# Technical Documentation for the GPS Receiver Application

Summary

[Technical Documentation for the GPS Receiver Application 1](#_Toc192680277)

[1. Overview 1](#_Toc192680278)

[2. System Architecture 1](#_Toc192680279)

[**Main Components** 1](#_Toc192680280)

[3. UML Class Diagrams 3](#_Toc192680281)

[4. UML Sequence Diagram for FIFO WRITE Mode 4](#_Toc192680282)

[5. Component Diagram 4](#_Toc192680283)

[6. General Sequence Diagram. 5](#_Toc192680284)

[6. Final Considerations 6](#_Toc192680285)

## 1. Overview

**Purpose:**  
This application is designed to receive GPS data in NMEA format via a serial port, parse it, and display the information in a graphical user interface (GUI). The system supports multiple operating modes:

* **Standalone Mode:**   
  The GPSReader reads raw NMEA sentences from the serial port. A ParserWorker (running in a separate thread) processes these sentences and emits signals with parsed data, which are used to update the GUI.
* **Plugin Mode – FIFO WRITE:**   
  The application writes the processed GPS data into a named pipe (FIFO). In this mode, the ParserWorker emits a signal with new coordinates (along with a message type), which is sent both to the GUI and to the FIFOWriter that writes to the FIFO. (A persistent reader is needed on the FIFO side; in testing, a dummy reader may be used as a workaround.)
* **Plugin Mode – FIFO READ:**   
  In this mode, a FIFODataSource continuously reads from the FIFO and updates the GUI with the data it obtains.
* **Plugin Mode – SOCKET:**   
  (Stub) Data is received via a socket connection.

**Main Functional Requirements:**

* Receive and parse NMEA sentences from a serial port.
* Display in the GUI:
  + Raw data log (the NMEA sentences as received)
  + Processed data (e.g., coordinates and message type)
  + System status messages (e.g., connection status, errors)
* Support different operational modes via configuration settings.
* Use multi-threading to run the parsing in the background.
* Handle FIFO operations robustly (both writing and reading) while managing errors like EPIPE or ENXIO.
* Provide a modular structure so that the system can be easily extended (for example, to add new modes).

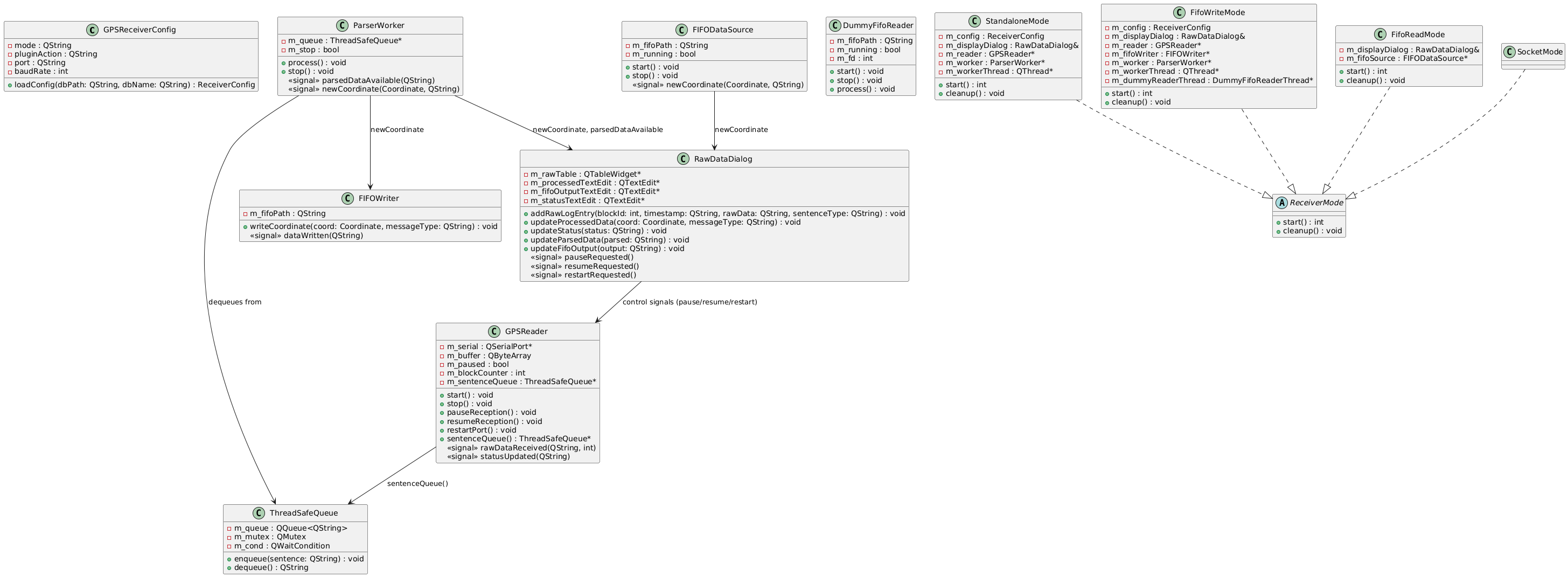
**2. System Architecture**

The system is designed in a modular way and follows a Producer–Consumer pattern:

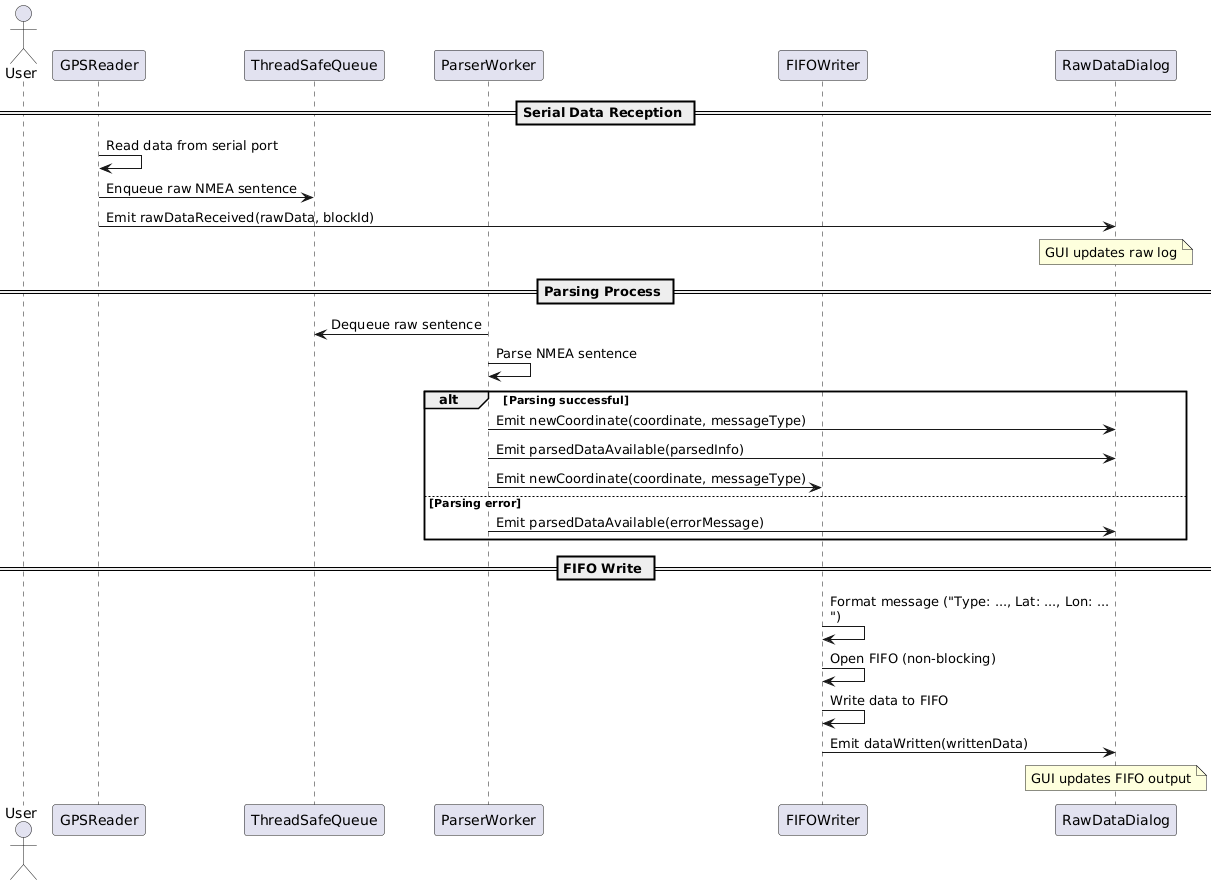
**Main Components**

* **GPSReceiverConfig:**
  + Loads configuration settings (mode, pluginAction, serial port, baud rate) from a SQLite database.
* **GPSReader:**
  + Manages the serial port connection.
  + Reads raw NMEA sentences and emits signals (e.g., rawDataReceived and statusUpdated).
  + Enqueues the raw sentences into a thread-safe queue for further processing.
* **ThreadSafeQueue:**
  + A thread-protected queue (using QMutex and QWaitCondition) that holds incoming sentences, allowing safe communication between the GPSReader thread and the ParserWorker thread.
* **ParserWorker:**
  + Runs in a separate QThread.
  + Dequeues raw sentences from the ThreadSafeQueue and parses them.
  + Emits signals:
    - parsedDataAvailable to update logs or status.
    - newCoordinate(Coordinate, messageType) when a valid coordinate is parsed.
* **FIFOWriter:**
  + Receives coordinate data (and message type) from the ParserWorker.
  + Opens the FIFO in non-blocking mode (using O\_NONBLOCK) and writes a formatted string (e.g., "Type: GPRMC, Lat: 38.084088, Lon: 15.635065\n").
  + Emits a dataWritten signal to inform the GUI about what was written.
  + Uses error-handling to log issues if no reader is present.
* **FIFODataSource:**
  + Used in FIFO READ mode.
  + Continuously reads from the FIFO and processes the string to extract data.
  + Emits signals (e.g., newCoordinate) that update the GUI.
* **DummyFifoReader (Optional):**
  + A workaround for testing in FIFO WRITE mode.
  + Runs in its own thread to keep the read-end of the FIFO open, so that the writer does not fail when no “real” reader is attached.
  + Must handle EOF and errors by closing and reopening the FIFO as needed.
* **RawDataDialog (GUI):**
  + Displays the raw NMEA log, the processed (parsed) data (e.g., coordinates with message type), and system status messages.
  + Provides control signals (e.g., pause, resume, restart) to control the GPSReader.
* **ReceiverMode Interface & Implementations:**
  + An abstract interface (ReceiverMode) defines methods such as start() and cleanup().
  + Concrete classes (e.g., StandaloneMode, FifoWriteMode, FifoReadMode, SocketMode) encapsulate the setup, execution, and cleanup logic for each operating mode.
* **ReceiverModeFactory:**
  + A factory that creates an instance of the appropriate ReceiverMode based on the configuration and command-line parameters.

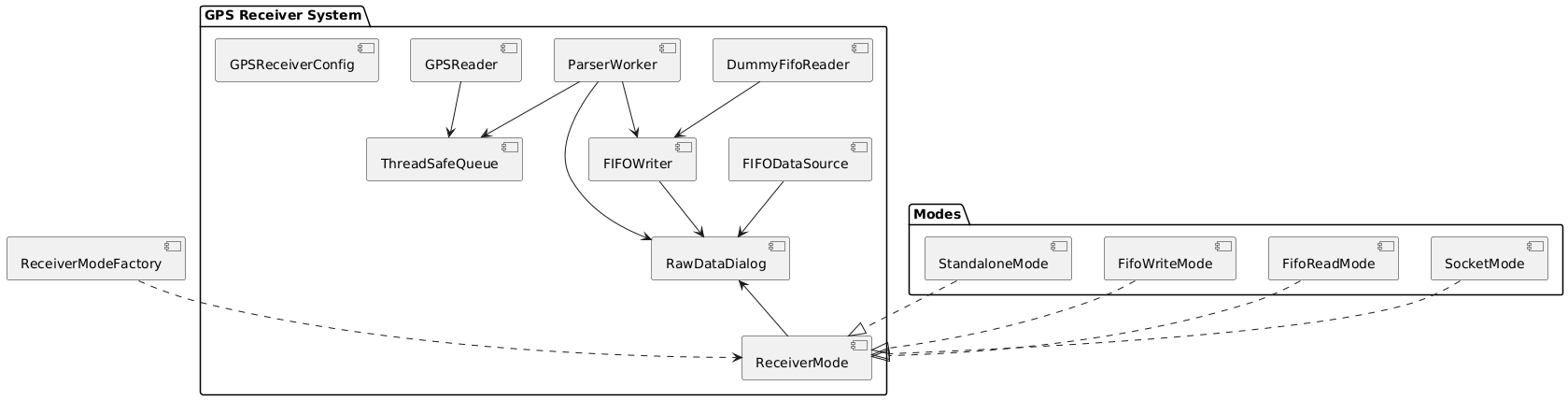
## 3. UML Class Diagrams



## 4. UML Sequence Diagram for FIFO WRITE Mode



## 5. Component Diagram



## 6. General Sequence Diagram.

In the next page there is a comprehensive PlantUML sequence diagram that covers the full flow of the system (using FIFO WRITE mode as an example). This diagram shows how raw NMEA data is read from the serial port, enqueued, parsed, and then delivered both to the GUI and (in FIFO WRITE mode) written into a FIFO. In production, the FIFO’s read-end should be kept open by a dedicated reader (or another instance in FIFO READ mode);

|  |  |
| --- | --- |
| PlantUML diagram | Explanation  **Serial Data Reception:**  The **GPSReader** starts and opens the serial port.  It reads raw NMEA sentences from the serial port and enqueues them into a **ThreadSafeQueue**.  It also emits a signal (rawDataReceived) to update the GUI’s raw data log (RawDataDialog).  **Parsing:**  The **ParserWorker**, running in a separate thread, dequeues a raw sentence from the queue and parses it.  On successful parsing, it emits signals: one to update the GUI with parsed data and another (newCoordinate) carrying a Coordinate structure and a message type.  On parsing error, it emits an error message to the GUI.  **FIFO WRITE (Plugin Mode):**  In FIFO WRITE mode, the **ParserWorker**’s newCoordinate signal is also connected to the **FIFOWriter**.  The FIFOWriter formats the coordinate and message type into a string and opens the FIFO in non-blocking mode.  If the FIFO is successfully opened and written to, it emits a dataWritten signal so that the GUI can update accordingly.  Optionally, if a DummyFifoReader is active (used to keep the FIFO’s read end open during testing), it reads and discards the data to prevent write errors.  **Cycle Repeats:**  The entire process repeats as new raw data arrives from the serial port. |

## 6. Final Considerations

* **Modularity:**  
  The use of an abstract ReceiverMode interface and dedicated classes (StandaloneMode, FifoWriteMode, FifoReadMode, SocketMode) isolates the code for each mode. A factory (ReceiverModeFactory) selects and creates the correct mode at runtime based on configuration and command-line options.
* **Extensibility:**  
  This design allows you to add new modes or modify existing ones with minimal changes to the overall system. For example, a new communication protocol can be added by implementing a new ReceiverMode.
* **Robustness:**  
  By using a thread-safe queue and running the ParserWorker in its own thread, the system avoids blocking the main thread. Proper error handling and cleanup help prevent crashes and resource leaks.
* **Production vs. Testing:**   
  While a dummy reader can be useful for testing FIFO WRITE mode, in production it is preferable to run a dedicated FIFO READ process (or mode) that continuously reads from the FIFO.

**7. UML Diagram Legend**

* **Actor:**  
  Represented by a stick figure icon. It denotes an external entity (such as a user or external system) interacting with the system.
* **Participant / Object:**  
  Represented by a rectangle with the name of the class or component inside. In sequence diagrams, these are the lifelines (vertical dashed lines) showing the object's existence over time.
* **Message Arrow:**
  + **Solid Arrow (→):** Represents a synchronous call where the sender waits for the receiver to finish processing.
  + **Open Arrow (-->):** Represents an asynchronous message or signal that does not block the sender.
* **Activation Bar:**  
  A thin rectangle drawn on a lifeline, indicating the period an object is performing an action.
* **Alt / Else Blocks (alt):**  
  Used to show alternative flows in a sequence diagram. Each branch is labeled with a condition.
* **Note:**  
  A rectangular or cloud-like shape attached with a dashed line to a diagram element. It contains additional comments or explanations.
* **Stereotypes (e.g., <<signal>>):**  
  Text within guillemets (<< >>) that provide extra semantic meaning (for example, indicating that a method is a signal).
* **Return Messages (dashed arrow):**  
  Represent the return of control or data from a called operation. Typically, they are drawn with a dashed arrow and may contain a return value.