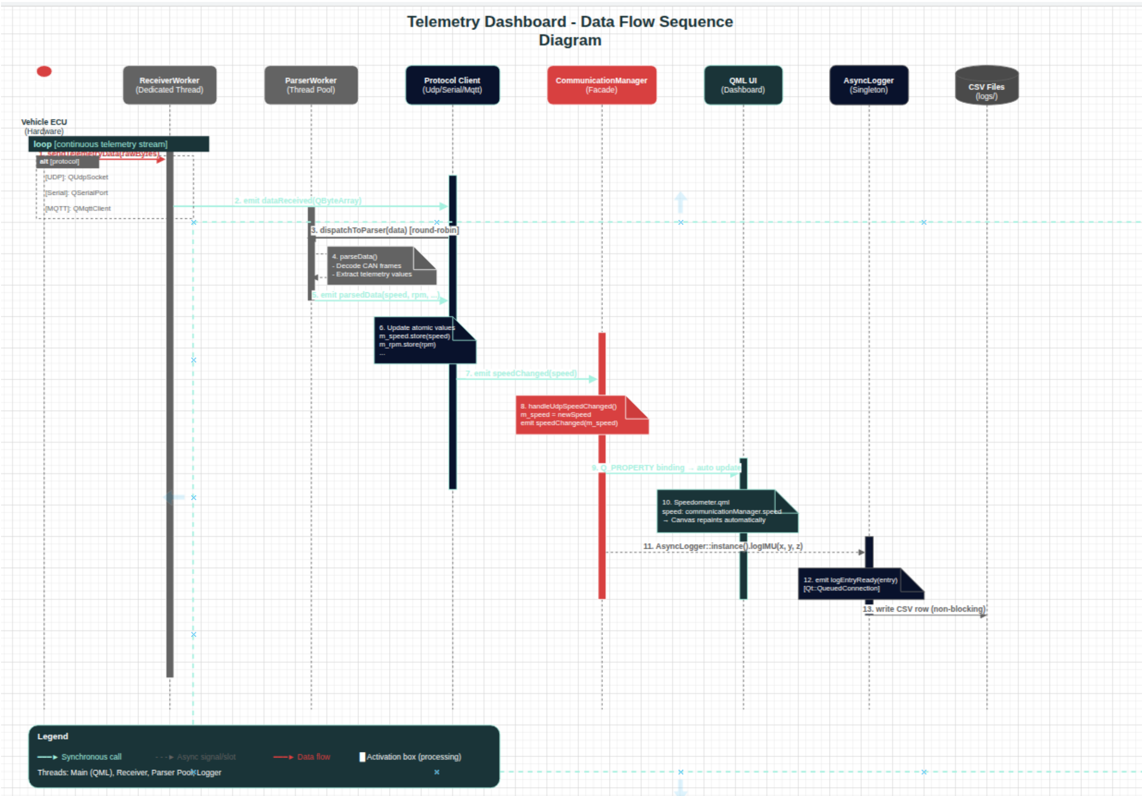

Figure 1.2: Data Flow Sequence Diagram

1.5 Project Structure

```
Car_Dashboard/
|-- Assets/                               # Images and resources
|    |-- racinglogo.png                   # Team logo
|    |-- GG_Diagram.png                   # G-force diagram background
|    |-- Steering_wheel.png               # Steering wheel image
|    +-- ...
|-- src/
|    |-- Controllers/                     # C++ backend logic
|    |    |-- can/                        # CAN bus decoder
|    |    |-- communication_manager/      # Central facade
|    |    |-- logging/                    # Async CSV logger
|    |    |-- mqtt/                       # MQTT client
|    |    |-- serial/                     # Serial port client
|    |    +-- udp/                        # UDP client
|    +-- UI/                              # QML frontend views
|         |-- InformationPage/            # Dashboard components
|         |-- StatusBar/                  # Top status bar
|         +-- WelcomePage/                # Session setup screens
|-- main.cpp                              # Application entry point
|-- Main.qml                              # Root QML component
+-- CMakeLists.txt                        # Build configuration
```

1.6 Threading Model

The application employs a sophisticated multi-threaded architecture to ensure responsive UI while handling high-frequency telemetry data:

- **Main Thread**: Qt event loop and QML rendering
- **Receiver Threads**: Dedicated threads for each protocol (UDP, Serial, MQTT)
- **Parser Thread Pools**: QThreadPool-based concurrent parsing
- **Logger Thread**: Dedicated thread for async file I/O

Chapter 2

Component Analysis

2.1 Controller Module

The Controller module (`src/Controllers/`) contains all backend logic for data acquisition, parsing, and logging. Each component follows a consistent pattern with header files in `include/` and implementation files in `src/`.

2.1.1 CommunicationManager

Location: `src/Controllers/communication_manager/`

Purpose: Central facade that abstracts multiple data sources and provides a unified interface to the QML layer.

Design Pattern: Facade Pattern

Key Features

- Aggregates `UdpClient`, `SerialManager`, and `MqttClient`
- Exposes unified Q_PROPERTY interface to QML (15 telemetry properties)
- Handles source switching via `SourceType` enum (`None`, `Serial`, `Udp`, `Mqtt`)
- Signal forwarding from all protocol handlers

Exposed Properties

Property	Type	Description
`speed`	`float`	Vehicle speed (km/h)
`rpm`	`int`	Engine RPM
`accPedal`	`int`	Accelerator pedal position (%)
`brakePedal`	`int`	Brake pedal position (%)
`encoderAngle`	`double`	Steering wheel angle (degrees)
`temperature`	`float`	Temperature reading (°C)
`batteryLevel`	`int`	Battery state of charge (%)
`gpsLongitude`	`double`	GPS longitude coordinate
`gpsLatitude`	`double`	GPS latitude coordinate
`speedFL`	`int`	Front-left wheel speed
`speedFR`	`int`	Front-right wheel speed
`speedBL`	`int`	Back-left wheel speed

Property	Type	Description
`speedBR`	`int`	Back-right wheel speed
`lateralG`	`double`	Lateral G-force
`longitudinalG`	`double`	Longitudinal G-force

Q_INVOKABLE Methods

```
Q_INVOKABLE bool startSerial(const QString &portName, qint32
    baudRate);
Q_INVOKABLE bool startUdp(quint16 port);
Q_INVOKABLE bool startMqtt(const QString &brokerAddress, quint16
    port,
                           bool useTls, const QString &clientId,
                           const QString &username, const QString
     &password,
                           const QString &topic);
Q_INVOKABLE bool stop();
```

2.1.2 MqttClient

Location: `src/Controllers/mqtt/`

Purpose: High-performance MQTT broker connection for wireless telemetry.

Threading Model

- **MqttReceiverWorker**: Dedicated thread for message reception from MQTT broker
- **MqttParserWorker**: Thread pool with round-robin message distribution

Key Features

- TLS/SSL support for secure connections
- Configurable parser thread count (default: CPU core count)
- Atomic data storage (`std::atomic<T>`) for thread safety
- Performance tracking (messages processed/dropped counters)
- Topic subscription with QoS configuration

Class Structure

```
class MqttClient : public QObject
{
    Q_OBJECT
    // 15 Q_PROPERTY declarations...

public:
    explicit MqttClient(QObject *parent = nullptr);
```

```
    ~MqttClient();

    Q_INVOKABLE bool start(const QString &brokerAddress, quint16
   port,
                           bool useTls, const QString &clientId,
                           const QString &username, const QString
    &password,
                           const QString &topic);
    Q_INVOKABLE bool stop();
    Q_INVOKABLE void setParserThreadCount(int count);
    Q_INVOKABLE void setDebugMode(bool enabled);

private:
    QThread m_receiverThread;
    MqttReceiverWorker *m_receiverWorker;
    QThreadPool m_parserPool;
    QList<MqttParserWorker *> m_parsers;
    int m_nextParserIndex;  // Round-robin distribution

    // Atomic data storage
    std::atomic<float> m_speed;
    std::atomic<int> m_rpm;
    // ... additional atomic members
};
```

2.1.3 UdpClient

Location: `src/Controllers/udp/`

Purpose: UDP datagram receiver for low-latency real-time telemetry.

Threading Model

- **UdpReceiverWorker**: Listens on specified UDP port using `QUdpSocket`
- **UdpParserWorker**: Pool-based parsing with load distribution

Key Features

- Non-blocking I/O with `QUdpSocket`
- Round-robin parser assignment via `m_nextParserIndex`
- Debug mode for troubleshooting network issues
- Datagram processing/dropped counters for performance monitoring

Start Method

```
Q_INVOKABLE bool start(quint16 port);
// Binds to the specified UDP port and begins receiving datagrams
```

2.1.4 SerialManager

Location: `src/Controllers/serial/`

Purpose: Serial port communication handler for wired telemetry connections.

Threading Model

- **SerialReceiverWorker**: `QSerialPort` handler in dedicated thread
- **SerialParserWorker**: Concurrent parsing pool

Configuration Parameters

- **Port Name**: Serial port identifier (e.g., `/dev/ttyUSB0`, `COM3`)
- **Baud Rate**: Communication speed (e.g., 115200, 921600)

Start Method

```
Q_INVOKABLE bool start(const QString &portName, qint32 baudRate);
// Opens the serial port with specified configuration
```

2.1.5 CanDecoder

Location: `src/Controllers/can/`

Purpose: CAN bus message decoding utility.

Functionality

- Parses raw CAN frames into structured telemetry values
- Handles different CAN message IDs and data formats
- Provides decoded values to the communication clients

2.1.6 AsyncLogger

Location: `src/Controllers/logging/`

Purpose: Thread-safe CSV logging for telemetry data recording.

Design Pattern

Singleton Pattern - accessed via `AsyncLogger::instance()`

Threading Model

- Dedicated `LoggerWorker` thread with message queue
- Non-blocking logging via signal/slot with `Qt::QueuedConnection`

Log Entry Types

```
struct LogEntry {
    enum Type { IMU, SUSPENSION, TEMPERATURE };
    Type type;
    qint64 timestamp;
    QString data;
};
```

Logging Methods

```
void logIMU(int16_t ang_x, int16_t ang_y, int16_t ang_z);
void logSuspension(uint16_t sus_1, uint16_t sus_2,
                   uint16_t sus_3, uint16_t sus_4);
void logTemperature(int16_t temp_fl, int16_t temp_fr,
                    int16_t temp_rl, int16_t temp_rr);
```

2.2 UI Module

The UI module (`src/UI/`) contains all QML components for the user interface. Components are organized into logical subdirectories based on their function.

2.2.1 InformationPage (Main Dashboard)

Location: `src/UI/InformationPage/`

This directory contains the primary dashboard components that display real-time telemetry data.

Component	**Purpose**	**Key Features**
`Information.qml`	Main dashboard container	Responsive scaling via `scaleFactor`, GG diagram with path tracking, layout management
`Speedometer.qml`	Vehicle speed gauge	Canvas-based rendering, animated needle, configurable max speed
`RpmMeter.qml`	Engine RPM gauge	Canvas-based arc gauge, redline indicator
`GpsPlotter.qml`	GPS map display	Qt Location integration, real-time marker tracking, zoom controls
`SteeringWheel.qml`	Steering angle display	Rotating image based on encoder angle

Component	Purpose	Key Features
WheelSpeed.qml	Individual wheel speeds	FL/FR/BL/BR display with position labels
AcceleratorPedal.qml	Throttle position	Progress bar visualization with percentage
BrakePadel.qml	Brake position	Progress bar visualization with percentage
BatteryLevelIndicator.qml	Battery SOC	Canvas bar chart with level indicator
TemperatureIndicator.qml	Temperature display	Thermometer icon with numeric value
EulerGauges.qml	IMU orientation	Roll/Pitch/Yaw gauge displays
EulerVisual.qml	3D orientation visual	IMU data 3D visualization

Responsive Scaling System

The dashboard implements a responsive scaling system using a `scaleFactor` property:

```
Rectangle {
    id: root

    // Dynamic scale factor based on design size (1400x780)
    property real scaleFactor: Math.min(width / 1400, height /
    780)

    // All dimensions scaled proportionally
    radius: 40 * scaleFactor
    border.width: Math.max(3, 5 * scaleFactor)

    // Child components receive scaleFactor
    Speedometer {
        scaleFactor: root.scaleFactor
        // ...
    }
}
```

GG Diagram Implementation

The G-force diagram tracks lateral and longitudinal acceleration:

```
property real maxLateralG: 3.5
property real maxLongitudinalG: 2.0
property real xDiagram: (communicationManager.lateralG /
    maxLateralG)
                        * (ggImage.width / 2 - 20 * scaleFactor)
property real yDiagram: (communicationManager.longitudinalG /
    maxLongitudinalG)
```

```
                        * (ggImage.height / 2 - 20 * scaleFactor)

// Path tracking with Timer
Timer {
    id: pathTracker
    interval: 100
    running: bottomRect.hasReceivedData  // Only when data
   received
    repeat: true
    // Track point positions over time
}
```

2.2.2 WelcomePage (Session Setup)

Location: `src/UI/WelcomePage/`

Component	Purpose
`WelcomeScreen.qml`	Connection configuration interface with protocol selection (UDP/Serial/MQTT), port/broker settings, session naming
`WaitingScreen.qml`	Loading indicator displayed during connection establishment
`MyButton.qml`	Reusable styled button component with hover effects

2.2.3 StatusBar

Location: `src/UI/StatusBar/`

Component	Purpose
`StatusBar.qml`	Top status bar displaying current time, session name, and connection port/protocol information

StatusBar Implementation

```
Rectangle {
    id: root
    property string nameofsession: ""
    property string nameOfport: ""
    property real scaleFactor: 1.0

    width: parent.width - 24 * scaleFactor
    height: Math.max(25, 35 * scaleFactor)

    Text {
        id: timeText
        text: Qt.formatDateTime(new Date(), "hh:mm A")
        font.pixelSize: Math.max(16, 25 * scaleFactor)
```

```
    }

    Timer {
        interval: 1000
        running: true
        repeat: true
        onTriggered: timeText.text = Qt.formatDateTime(new Date()
   , "hh:mm A")
    }
}
```

Chapter 3

Software Best Practices

This chapter highlights the software engineering best practices implemented throughout the Telemetry Dashboard codebase.

3.1 Separation of Concerns

The project maintains a clean separation between business logic and presentation:

- **Controllers** (`src/Controllers/`): Handle all data acquisition, parsing, and business logic
- **UI** (`src/UI/`): Focus purely on data visualization and user interaction
- **No Business Logic in QML**: QML components only bind to properties and respond to changes

3.2 Facade Pattern

The `CommunicationManager` implements the Facade design pattern:

- Provides a simplified, unified interface to the QML layer
- Hides the complexity of multiple communication protocols
- Allows protocol switching without UI changes
- Centralizes error handling and state management

```
class CommunicationManager : public QObject
{
private:
    UdpClient *m_udpClient;
    SerialManager *m_serialManager;
    MqttClient *m_mqttClient;

    enum class SourceType { None, Serial, Udp, Mqtt };
    SourceType m_currentSource;

    // Single unified interface exposed to QML
    Q_PROPERTY(float speed READ speed NOTIFY speedChanged)
    // ... 14 more properties
};
```

3.3 Worker Thread Pattern

Each communication protocol implements a consistent Receiver + Parser worker separation:

1. **Receiver Worker**: Dedicated thread for I/O operations (socket/serial listening)
2. **Parser Workers**: Thread pool for CPU-intensive parsing operations
3. **Round-Robin Distribution**: Even load distribution across parser threads

This pattern ensures:

- Non-blocking I/O operations
- Parallel data parsing
- Responsive UI even under high data rates

3.4 Thread Safety

Multiple mechanisms ensure thread-safe operation:

3.4.1 Atomic Variables

```
// All shared data uses std::atomic<T>
std::atomic<float> m_speed;
std::atomic<int> m_rpm;
std::atomic<double> m_encoderAngle;

// Atomic getters
float speed() const { return m_speed.load(); }
```

3.4.2 Thread Pool

```
QThreadPool m_parserPool;
QList<ParserWorker *> m_parsers;
int m_nextParserIndex;

// Round-robin task distribution
void handleDataReceived(const QByteArray &data) {
    m_parsers[m_nextParserIndex]->parseData(data);
    m_nextParserIndex = (m_nextParserIndex + 1) % m_parsers.size
    ();
}
```

3.4.3 Signal/Slot with Queued Connections

```
// Cross-thread communication via Qt's queued connections
connect(m_receiverWorker, &ReceiverWorker::dataReceived,
        this, &Client::handleDataReceived,
```

```
        Qt::QueuedConnection);
```

3.5 Reactive Data Binding

Qt's Q_PROPERTY system enables automatic UI updates:

```
// C++ Controller
Q_PROPERTY(float speed READ speed NOTIFY speedChanged)

void setSpeed(float newSpeed) {
    if (m_speed != newSpeed) {
        m_speed = newSpeed;
        emit speedChanged(newSpeed);  // Automatic UI update
    }
}
```

```
// QML UI - automatically updates when signal emitted
Speedometer {
    speed: communicationManager.speed
}
```

3.6 Singleton Pattern

The `AsyncLogger` uses the Singleton pattern for centralized logging:

```
class AsyncLogger : public QObject
{
public:
    static AsyncLogger &instance();

    void initialize(const QString &logDirectory = ".");
    void shutdown();
    void logIMU(int16_t ang_x, int16_t ang_y, int16_t ang_z);

private:
    explicit AsyncLogger(QObject *parent = nullptr);
    ~AsyncLogger();

    // Delete copy constructor and assignment
    AsyncLogger(const AsyncLogger &) = delete;
    AsyncLogger &operator=(const AsyncLogger &) = delete;
};

// Usage
AsyncLogger::instance().logIMU(x, y, z);
```

3.7 Responsive Design

The UI implements a proportional scaling system:

```
// Base design size: 1400x780
property real scaleFactor: Math.min(width / 1400, height / 780)

// All dimensions scaled with minimum values for legibility
width: Math.max(500, 675 * scaleFactor)
font.pixelSize: Math.max(16, 25 * scaleFactor)
radius: Math.max(12, 20 * scaleFactor)
```

3.8 Modular Architecture

Each protocol module is self-contained with consistent structure:

```
protocol/
|-- include/
|   |-- protocolclient.h
|   |-- protocolparserworker.h
|   +-- protocolreceiverworker.h
+-- src/
    |-- protocolclient.cpp
    |-- protocolparserworker.cpp
    +-- protocolreceiverworker.cpp
```

Benefits:

- Easy to add new protocols
- Clear dependency boundaries
- Independent testing possible
- Consistent API across protocols

3.9 Error Handling

Consistent error handling pattern across all components:

```
signals:
    void errorOccurred(const QString &error);

// Error propagation from workers to main client
void handleWorkerError(const QString &error) {
    qWarning() << "Error:" << error;
    emit errorOccurred(error);
}
```