Aeroplane Management System — Project Proposal

Project Title

Aeroplane Management System (AMS) — Implementation in C

Abstract

This project proposes the design and implementation of an Aeroplane Management System (AMS) written in the C programming language. The AMS is a console-based application to manage aircraft records, flight schedules, crew assignments, maintenance logs, reservations, and seat booking. The system will focus on robust data structures, file-based persistent storage, modular code, and clear command-line interaction suitable for educational use and as a foundation for further expansion.

Problem Statement

Airline and airport operations require reliable methods to store and retrieve aircraft information, manage flight schedules, track maintenance, assign crew, and handle passenger bookings. Small-scale systems or academic projects often lack an integrated program implemented in low-level languages that demonstrates data structures, file I/O, and system design principles. This project fills that gap by producing a working C-language system that is modular, documented, and extendable.

Objectives

- Implement core modules for aircraft, flights, crew, maintenance, reservations, and seat booking.
- Use C data structures (structs, arrays, linked lists, file records) to model entities.
- Provide persistent storage using binary and/or text files with safe read/write routines.
- Implement user-friendly command-line menus with input validation and error handling.
- Demonstrate sorting, searching, and simple report-generation algorithms.
- Build a modular codebase with clear headers and implementation files suitable for a university project.

Scope

Included features: - Aircraft management: add, update, delete, list aircraft records (tail number, model, capacity, status). - Flight schedule management: create, edit, cancel, list flights (flight number, origin, destination, departure/arrival time, aircraft assigned). - Crew management: store crew members, assign to flights, view crew rosters. - Maintenance logs: record inspections, maintenance events, next inspection dates; mark aircraft as grounded for maintenance. - Reservation & Booking: passenger reservation records for a flight (name, seat, class, status). Functions for booking, checking seat availability, counting booked vs. remaining seats, and calculating fares for Business and Economy class. - Reporting: generate lists and basic summaries (e.g., flights per day, grounded aircraft, available seats).

Out-of-scope / future work: - Networked multi-user access or database server back-end. - Full reservation/ payment gateway. - Advanced scheduling optimization.

Target Users

- Students learning systems programming in C.
- Instructors needing a demonstrable project covering file I/O and data structures.
- Hobbyists building a simple airline ops simulator.

Functional Requirements

1. **Aircraft Module**
2. `AddAircraft()` — Create new aircraft record.
3. `EditAircraft()` — Update details.
4. `DeleteAircraft()` — Remove aircraft (safe delete or mark inactive).
5. `ListAircraft()` — Display all records with filters (status, model).
6. **Flight Module**
7. `CreateFlight()` — Add a new flight entry.
8. `AssignAircraftToFlight()` — Link aircraft to flights (check availability and maintenance status).
9. `EditFlight()` / `CancelFlight()`.
10. `SearchFlights()` — By date, origin, destination.
11. **Crew Module**
12. `AddCrewMember()`, `AssignCrewToFlight()`, `ListCrew()`.
13. **Maintenance Module**
14. `LogMaintenance()` — Record maintenance events and set aircraft status.
15. `DueInspections()` — List upcoming inspections.
16. **Reservation & Booking Module**
17. `AddReservation()` — Add passenger record for a flight.
18. `CancelReservation()`.
19. `ListReservations()` for a flight.
20. `BookSeat()` — Assign seat (Business or Economy) if available.
21. `CheckSeatAvailability()` — Show how many seats are booked vs. left.
22. `CalculateFare()` — Display fare for Business vs. Economy class.
23. `GetPassengerSeatInfo()` — Retrieve seat details for a passenger.

24. **Persistence**

25. Read/write each module's data to dedicated files (e.g., `aircraft.dat`, `flights.dat`, `crew.dat`, `maintenance.dat`, `reservations.dat`).

26. Implement safe file update (temp file swap) and consistent record formats.

27. **CLI**

28. Text-based main menu and submenus.
29. Confirmation prompts for destructive actions.

Non-functional Requirements

- Code written in ANSI C (C99 standard recommended).
- Modular organization: multiple `.c` and `.h` files, with a `Makefile` for build.
- Clear error handling and input validation.
- Reasonable performance for datasets of up to several thousand records.
- Portable to Linux / Windows (via MinGW) compilers.

High-Level Design

Data Structures

- `struct Aircraft { char tail[10]; char model[32]; int capacity; char status[16]; time_t last_maintenance; }`
- `struct Flight { char flight_no[8]; char origin[32]; char dest[32]; struct tm dep_time; struct tm arr_time; char aircraft_tail[10]; char status[16]; }`
- `struct CrewMember { int id; char name[64]; char role[16]; }`
- `struct Maintenance { int id; char tail[10]; time_t date; char description[128]; }`
- `struct Reservation { int id; char flight_no[8]; char passenger_name[64]; int seat_no; char seat_class[16]; char status[16]; float fare; }`

Storage: use binary files for compactness; optionally provide a text export/import utility.

Modules / Files

- `main.c` — application entry, main menu loop.
- `aircraft.c` / `aircraft.h` — aircraft CRUD and file operations.
- `flight.c` / `flight.h` — flights and scheduling operations.
- `crew.c` / `crew.h` — crew management.
- `maintenance.c` / `maintenance.h` — maintenance logging.
- `reservation.c` / `reservation.h` — reservations and seat booking.
- `utils.c` / `utils.h` — common helpers: input helpers, date/time parsing, file helpers.
- `io.c` / `io.h` — generic file read/write wrappers if desired.
- `Makefile` — build rules.

Algorithms

- Linear search for small datasets; implement optional indexed search (e.g., in-memory hash or sorted array with binary search) for flights by flight number.
- Sorting functions for listing operations (by date, flight number) using `qsort()`.
- Seat allocation algorithm to ensure no overbooking.

User Interface

- Console-driven menus with numbered options and keyboard input.
- Clear formatting of tables using fixed-width columns.
- Seat map view per flight (optional extension).
- Date/time input with validation helper (e.g., `DD-MM-YYYY HH:MM`).

Testing Plan

- Unit tests for core utilities (parsing, file helpers) — manual test programs.
- Manual test plan for each module: create, read, update, delete flows; persistence across runs.
- Edge cases: duplicate keys (tail number / flight number), invalid dates, seat overbooking prevention, file I/O error handling.

Project Timeline (12 weeks — example)

- Week 1: Requirements finalization, environment setup, skeleton project.
- Week 2–3: Implement Aircraft module + file persistence.
- Week 4–5: Implement Flight module and basic scheduling.
- Week 6: Crew and Maintenance modules.
- Week 7–8: Reservation and Seat Booking module.
- Week 9: CLI polishing, input validation.
- Week 10: Sorting/searching optimizations, reports.
- Week 11: Testing, bugfixing.
- Week 12: Final demonstration, submission packaging.

Resources and Tools

- C compiler (GCC / Clang / MinGW).
- Text editor / IDE (VS Code recommended).
- Version control (Git).
- Optional: Unit test framework for C (e.g., Check) for automated tests.

Deliverables

- Complete source code with modular `.c` and `.h` files.
- `Makefile` or build instructions.
- User manual (README) explaining usage and sample commands.
- Test cases and sample data files.
- Final report describing design decisions and future work.

Success Criteria

- All core modules implemented and demonstrably working.
- Persistent storage across program runs.
- Seat booking, availability check, and fare calculation working correctly.
- Clean and readable source code with comments.
- Basic test cases passed and demonstration of typical workflows.

Risks and Mitigation

- **File corruption risk** — mitigate using safe file write patterns (write to temp file then rename).
- **Complexity of date/time handling** — use `struct tm` and standardized parsing functions; limit accepted formats.
- **Seat booking conflicts** — implement strong validation before confirming seat.
- **Time constraints** — adopt iterative development and prioritize core features early.

Future Extensions

- Migrate storage to a lightweight database (SQLite).
- Add GUI (desktop) or web front-end.
- Add multi-user, networked access and role-based authentication.
- Implement advanced scheduling and conflict detection algorithms.
- Dynamic pricing for fares.

Prepared by: Syed Raahim Ali (CT-191) & Faizan Khan (CT-195)