Software Requirements Specification (SRS)

# Project Title: Multi-Source Live Aviation Monitoring and Display System

## Platform:

Linux (C Language with IPC and Multithreading mechanisms)

# 1. Introduction

## 1.1 Purpose

This SRS document defines the requirements for a real-time aviation monitoring system that acquires, processes, synchronizes, and displays live video and sensor data from multiple aircraft, drones, and radar sources. The system ensures minimal latency and accurate visualization for air traffic control and onboard monitoring.

## 1.2 Scope

The system captures live video and sensor data (altitude, speed, GPS) from multiple sources, processes them in parallel, performs object detection and tracking, encodes and streams the data, and displays it on a real-time dashboard. It uses Linux IPC mechanisms (Shared Memory, Semaphores, Message Queues, Sockets) and multithreading for concurrency and synchronization.

## 1.3 Definitions, Acronyms, and Abbreviations

- IPC: Inter-Process Communication  
- RTS: Real-Time Streaming  
- UI: User Interface  
- GPS: Global Positioning System  
- TCP: Transmission Control Protocol  
- POSIX: Portable Operating System Interface

# 2. Functional Requirements

|  |  |  |
| --- | --- | --- |
| ID | Function | Description |
| FR1 | Multi-Source Data Acquisition | Acquire live video and sensor data from aircraft, drones, and radar using multithreading. |
| FR2 | Parallel Data Processing | Process video (e.g., stabilization, overlays) and sensor data (e.g., altitude, GPS) in parallel. |
| FR3 | Object Detection & Tracking | Detect and track moving objects across multiple video feeds. |
| FR4 | Data Encoding & Communication | Encode and compress video/sensor data; transmit via sockets and message queues. |
| FR5 | Real-Time Display & Dashboard | Display synchronized video and sensor data on an interactive UI dashboard. |
| FR6 | Fault Detection & Recovery | Detect failures in modules and recover using signal handling and watchdog threads. |

# 3. Non-Functional Requirements

- Concurrency: Support multithreaded and multi-process architecture for real-time performance.  
- Reliability: Handle module failures gracefully using signal handling and watchdogs.  
- Efficiency: Optimize CPU and memory usage for real-time responsiveness.  
- Scalability: Extendable to additional aircraft or sensor types.  
- Synchronization: Maintain temporal alignment across video and sensor streams.

# 4. Software and Hardware Requirements

## 4.1 Software Requirements

- OS: Ubuntu Linux  
- Compiler: GCC  
- Tools: ipcs, ipcrm, shmget, semget, msgget, socket, pthread  
- Language: C (with POSIX threads and IPC)

## 4.2 Hardware Requirements

- Simulated aircraft video and sensor data (text-based or emulated)  
- No physical aircraft or cameras required for simulation

# 5. System Overview (Process-Based)

Process Flow:  
1. Data Acquisition Threads: Capture video and sensor data concurrently.  
2. Shared Memory Manager: Stores sensor data for access by multiple processes.  
3. Video Processor: Stabilizes and enhances video frames.  
4. Sensor Overlay Module: Adds altitude, speed, and GPS overlays to video.  
5. Object Tracker: Detects and tracks moving objects across feeds.  
6. Encoder: Compresses video and sensor data for transmission.  
7. Streaming Client: Sends encoded data to the display server.  
8. UI Dashboard (TCP Server): Displays synchronized video and sensor overlays.  
9. Watchdog & Signal Handler: Monitors system health and handles failures.

# 6. Constraints

- Each module must release IPC resources upon exit using cleanup.c.  
- Manual launching of modules in separate terminals is required.  
- Simulated data is used for development and testing.

# 7. Appendices

## A. Assumptions

- Users will run each module manually in separate terminals.  
- Simulated video and sensor data are acceptable for testing.

## B. Glossary

- Frame: A single image or data block from a video stream.  
- Client: The UI dashboard receiving and displaying processed data.  
- Sensor Data: Includes altitude, speed, and GPS coordinates.  
- Watchdog: A monitoring thread that detects and recovers from failures.

24-10-2025:

**Video Frame and Sensor Data Acquisition :**

In this phase, we implemented and refined two critical components of the aviation monitoring system — the Video Frame Acquisition Thread and the Sensor Data Thread.

Together, they handle the synchronized extraction of visual data (video frames) and simulated flight sensor data (altitude, speed, and GPS coordinates), ensuring that both streams are available for downstream processing and UDP transmission.

**Video Acquisition**

* The video acquisition module is responsible for extracting and transmitting video frames from the input feed at a fixed frame rate.
* The video is read using OpenCV’s VideoCapture and processed frame-by-frame.
* The extraction rate is controlled to maintain a consistent 15 FPS by calculating a frame skip interval based on the source video’s FPS.
* Each frame is resized to 320×240 pixels for efficient UDP transfer and then flipped vertically to correct orientation.
* Frames are saved in both PPM (for transmission) and JPG (for local viewing) formats inside the resources/frames/ directory.
* During transmission, each frame is packed into a UDP packet containing the image data and its corresponding sensor readings.
* The system ensures safe access to shared data using mutex locks (frame\_mutex and sensor\_mutex), preventing conflicts between threads.
* Frame transmission timing is managed using usleep() to maintain the 8 FPS output rate.

**Sensor Data Acquisition**

* The sensor data acquisition module generates and updates flight telemetry data synchronized with each video frame.
* Sensor values such as altitude, speed, and GPS coordinates are initialized for every frame at system startup.
* Altitude and speed gradually increase with each frame, while latitude and longitude are slightly adjusted to simulate flight movement.
* The sensor\_data\_thread continuously monitors the current frame index and updates the corresponding current sensor data in shared memory.
* Thread synchronization is achieved through mutex locking, ensuring consistent data between video and sensor threads.
* The update loop runs every 100 ms, providing near real-time tracking of telemetry data alongside the video feed.

**Shared memory Communication :**

The shared memory module establishes a centralized communication space for all system threads - including video acquisition, sensor updates, and the processing pipeline.

This ensures efficient, real-time data sharing without file I/O or network overhead.

Initialization :

The shared memory region is created using POSIX shared memory (shm\_open, mmap).

Synchronization primitives are initialized:

4 mutexes: for sensor, frame, detection, and UI synchronization

1 barrier: to coordinate processing stages

2 semaphores: for signaling frame readiness and processing completion

1 condition variable: to notify the UI thread of updates

Once initialized, every thread (video, sensor, processing, etc.) accesses this shared space concurrently — safely managed via mutex locks.

**Writing to Shared Memory:**

The video acquisition thread writes:

current\_frame index

frame\_ready\_for\_processing = true after saving a new frame

The sensor thread writes:

Real-time telemetry data (altitude, speed, latitude, longitude) into frame\_sensors[i]

Updates current\_sensor for the current frame

**Synchronization:**

Each write operation is wrapped between:

pthread\_mutex\_lock(&shm->frame\_mutex);

... // update frame or sensor data

pthread\_mutex\_unlock(&shm->frame\_mutex);

ensuring that no two threads modify the same memory region simultaneously.

**Processing Video and Sensor Data :**

The processing pipeline simulates a modular, multi-stage data processing workflow operating on frames and sensor data stored in shared memory.

Structure

The pipeline consists of three sequential stages, each running in its own thread:

* Stage 1: Initial pre-processing placeholder
* Stage 2: Intermediate computation placeholder
* Stage 3: Final stage that marks processing completion

All stages run concurrently and remain synchronized through the processing barrier and shared flags.

Operation Flow

Frame Ready:

Once a frame is captured and marked as frame\_ready\_for\_processing, the processing stages become active.

Processing:

Each stage thread operates independently (currently silent placeholders). In a full system, this could include:

* Frame enhancement or stabilization
* Object or obstacle detection
* Data fusion with sensor readings

Completion Signal:

The third stage sets:

pthread\_mutex\_lock(&shm->frame\_mutex);

shm->processing\_complete = true;

pthread\_mutex\_unlock(&shm->frame\_mutex);

indicating that the frame’s processing cycle is done and results are ready for the next module (e.g., detection or UI).

Synchronization:

Threads can use sem\_post() and sem\_wait() to coordinate the processing of each new frame.