Warsaw University of Technology

FACULTY OF
POWER AND AERONAUTICAL ENGINEERING

Institute of Aeronautics and Applied Mechanics

Bachelor’s diploma thesis

in the field of study Aerospace Engineering
and specialisation Automatics and Aviation Systems

Simulation of Flight Management System in X-Plane environment

Mateusz Brzozowski
student record book number 259 652

thesis supervisor
Dr Maciej Zasuwa

Warsaw, 2018

Abstract

Title: Simulation of Flight Management System in X-Plane environment

Keywords: FMS, Flight Management System, simulation, X-Plane

The aim of this engineering diploma thesis was to develop and describe a simulation of a portion of a Flight Management System functionality, more specifically its flight planning and lateral guidance functionality, in form of a dynamically linked library for X Plane 11 flight simulation software. In this document, the details on structure and implementation of the software are provided.

The thesis opens with establishment of goals and scope of the simulation. Following section consists of an introduction to the X Plane 11 environment, the description of the simulated system and the theoretical basis for the implementation. The theoretical basis consists of descriptions of mathematical formulae and algorithms used in the simulation as well as the basic structure of the resultant simulation. What follows is a detailed description of structure of the software including class relationships, the data flow within the software, as well as descriptions of individual functions and classes and methods involved in the implementation.

Finally, a description of UI generated by the code is given and the operation is described. The thesis ends with the conclusions on the results of existing development and its possible extension.

Streszczenie

Tytuł: Symulacja systemu FMS w środowisku X Plane

Słowa kluczowe: FMS, Flight Management System, symulacja, X Plane

Celem tej pracy inżynierskiej było stworzenie i opis symulacji części funkcji systemu FMS, w tym planowania lotu i nawigacji w poziomie, w formie biblioteki dynamicznej do X Plane 11. W tym dokumencie opisano szczegóły struktury i implementacji tego oprogramowania.

W początkowej części pracy ustalony jest cel i zakres symulacji. Kolejne sekcje składają się z wprowadzenia do środowiska X Plane 11, opisu symulowanego systemu i bazy teoretycznej do implementacji. Baza teoretyczna obejmuje opisy formuł matematycznych i algorytmów użytych w symulacji, jak również podstawową strukturę symulacji systemu. Następnie przedstawiony jest szczegółowy opis struktury oprogramowania, w tym powiązań klas, przepływu danych wewnątrz oprogramowania oraz opisy poszczególnych funkcji i klas i metod, na które składa się implementacja.

W końcowych sekcjach opisany jest wygenerowany interfejs użytkownika oraz korzystanie z niego. Praca kończy się wnioskami na temat stworzonego oprogramowania oraz jego potencjalnego dalszego rozwoju.

Contents

1. Introduction

1.1. Goal of the thesis

The purpose of this diploma project was to develop a general FMS simulation in the X-Plane 11 environment. It involved the design and programming of a dynamically linked library with the use of X-Plane SDK that simulated the basic informative, route management and lateral navigation functionality of a Flight Management Computer together with a simple UI imitating a Control Display Unit. While the interface is based on a general layout of the units installed in Boeing 737 series of aircraft, this plugin is designed as a universal tool that can used with any X-Plane based aircraft simulation, therefore most of its functionality was written from scratch to imitate capabilities of a real FMC in selected areas.

This thesis describes the design, functionality and typical usage scenarios of this software, illustrating them with the C++ implementation where applicable.

1.2. Examples of the simulated system

1.2.1. Boeing 737:

The primary basis for this project in terms of design philosophy, particularly the UI, is the GE Aviation model 2907C1 and earlier, featured in the Boeing 737 series aircraft. The system was developed from the existing Performance Data Computer System. In addition to the performance computation previously provided by PDCS, the FMC included navigation functionality, allowing it to fully remove the need for a third crew member in the cockpit. The full Flight Management System allowed for presence of 2 separate FMCs for redundancy, but only a single FMC was installed in most aircraft. Moreover, Flight Management Systems with Multifunctional CDUs allowed for direct access to subsystems of FMS other than the FMC, such as the ACARS, decentralising it [1].

The CDU interface uses a combination of text input and contextual selection to enter or retrieve data. The bottom-most line on the screen is always reserved for the scratchpad, where all text entered with the alphanumeric keyboard is held. The left and right edges of the screen are lined with Line Selection Keys, six for each side. They are used to input text from the scratchpad into an editable field, copy contents of a field to the scratchpad, activate functions or navigate to other pages, depending on context. The pages which include editable fields or selections are arranged into a 14-line pattern, with a title line, 6 main lines (aligned with LSKs) that the editable fields are placed in, 6 corresponding descriptor lines (typically presented with a smaller font and above the corresponding fields) and the scratchpad in the bottom line. On some pages the display might deviate from this pattern.

The navigation is based on context selections with the LSKs and the page shortcut keys on the top of the keyboard. The pages are arranged in logical manner based on the order they are used in during flight: the INIT-REF key brings up the current page in the FMS initialization procedure, selecting INDEX lists all the pages inaccessible through shortcut keys, allowing to navigate to them out of order. The bottom two LSKs typically navigate to the previous (left) and next (right) page in the flight preparation procedure and the pages that might be needed at any stage of flight (N1 limits, route, legs etc.) are accessed through dedicated keys at the top of the keyboard, with keys corresponding to pages that are typically used in certain order (like climb, cruise and descent) being arranged in the same order. If a page features multiple subpages, they are accessed with Next Page and Previous Page keys. Multifunctional CDUs also include a MENU key that lists other components of the FMS allowing access to their interfaces, which follow similar design principles.

Finally, there are two annunciation panels on the left and right edges of the keyboard, which display system errors, usually accompanied by a warning sound and an error message displayed in the scratchpad.

1.2.2. Airbus A320:

The Airbus A320 series the Flight Management System (or Flight Management and Guidance System as it is referred to in Airbus nomenclature) might differ from aircraft to aircraft, with systems from different manufacturers (Thales, Honeywell) available to the customers. The system is centralised, with the Flight Management and Guidance Computers at its core. Each aircraft has two FMGC units that typically work in dual mode (sharing workload) unless one of them malfunctions. The MCDU, talks directly to the FMGCs.

The Honeywell Pegasus MCDU mostly used in the A320 employs similar design features to the CDU in terms of user interface. The data and pages are grouped in a different manner and the order it is entered in during initialization is different. The information pages are grouped in Data index invoked with a dedicated DATA key, the initialization procedures are done with the INIT key. Instead of performance summary pages dedicated to stages of flight with their own page keys, the page relevant to the current stage of flight is invoked with the PERF key. The engine de-rate is integrated into other pages, there are dedicated pages to radio tuning and the general format of the is different with 3 columns instead of 2 and possibility of scrolling up and down within the same subpage in addition to switching subpages (with corresponding 4-way slew keys instead of next and previous page keys), which leads to more data being presented in a single page.

In addition to two annunciator panels, similar to the layout featured in 737, a third system status panel is present above the MCDU screen.

1.3. Existing simulation software

Software covering the scope of this project already exist for X Plane, albeit in a slightly different format. In addition to aircraft specific modelling, with detailed recreations of FMS being available in modules aimed at both amateur and professional market, there are some universal solutions which can be adapted to use with any aircraft modelled for the platform.

One such solution is X-FMC, however at this time the software is only compatible with X Plane 10. The X-FMC is plug-in based with its own, OpenGL UI graphics to render the CDU, custom navigation database independent from the internal database of X Plane and replicating most of the functionality of a modern FMC including full VNAV with engine control. The performance data needs to be supplied for the VNAV/autothrottle to function correctly, although it does not require a detailed performance database and approximates the vertical profile based on more general performance characteristics, such as rated climb rate in given altitude range. The UI layout is based on the 737NG FMS.

1.4. Requirements and scope of the development

Simulation of the FMC functionality in the base X-Plane 11 environment without the access to aircraft-specific data requires a combination of data extracted from simulator's database and educated assumptions to achieve expected functionality. The usage and visual layout are loosely based on the 737 series, particularly the NG, however much of the 737-specific functionality that would not be applicable to different aircraft had to be omitted. Additionally, the software does not simulate VNAV at this time, thus the functionality related to it is limited. The data entry is performed through a virtual keyboard. The interface has been prepared using exclusively the functionality provided by the X-Plane 11 SDK to reduce the reliance on external libraries.

To properly simulate the route management component of an FMC, the software had to support and manage a database of navigational aids, fixes and airways, as well as manage entry of flight plans into the system. The native navigation database of X-Plane 11 has been used as the source of information. This allows the software to work with X-Plane 11 without the need to acquire navigation data from another source.

The active navigational functionality simulated in this software is limited to lateral navigation at the time of writing this thesis. Possible further development includes vertical navigation and support for Coded Instrument Flight Procedures included in X-Plane 11 database.

2. Theory of operation of the software

In order to develop the simulation of a Flight Management System that would be generic in nature and usable with multiple different aircraft, a number of simplifications, limitations and assumptions have to be established. Firstly, since different aircraft have different navigation systems, autopilot systems, and require keeping track of different data as well as using different mathematical models for performance calculation, fully simulating functionality of an FMS requires its development in conjunction with a simulation of a full aircraft, with access to all the data regarding its flight model. Since it was not possible in this case and partially goes outside the scope of the project, the focus has been limited to following major elements of the FMS functionality:

- navigation
- flight planning
- lateral guidance

As well as the UI imitating a control display unit of an FMS

While the lateral guidance does require interfacing with aircraft's autopilot system, the extent of this interaction is much more limited than in case of vertical navigation. Additionally, the lack of access to the core modelling of the navigational systems limits the ability to accurately simulate the navigational aspect; however, the necessary data obtained by this element of FMC's operation is readily available from the simulation platform itself.

2.1. Architecture of the simulated system

To develop a functional simulation of a complex avionic system, information must be gathered on the system itself as well as the environment it is to be simulated in. While the object in this case is a generic flight management system, it still has to match the functionality of a real systems in certain aspects.

Figure 1 shows a schematic of a typical Flight Management System based on [2].

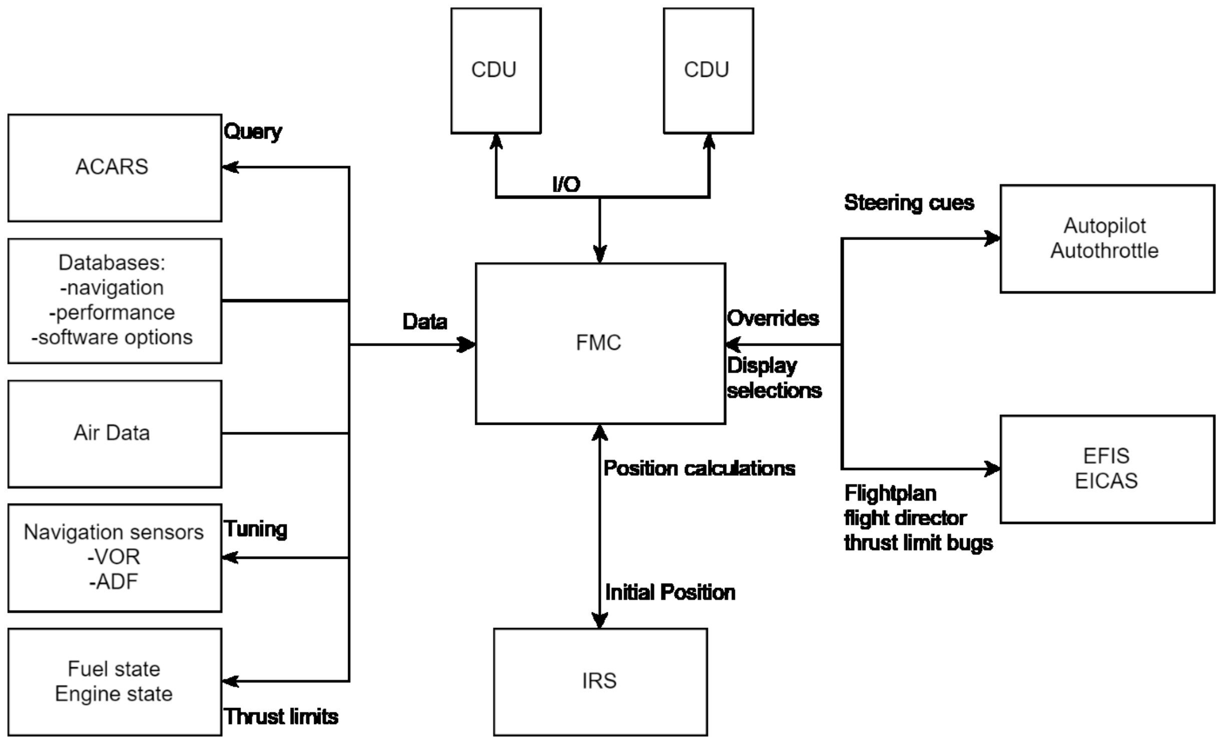


Figure 1: Typical Flight Management System interface diagram

A typical FMS consists of a single Flight Management Computer (multiple, synced units can be used instead, for redundancy) that is interacted with via Control Display Units (typically two, one for the pilot and one for the co-pilot).

The FMS takes data for operation from multiple sources:

- databases – in case of 737 [1] there are three databases: NDB – navigation database, MEDB – model/engine database, storing performance data and OP PROGRAM, storing software options
- sensors – a typical flight management system takes positional information from all available sources: VOR and ADF receivers, GPS/GNSS, INS etc. and runs them through a Kalman filter to obtain the most accurate positional information possible; additionally, the altitude, airspeed etc. sensors that are the source of data for PFD can be used.
- sensors for fuel state, engine state and other aircraft status parameters.

Having this information, the FMC is capable of automatically performing a variety of tasks, that were traditionally the domain of flight navigators, such as management of flight plan, fuel usage, calculation of optimal flight altitudes, calculation of V-speeds etc. These can be divided into the following categories [2]:

- navigation – determination of the current geographical position of the aircraft
- flight planning – management of the route the aircraft has to follow
- trajectory prediction – computation of the flight profile along the specified route, in modern aircraft it is a 4-dimensional trajectory
- performance – prediction of aircraft performance, fuel usage etc.
- guidance – interfacing with the autopilot and PFD to automate the flight

Figure 2 shows the typical functional relationship between these subroutines, based on [2].

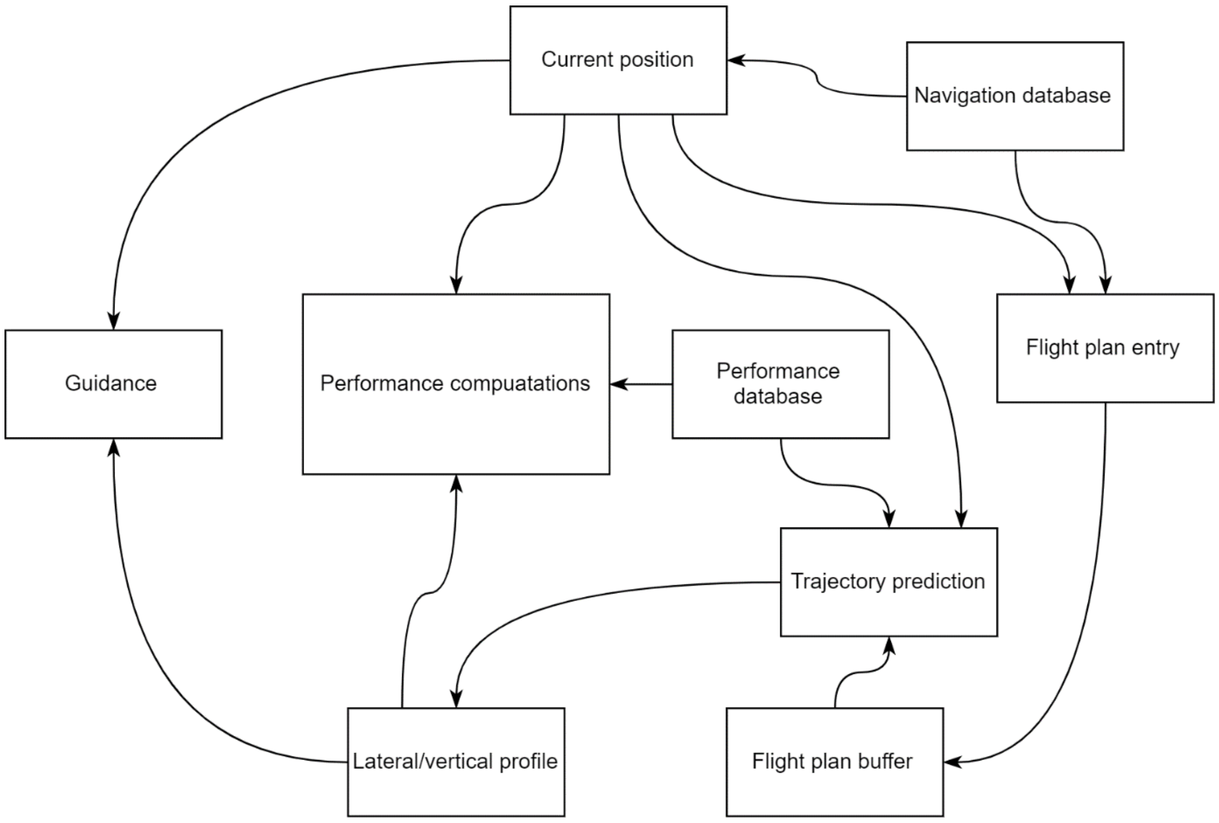


Figure 2: Functional relationships in a typical Flight Management System

2.2. Simulation environment

The environment used for this project: X-Plane 11 flight simulation software by Laminar Research, as of versions 11.00+ contains a coherently structured navigation database, stored in plaintext format. The natively supplied database is based on AIRAC cycle 1611 but it can be updated to newer versions. While the SDK functionality outdates the new database format and does not have tools to fully support it, files can be directly accessed and parsed, allowing developers to manually access the stored data.

The fix and navaid databases contain sets of features for each entity, including coordinates, frequencies and a set of identification features to uniquely distinguish them from each other [3; 4]. The airway database contains, for each segment of an airway, the ID, region code and type of the entry and exit navaid/fix, as well as additional information such as altitude restrictions and the identifier of the airway the segment is a part of [5]. In addition to those 3 files, a library of airfield specific CIFP is included, all formatted based on ARINC 424.20 standard [6].

In addition to database, the X-Plane features a Software Development Kit enabling users to add custom functionality in form of plugins. It features relatively low-level functionality, that involves, among other things:

- extraction and insertion of data into the simulation using datarefs
- basic high-level UI functions (widgets)
- low level UI functions (to be used with OpenGL programming)
- flight loops – functions that are called at a set pace based on either simulation time or the number of flight model calculation cycles done

- manipulation of objects, aircraft and scenery
- navigation data access

In the last case, the functionality currently provided in SDK is largely outdated, as it has been designed for the previous versions of X-Plane. To use the new database format custom parsers had to be developed. Another challenge faced when working with the X-Plane SDK is that it is designed to be used in C based plugins and incorporating it into the object-based C++ environment requires either substantial amounts of additional code to wrap the SDK functions into classes or allowing coexistence of procedural and object-oriented programming within the same project. In this case, the latter method has been chosen.

2.3. Simulation of flight planning

The simulation of the flight planning portion involves development of the necessary framework on top of the functionality provided by the SDK. This involves programming necessary functionality to read the database, manage the retrieved information in a meaningful way, manage and store the flight plan in memory. The flight plan consists of a series of segments between waypoints, that can be either navaids or fixes based on those navaids, as well as arbitrary geographical coordinates. The database consists of information about those fixes and navaids, as well as airways – pre-defined paths through the airspace, defined in terms of multiple waypoints.

2.3.1. Pathfinding

In order to find a path between to waypoints along an airway, the database is traversed and all segments with a matching substring are inserted into an *AirRoute* class object (see chapter 3.4.6 for details). Once the airway is collected it is either stored into memory, or if both endpoints are known a path is found between them using Dijkstra's algorithm. While a standard airway consists of a series of waypoints along a continuous, non-branching path, that could be traversed with a simple BFS implementation, this approach allows for proper traversal of any, even non-standard structure and allows to extend the system to route-finding, and general flight planning beyond typical FMC functionality.

The Dijkstra's algorithm for this task, with a goal to get through the airway following the lowest number of segments rather than actual lowest distance, can be defined as follows:

Suppose an airway is a directed graph consisting of nodes in form of waypoints, and a collection of edges between those nodes in form of airway segments. There is a single initial node, which is the waypoint we want to search for path from and a single final node we are trying to reach along the shortest possible path:

1. Mark all waypoints as unvisited, assign distance of 0 to the final waypoint and infinity (large value) to others. Add the final waypoint to the queue of unvisited nodes.
2. Check all the unvisited nodes adjacent to the current node (with edges leading to this node). For each of them, calculate the current lowest distance to the final waypoint (in this case distance stored for the current waypoint plus 1). If it is lower than the currently stored value, replace it with the new one and store the current waypoint as its predecessor along the shortest path. Add each of them to the unvisited queue
3. Remove the current node from queue and move to the next one

4. If the current node is the starting waypoint, store it as the beginning of the resultant path and back-track to the final waypoint by visiting the stored predecessor of each waypoint in order. We obtain the shortest path traversal of the airway, the algorithm is finished
5. Otherwise go to step 2.

2.4. Simulation of guidance

To provide a minimum of FMS functionality basic lateral navigation had to be implemented. In order to successfully navigate a lateral path, consisting of a series of geographical coordinates, the great-circle paths have to be calculated between each waypoint and the required turn radii at the set bank angle have to be considered.

2.4.1. Great-circle navigation

Since waypoints are situated on the surface of a spheroid rather than a flat surface, the legs are circles intersecting the centre of that spheroid rather than straight lines. Knowing the latitude and longitude of two points on said spheroid we can calculate the initial and final azimuth α_1 and α_2 using formulae:

$$\alpha_1 = \arctan\frac{\sin(\lambda_2 - \lambda_1)}{\cos\phi_1 \tan\phi_2 - \sin\phi_1 \cos(\lambda_2 - \lambda_1)}$$

$$\alpha_2 = \arctan\frac{\sin(\lambda_2 - \lambda_1)}{-\cos\phi_2 \tan\phi_1 + \sin\phi_2 \cos(\lambda_2 - \lambda_1)}$$

Where:

λ_1, λ_2 – longitudes of the first and the second point

ϕ_1, ϕ_2 – latitudes of the first and the second point

To obtain the distance between two points, we can use a modified version of Vincenty's inverse formulae for geodesic on ellipsoid [7]; while these formulae can produce results of submillimetre accuracy, when iterated using change in longitude difference for convergence, we can also obtain quick, less accurate solutions by ignoring oblateness ($f = 0 \Rightarrow a = b = R_1 \therefore U_1 = \phi_1 \wedge U_2 = \phi_2$) and performing only a single pass with a spherical Earth assumption ($\lambda = L$). In such case, the formulae reduce to:

$$\sin^2\sigma = (\cos\phi_2 \sin L)^2 + (\cos\phi_1 \sin\phi_2 - \sin\phi_1 \cos\phi_2 \cos L)^2$$

$$\cos\sigma = \sin\phi_1 \sin\phi_2 + \cos\phi_1 \cos\phi_2 \cos L$$

$$\sigma = \arctan\frac{\sin\sigma}{\cos\sigma}$$

$$s = R_1\sigma$$

Where, as per [7]:

f – oblateness of the Earth

a, b – semi-axes of the Earth

R_1 – mean Earth radius

U_1, U_2 – reduced latitude, $\tan U := (1-f)\tan\phi$

L – difference in longitude ($\lambda_2 - \lambda_1$)

s – length of the geodesic

λ – difference of latitude on an auxiliary sphere

σ – arc length on an auxiliary sphere

The above yields ground distance with reasonable accuracy. If vertical navigation were to be implemented and the results obtained here used for fuel calculations, the general formulae taking oblateness into account could be used to obtain greater accuracy, assuming a WGS84 ellipsoid. The altitude would then have to be incorporated into the formulae as well: one way to do this would be by using an auxiliary ellipsoid with semi-axes longer by the flight altitude to calculate cruise distances. For climb and descent, distances could be then obtained with simple trigonometry. Alternatively, vector calculations could be performed instead.

2.4.2. Turning performance

When navigating the path, predictions of aircraft's turning radius must be made to start the manoeuvres on time. Since the turn rate is independent of speed when flying at a fixed altitude, we can calculate the turning radius of the aircraft at the given speed using the formula:

$$r = \frac{V_g^2}{g\tan\theta}$$

Where:

V_g – ground speed

g – gravitational acceleration

θ – bank angle

Having that, we can predict the distance to the waypoint at which the turn has to be initiated:

$$d = \tan\left(\frac{1}{2}\Delta\alpha\right) * r$$

Where $\Delta\alpha$ is the change in course. An appropriate subroutine would then run every frame of simulation or at a fixed timestep, to test it against the current distance to the waypoint and switch to the next one in flight plan once it is higher (or when it is predicted to be higher at the next timestep).

2.4.3. Course control

To guide the autopilot along a from-to course (e.g. along the airway), a simple control rule can be used to set the magnetic heading to hold:

$$\alpha_h = \alpha_2 - \delta + (\alpha_2 - \alpha_1) - \beta = 2\alpha_2 - \delta - \alpha_1 - \beta$$

Where:

α_h – magnetic heading to hold

α_1 – true heading from the previous waypoint at current position

α_2 – true heading to next waypoint at our current position

δ – magnetic declination

β – drift due to wind

For high deviation from the course, a perpendicular path is flown first, and interception begins once the difference between outgoing and incoming headings reduces below a certain threshold.

2.5. Simulated architecture

Since the simulation involves primarily the software portion of FMS, the general architecture of the plugin is similar to the simulated portion of a real FMS, except that most of incoming and outgoing data is represented by functions provided by the X-Plane API. Theoretically, a real FMC could work with the simulator provided an appropriate interface (simulating real inputs and outputs involved in FMC operation) was available. Figure 3 shows the general architecture of the plugin.

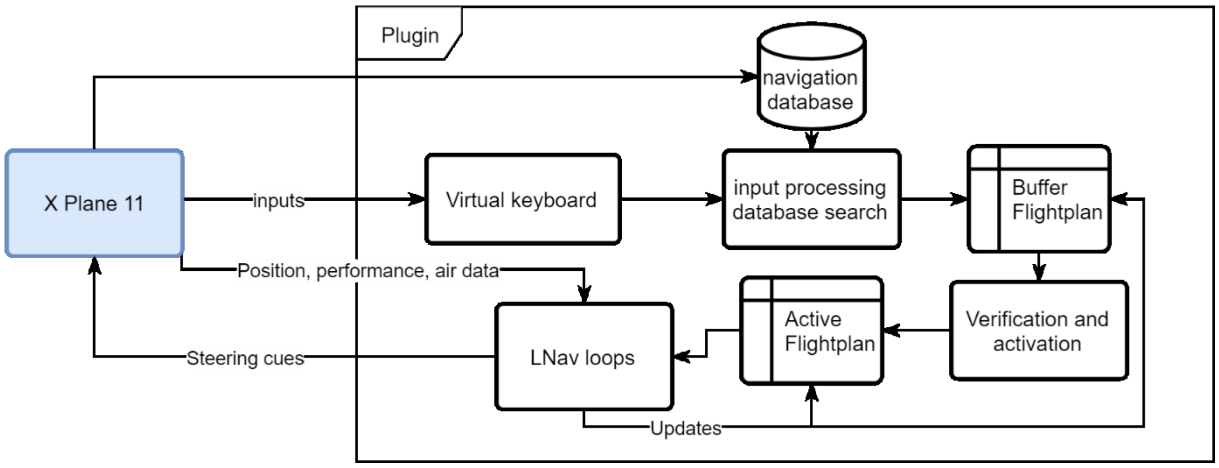


Figure 3: General architecture of the plugin

3. Implementation

3.1. Class relationship

The class structure of this software can be separated into the data management part and the UI management part. The former contains a set of classes used for storage and management of flight plan and the latter manages formatting and input/output of the UI. Both are referenced by non-member functions that manage communication with the C-based X-Plane SDK. Direct relationships between these two groups is limited.

The class relationship diagrams are included in Appendix A. The first page includes classes used in back end (navigation data and flight plan management), the second page includes classes used in front end (UI).

3.2. Initialization sequence

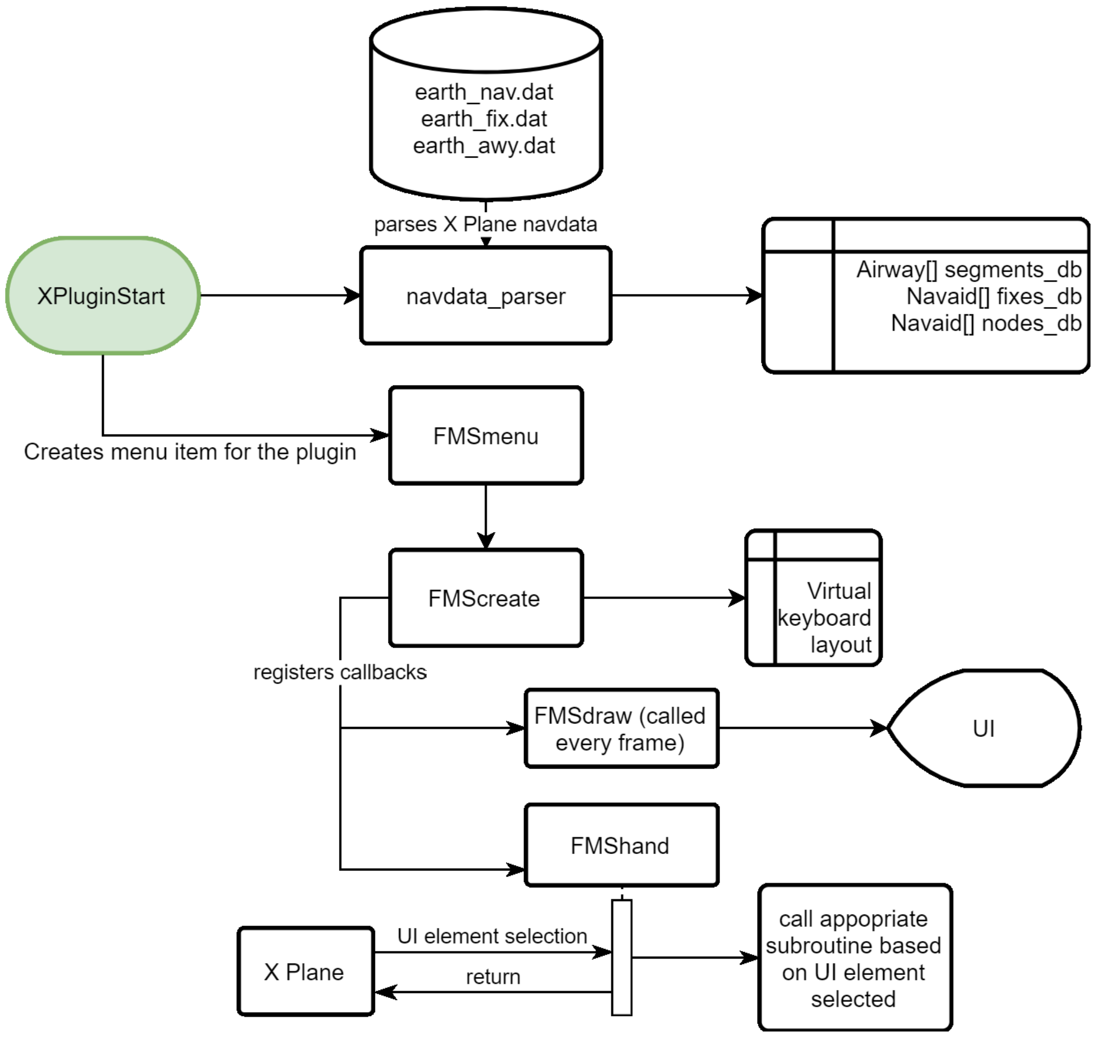


Figure 4: Flowchart of plugin initialization

Figure 4 shows the initial sequence of actions performed by the plugin when it is first called by X Plane. Firstly, the *XPluginStart* function is called (a mandatory signature must be kept for this function for X Plane to correctly interface with the plugin). The function first uses SDK functions to create a menu item for the plugin, assigning *FMSmenu* as a function for X Plane to call when the menu option to launch the plugin is accessed. Next the *navdata_parser* function is called. This function parses the navigation database and loads it formatted into the memory for quick access.

The *FMSmenu* function calls *FMScreate* which defines the widgets for UI and registers *FMSdraw* and *FMShand* as callbacks for CDU screen drawing and input handling respectively. This is the first stage of initialization.

The second stage of initialization is deferred to run after the aircraft is loaded (Figure 5). *XPluginReceiveMessage*, another mandatory function that is run every time X Plane interfaces with the plugin, is set up to react when the aircraft is loaded by registering all callbacks used for discrete calculations (LNav guidance, position updates etc.) as well as a callback called *DeferredInitACLOAD*, which is set up to only run once. This function in turn calls *initialize_interface*, which defines all the CDU pages and their contents, and *initialize_data* which maps datarefs – the interfaces used by X Plane for data exchange with plugins and outside programs, defined through a set of text strings – to variables that can be accessed and updated using API functions when necessary.

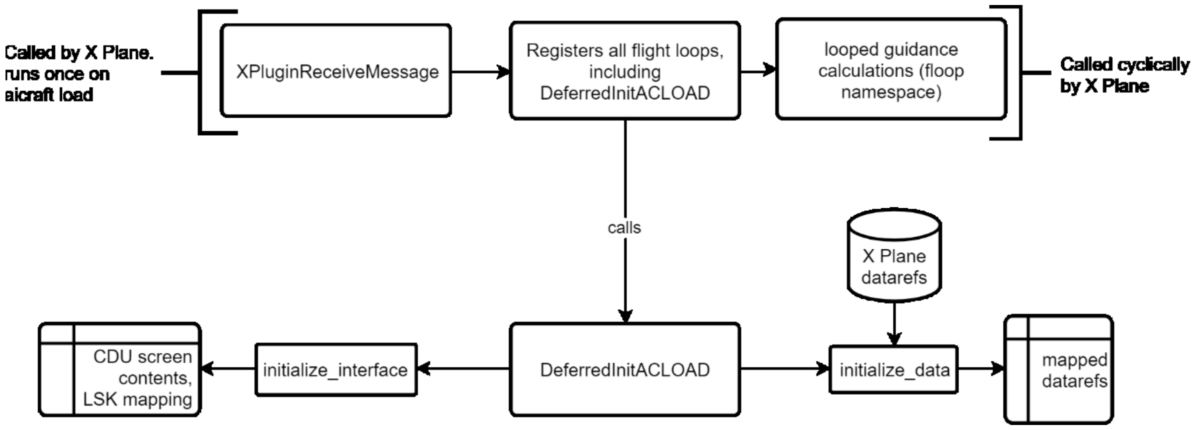

Called by X Plane. runs once on aicraft load
XPluginReceiveMessage
Registers all flight loops, including DeferredInitACLOAD
looped guidance calculations (floop namespace)
Called cyclically by X Plane
calls
X Plane datarefs
CDU screen contents, LSK mapping
initialize_interface
DeferredInitACLOAD
initialize_data
mapped datarefs

Figure 5: Deferred initialization

3.3. Data flow

While some of the functions run automatically or in a cycle, called by X Plane either every frame or in discrete timesteps, most of the functionality is invoked by user input. The input is performed through the UI in form of a virtual CDU keyboard and depending on which button is selected one of the following scenarios will happens:

- Alphanumeric keyboard presses – the letters and/or numbers are entered into the scratchpad for later use by other functions and/or deletion by the user
- Page button presses – changes currently displayed page or – in case of Next/Previous page buttons – changes displayed subpage
- Line Selection Key (LSK) presses – context-based action. Each subpage definition contains an array of function pointers that are linked to line selection keys. When an LSK is pressed, this function is invoked. The functions available range from simple data entry to context-based navigation to other pages.

- LNav switch – permits flight loops directly responsible for guidance to write to autopilot datarefs (note: some aircraft will not respond due to using different-than-standard datarefs)
- EXEC key – linked to a global function pointer not directly tied to any page. Usually it is the *activate_flightplan* function, which starts any flight loops that were not running and copies the flight plan buffer (*current_flightplan*) to the active flight plan used for guidance (*active_flightplan*)

Figure 6 shows the data flow and call order of the plugin.

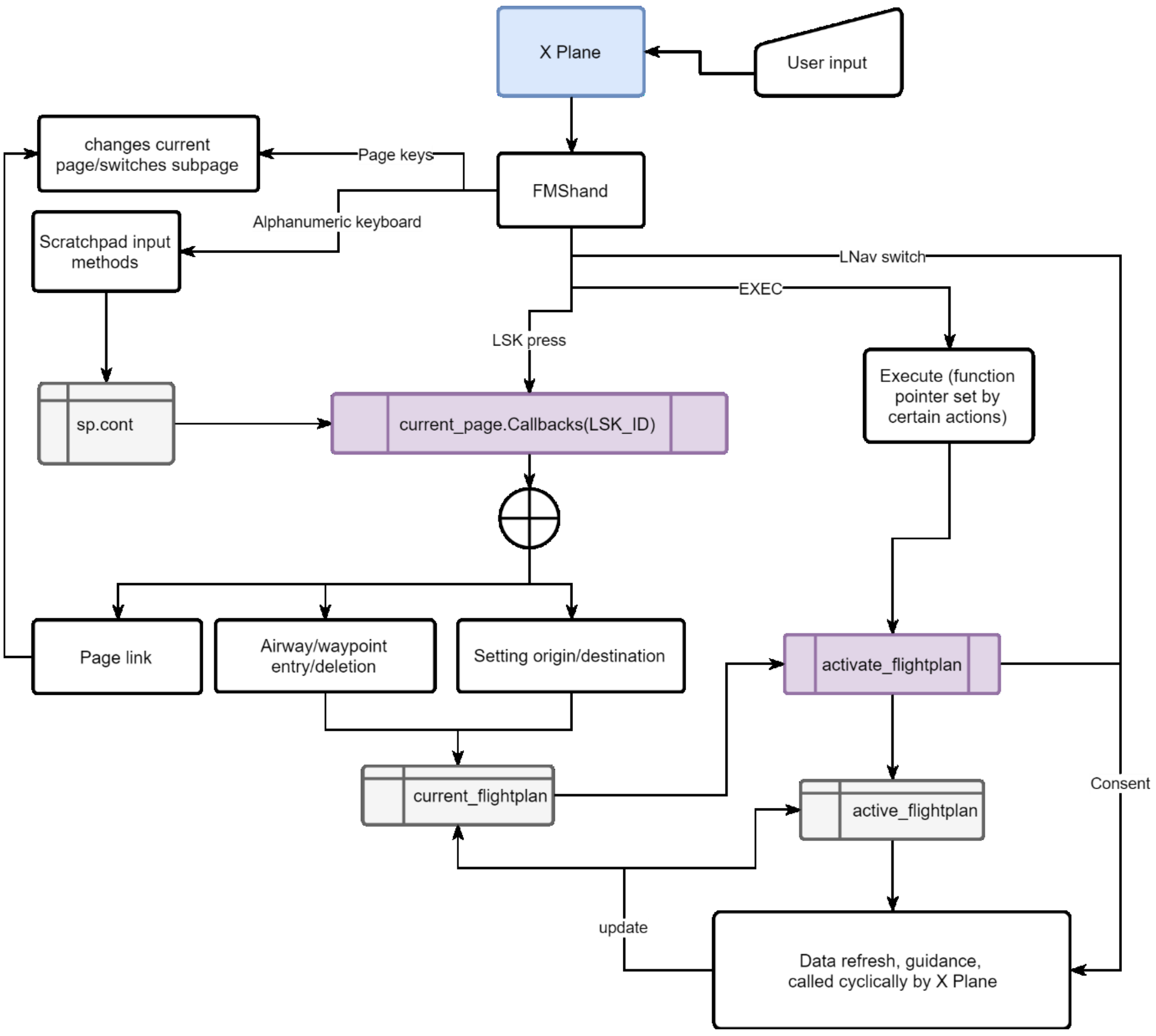

X Plane
User input
changes current page/switches subpage
Page keys
FMShand
Alphanumeric keyboard
Scratchpad input methods
LNav switch
EXEC
LSK press
Execute (function pointer set by certain actions)
sp.cont
current_page.Callbacks(LSK_ID)
Page link
Airway/waypoint entry/deletion
Setting origin/destination
activate_flightplan
current_flightplan
active_flightplan
Consent
update
Data refresh, guidance, called cyclically by X Plane

Figure 6: General data flow and call order

3.4. Class implementations

3.4.1. Azimuth

The Azimuth class is purpose-written to store and operate on azimuth angles. While regular angular operations permit arbitrary angles, including values lower than null and higher than full angle, all operations on geographical azimuths must be performed $\bmod 360$ (or $\bmod 2\pi$ when operating in radians). The class is designed to automatically take care of overflow/underflow beyond the 0-360° during mathematical operations and support both radians and degrees as the unit dynamically.

Data members

_heading – protected data member storing the single-precision floating point geographical azimuth. The class is designed so that this member is only accessed from outside class's scope by referring to the object itself, or using methods for units conversion

Unit – public, enumerated type derived from integer to store the information about the unit used, can be "deg (0)" for degrees or "rad (1)" for radians

_mode – protected Unit type data member storing the current unit the object operates on

Methods

Trim() – protected method that constraints *_heading* to the domain on call. It calculates lowest positive congruence of the data member with appropriate modulus ($\bmod 360$ for degrees, $\bmod 2\pi$ for radians):

```
void Azimuth::Trim()
{
  if (_mode == deg)     //for degrees: constrain to modulo 360
  {
    while (_heading < 0.) _heading += 360.;
    while (_heading > 360.) _heading -= 360.;
  }
  if (_mode == rad)     //for radians: constrain to modulo 2pi
  {
    while (_heading < 0.) _heading += (2.*PI);
    while (_heading > (2.*PI)) _heading -= (2.*PI);
  }
}
```

GetRad() – public method that returns the value of azimuth in radians as a single precision floating point, regardless of currently selected unit. If necessary, a conversion is made without changing the unit of stored value.

GetDeg() – same as above, except it returns the value in degrees.

SetMode(Azimuth::Unit mode) – takes a *Unit* type argument, and sets the unit of the azimuth to this value. If the new unit is different than the previous one, a conversion is performed, otherwise the value of *_heading* remains unchanged.

Rev() – returns reverse azimuth.

AbsDiff(Azimuth a1, Azimuth a2) – returns the acute angle between two azimuths, non-member function.

Constructors and operators

A number of special constructor and operator methods have been prepared to achieve the aforementioned functionality. Besides regular copy constructors, a copy constructor taking floating-point arguments has been prepared, allowing to initialize the *Azimuth* class objects directly with floating-point values. Additionally, an implicit conversion operator has been prepared for the float type, but it might be removed in future, if it is unnecessary (to prevent instability). Special arithmetic operators have been prepared that preserve the range of allowed values by calling *Trim()* when necessary, as well as allow mixed operations with floating-point types. Finally, the comparison operators have been re-purposed to show whether the left-hand side operand is within the given range of azimuths relative to right hand side, as the concepts of greater and lower are not very useful for course comparisons. The implementation of the latter is shown below:

```
bool operator<(const Azimuth& lhs, const Azimuth& rhs)
{
  Azimuth temp = lhs;
  temp -= rhs;
  temp.SetMode(Azimuth::deg);
  if ((float)temp > 180.) return true;
  else return false;
}
```

3.4.2. Coord

This simple data structure is used to keep geographical coordinates in a single object, for practical reasons. It does not feature any functionality to safeguard the values from going beyond range, as its use is very limited and not strictly necessary.

Data members

Unit – public, enumerated type used to specify the unit (radians or degrees), identical to the one in *Azimuth*.

_mode – same as in *Azimuth*

lat, *lon* – latitude and longitude as floating-point variables, public

Methods

Besides a standard constructor and another one to construct the object with latitude and longitude represented as floats, the structure features a *SetMode()* method that differs from the one in *Azimuth* only in performing operations on one more data member (both latitude and longitude have to be converted):

```
void Coord::SetMode(Unit mode)
{
  if (mode == _mode) return;
  else
  {
    if (mode == rad)
    {
          lat = lat RAD;[1]
          lon = lon RAD;
        }
        if (mode == deg)
        {
          lat = lat DEG;
          lon = lon DEG;
        }
        _mode = mode;
    }
}
```

Additionally the following methods are implemented:

GetString() – returns an STL string representation of the data stored, complicit with the format used in 737 CDU (HDD°MM.M HDDD°MM.M[2]):

SetByString(const string& input) – takes a string containing the coordinates in aforementioned format and writes the data in decimal format into the respective data members. Sets *_mode* to deg.

3.4.3. Aircraft

This structure carries the current status information for the aircraft. It is exclusively a data holding entity and contains no methods of its own. The member names are mostly self-explanatory, and the source of data is, in most cases, supposed to be the simulation datarefs.

Data members

airspeed – the current airspeed in knots of the aircraft, to be used with the datarefs

fuel_state – current fuel state, from datarefs

max_fuel – fuel capacity of the currently selected aircraft

mass – empty mass of the aircraft

max_mass – MTO

heading – current heading from datarefs

next_heading, previous_heading – headings to the previous and next waypoint, calculated by LNAV

lift, drag – aerodynamic forces, currently empty (no documented method of extracting these data from the simulation effectively)

lon, lat – geographic coordinates, taken from datarefs (might be salted to simulate INS drift/GPS inaccuracy in future)

alt – altitude, feet, taken from datarefs

windspeed – calculated windspeed

winddir – calculated wind direction

[1] RAD and DEG are pre-processor definitions for, respectively, multiplication and division by π/180

[2] H – hemisphere, D – degrees, M – arc minutes

3.4.4. Navaid

The *Navaid* class/structure is used to store navaid and fix information. The class features a number of constructors and a specialized stream operator for reading the native X-Plane 11 database.

Data members

type – uses an enumerated type *NavTypeNew* (see implementation) to describe the type of navaid, using a numbering system identical to the one in X-Plane 11 database (as per row code specifications in XP-NAV1100). A *NavType* enumerated type has also been implemented based on the X-Plane 10 numbering, but it is left unused

lat, lon – latitude and longitude, floating point

heading – stores localizer heading for localizers, slaved variation for VORs, otherwise unused

elev – integer, stores elevation in feet above MSL

range – integer, maximum reception range for navaids

freq – integer, radio frequency of the navaid

id – uniqe/locally unique identifier for the navaid

regid – terminal region identifier, usually either ICAO code of an airfield or ENRT for navaids unasociated with airfields

reg – region code as per ICAO No. 7910 (per file format specification)

name – short description or name of the navaid

Methods

In addition to standard constructors, two comparison operators (equals and does not equal) have been specified. Most importantly, a custom *istream* operator has been created to read the Earth_nav.dat and Earth_fix.dat files containing the native X-Plane 11 navigation data, based on file format specifications [3; 4].

3.4.5. Airway

This class consists of data members and methods written to handle the native X-Plane 11 airway database. Individual objects of this class hold information about a single navaid to navaid segment of an airway. Similarly to the *Navaid* class, this class is built around the format used in the aforementioned database, as described in [5] and contains a purpose build stream operator for their extraction.

Data members

entryid, *exitid* – private members containing the identifiers (as per their description in Navaid section) of entry and exit navaids, only for initial readout of the database

entryreg, *exitreg* – as above but for the region code, private

entrytype, *exittype* – types of navaids at the endpoints, private

awy – private member containing a hyphen-delimited string with names of airways containing this segment *name* – a string containing the segments individual name, usually empty, public

entry, *exit* – public members, pointers to the Navaid type objects loaded into the memory. Used to access the entry and exit navaids directly through this object. Upon construction of the object (usually when the database is initialized from Earth_awy.dat) these pointers are NULL (to save processing time needed to search the navaid and fix databases every time a segment is loaded) and have to be mapped using the *MapNavaids()* method when they are needed.

dir – public member, a single character determining the directional restriction of the segment. 'F' means that the segment can only be traversed from entry to exit, 'B' means it can only be traversed from exit to entry, 'N' means no directional restrictions [5]. Crucial for pathfinding.

cls – airway class, 1 for low, 2 for high, as classified within X-Plane environment [5]

base, *top* – minimal and maximal flight levels to fly this segment at, limited usage for now (no VNAV)

Methods

MapNavaids() – searches the database for exit and entry navaid objects and stores pointers to them in entry and exit members. Returns true if the navaids are found and successfully mapped, false if they are not (indicating error within the database)

CheckAwy(const string& iawy) – checks if the segment is a part of the "iawy" airway. Searches the "awy" member and returns true if found (the segment is part of the airway in argument), false if not found (the segment is not a part of this airway). Operation based on Knuth-Morris-Pratt algorithm

GetExit(const string& entry) – takes a reference to a Navaid object as an argument and if it is an end point of this segment, it returns a pointer to the other end if the airway can be traversed in this direction, otherwise returns NULL

Additionally, a custom stream operator had been implemented to read the Earth_awy.dat, based on file format specification [5].

3.4.6. AirRoute

The *AirRoute* class is used to store and operate on entire airways, consisting of multiple segments. It contains necessary methods to find the path through an airway. It is crucial for the route management within the software.

Data members

vertices – a vector of Navaid type objects, stores the list of navaids and fixes that constitute the vertices of the airway, private member

adjecents – an adjacency list for Dijkstra's pathfinding algorithm (see *BFS()*), private member

name – a string with the name of the *AirRoute* object, usually the airway identifier

route – a vector of Airway type objects containing all segments this airway consists of

Methods

BFS(start, end) – takes two *Navaid* type objects and finds a path through this airway between them using Dijkstra's algorithm. Returns a vector of Navaids sorted in the order of traversal. A more detailed description of this algorithm is available in previous chapter (2.3.1). The following listing contains key fragments of the code:

```
//declarations, initial setup

std::queue<int>to_check;     //queue of vertices to explore

int iEnd;                    //will hold the final index

for (iEnd = 0; iEnd < vertices.size() && vertices[iEnd] != end; iEnd++);
  //find the index of endpoint

(…)

visited[iEnd] = true;        //mark endpoint as visited
distance[iEnd] = 0;          //set shortest path to this point to 0
predecessor[iEnd] = iEnd;   //set predecessor for endpoint to itself
to_check.push(iEnd);         //push it into the queue to explore
while (!to_check.empty())    //until the queue is empty
{
  current = to_check.front(); //set the front of the queue as the currently
evaluated vertex
  to_check.pop();            //remove it from queue
  for (int i = 0; i < adjecents[current].size(); i++) //go through its
adjacency list
  {
    if (!visited[adjecents[current][i]])  //for each unvisited adjacent
    {
      to_check.push(adjecents[current][i]);    //push it into queue
      visited[adjecents[current][i]] = true;   //mark as visited
      if (distance[adjecents[current][i]] > distance[current] + 1)
      {
        //update shortest path from endpoint if necessary
        distance[adjecents[current][i]] = distance[current] + 1;
        predecessor[adjecents[current][i]] = current;
      }
    }
  }
}

//go through the predecessor list to collect the shortest path to endpoint
```

Segment(vector<Navaid> traversal) – takes a vector of *Navaid* type objects containing the traversal order of the airway (obtained from BFS(...)) and returns a vector of *Airway* type objects in the same order.

CheckNavaid(Navaid find) – takes the Navaid type argument "find" and searches the vertices vector for it, returns true if found, false if not found.

CheckNavaidID(string find) – same as above, except it takes a string containing the navaid ID rather than the actual object.

3.4.7. Leg

The *Leg* class represents a single leg of the flight plan, i.e. a set of data required to successfully navigate a segment of the flight plan. It makes use of multiple data structures and classes defined up to this point.

Data members

waypoint – a Navaid class object representing the target waypoint of this leg

segment – an Airway class object representing the airway segment to fly via, if the leg is direct to a waypoint, a "direct" placeholder segment is used

airway_id – a string containing the name of the airway that this leg is situated in ("DIRECT" for a direct path

restriction – the altitude restriction for this leg of the flight plan

restrict_rel – uses enumerated type *Rel* that can be either A = +1 (above), E = 0 (exact altitude to fly) or B = -1 (below), determines the direction of the altitude boundary established by *restriction*. Both members have been prepared for V-NAV support, so are scarcely used

elevation – exact altitude to fly at as determined by the user, currently of limited use

distance – floating point, length of the leg in nautical miles or remaining distance to fly along this leg

_isWaypoint – private, Boolean value, marks the leg to be displayed in the ROUTE page

_discontinuity – private member, marks path discontinuity at this leg

Methods

isWaypoint() – returns whether *_isWaypoint* flag is set

isDiscont() – returns whether *_discontinuity* flag is set

SetDiscont() – sets the *_discontinuity* flag as true

SetCont() – sets the *_discontinuity* flag as false

3.4.8. Flightplan

The *Flightplan* class is the core of the flight plan management in this software. It is the representation of the flight plan and all related data necessary for navigation. It includes most of the methods involved in management, entry and execution of the flight plan.

Data members

_enroute – Boolean, private member, set to true when the aircraft begins following the flight plan

_discontinuous – Boolean, private member, used to keep track of absence of discontinuities in the path

name – string, name of the flight plan, can be empty

legs – a double-ended queue storing the actual flight plan in form of Leg class objects *stored_awy* – a vector of *AirRoute* type objects storing mapped airways retrieved from memory that have been inserted into the flight plan without defining endpoints

origin, *destination* – Navaid class objects representing the origin and destination airports

Methods

This class features the highest amount of methods out of any class in this software, as most functionality related to flight plan management is concentrated here.

Origin(Navaid orig), Destination(Navaid dest) – sets the origin or destination to the argument

GetDestName(), GetOrigName() – returns a string containing the ID of the origin or destination (the ICAO code of an airfield)

Next() – switches to the next waypoint, removes the previous waypoint from the queue. If *_enroute* flag is false, it sets it to true

Wipe() – clears the *Flightplan* object of all data:

IsEnroute() – returns the value of the *_enroute* flag

RecallRoute(string name) and *RecallRoute(int index)* – a pair of overloaded methods that return a reference to an *AirRoute* class object stored in *stored_awy*, based on, respectively, name (ID) of the airway and the index of the element in legs that contains the required *airway_id*. The latter is defined in terms of the former, but it is the one used most often. If the *AirRoute* is not found in *stored_awy*, a reference to a placeholder “invalid” *AirRoute* class object is returned instead:

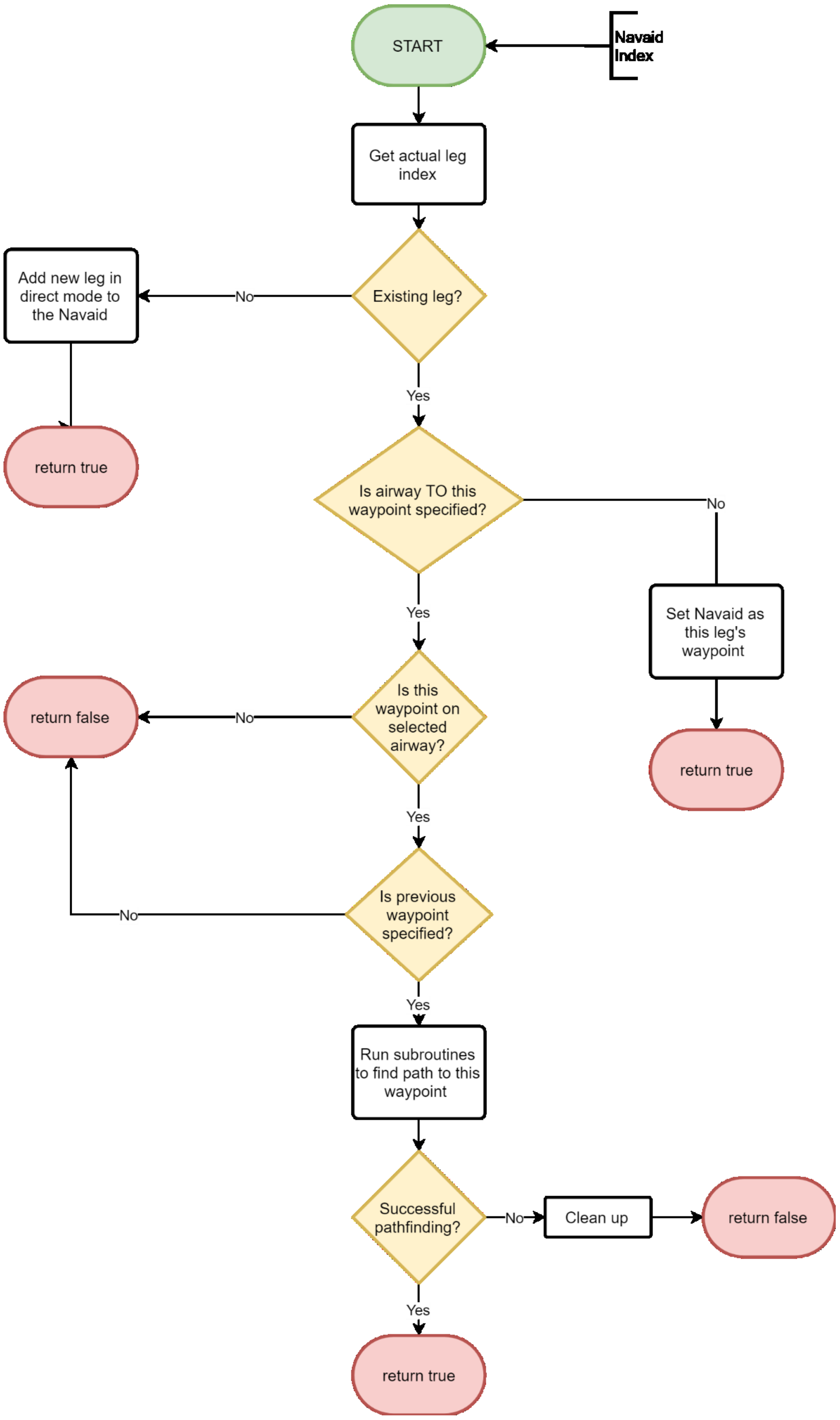
```
AirRoute& Flightplan::RecallRoute(std::string name)
{
  if (stored_awy.empty())return invalid;
  std::vector<AirRoute>::iterator it;
  for (it = stored_awy.begin(); it != stored_awy.end(); ++it)
  {
    if ((*it).name == name) return *it;
  }
  return invalid;
}

AirRoute& Flightplan::RecallRoute(int index)
{
  //index = FindWpIndex(index);
  std::string airway_name = legs[index].airway_id;
  return RecallRoute(airway_name);
}
```

FindWpIndex(int index) – returns the actual index to the legs deque of a waypoint as seen in the ROUTE page. Since the legs defined in ROUTE are stored within the same double-ended queue as those displayed in LEGS, a method that takes the index of the leg as it appears in the ROUTE page and returns its actual index in the container had to be defined. The function traverses the “legs” deque once, counting the legs with the *_isWaypoint* value set to true, and returns the current index to “legs” once it reaches the requested *_isWaypoint* == *true* leg. If the amount of these legs is lower than the index requested, it returns the size of *legs*.

GetWp(int index) – uses the method defined above and returns the reference to the element in *legs* found under that index.

AddWaypoint(Navaid wp, int index) – one of the most important methods of this class. It takes a *Navaid* class object to manage and the index of the waypoint to enter per the order on ROUTE page. It inserts the waypoint into the flight plan, creating a corresponding *Leg* class object (or updating an existing one) and managing the order of legs to accommodate the new entry. If airway out/in and the other endpoint are specified, the *AddAirway* method is called to update path. Returns true if the waypoint is added successfully, false if it is not. It is meant to be used in conjunction with the non-member *select_waypoint* function, defined later in this document (3.5.1).

Figure 7 shows the flowchart of the *AddWaypoint* method:



START
Navaid Index
Get actual leg index
Existing leg?
No
Add new leg in direct mode to the Navaid
return true
Yes
Is airway TO this waypoint specified?
No
Set Navaid as this leg's waypoint
return true
Yes
Is this waypoint on selected airway?
No
return false
Yes
Is previous waypoint specified?
No
Yes
Run subroutines to find path to this waypoint
Successful pathfinding?
No
Clean up
return false
Yes
return true

Figure 7 *AddWaypoint* method flowchart

DelWaypoint(index) – deletes the waypoint from the list together with the paths leading from this waypoint.

AddAirway(AirRoute airway, int index) – adds an airway into the flight plan. If entry and exit are specified, the path along this airway is calculated using *BFS* method from *AirRoute*. Needs a specifically retrieved and prepared *AirRoute* object. Returns true if the airway is added to flight plan successfully, false if it is not. It is meant to be used in conjunction with the non member *select_airway* function. Figure 8 shows the detailed flowchart of this method.

DelAirway(int index, bool forget) – deletes the airway as pointed by the index (according to listing on ROUTE page) by tracing it along the path between two major waypoints. If *forget* is set to true, it erases the *airway_id* (sets to *direct*), meaning the path will not be recreated upon inserting start and end waypoints.

CheckContinuity() – traces the path searching for discontinuities, if found any it sets appropriate flags for the discontinuous legs and the *_discontinuous* variable for the entire flight plan.

IsAirwaySelected(index) – returns true if an airway is specified for the given index on ROUTE page.

IsWaypointSelected(index) – same as above for the airway.

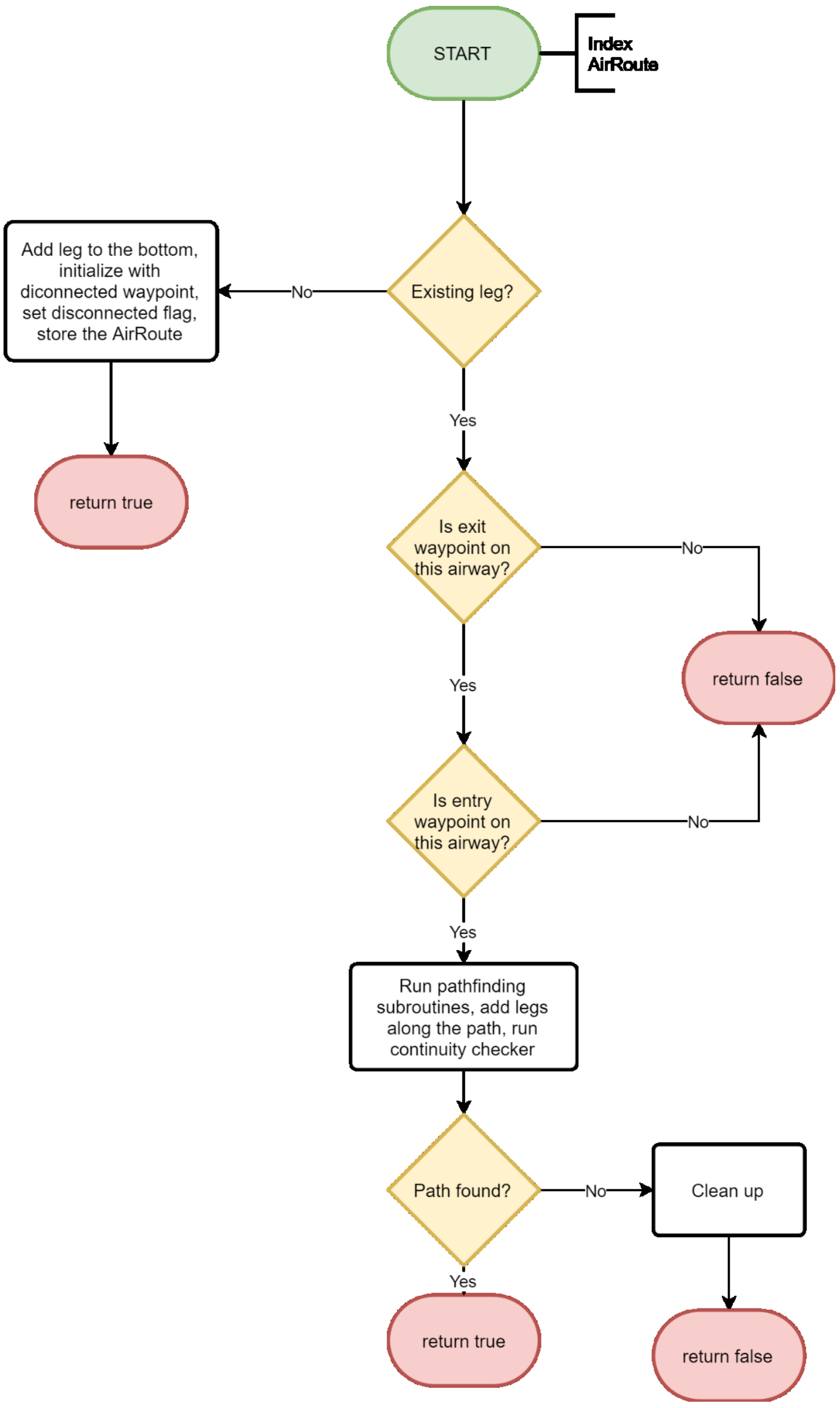

START
Index
AirRoute
Existing leg?
No
Add leg to the bottom, initialize with diconnected waypoint, set disconnected flag, store the AirRoute
return true
Yes
Is exit waypoint on this airway?
No
return false
Yes
Is entry waypoint on this airway?
No
Yes
Run pathfinding subroutines, add legs along the path, run continuity checker
Path found?
No
Clean up
Yes
return true
return false

Figure 8 *AddAirway* flowchart

3.4.9. Screen

The Screen class is part of the user interface. It represents the text displayed on the CDU screen and provides tools necessary for its entry, modification and preparation for interface with X-Plane SDK and its display in simulation

Data members

_content – a sole data member of this class, a pointer to a container of strings representing individual lines of display; upon construction of the object it is formatted into a 13-element vector holding the contents of a single screen. The "|" symbol is used to separate left- and right-justified text. The member is private and is only accessed by means of methods.

Methods

Mod, *LMod*, *RMod (int line, string mod)* – replaces, respectively, the entire content, the left-justified part and the right-justified part of the line pointed by *line*" argument with *mod* string.

Clear() – clears the entire screen, replacing all lines with empty strings

GetScrn(char (&container)[13][46]) – takes a reference to a 46/13 matrix of characters and copies *_content* into it, formatting it for display by the drawing handler:

```
void Screen::GetScrn(char (&container)[13][46])
{
  std::size_t pos;
  std::string* buffer;
  buffer = new std::string[13];

  for (int i = 0; i < 13; i++)
  {
    buffer[i].assign(_content[i]);  //make a copy of contents
  }

  for (int i = 0; i < 13; i++)
  {
    pos = buffer[i].find('|');
    if (pos != std::string::npos)
    {
      buffer[i].replace(pos, 1, " ");  //replace formatting character '|'
with enough spaces to stretch the string to screen
      while (buffer[i].length() < 45)
      {
        buffer[i].insert(pos, " ");
      }
    }
    if (buffer[i].length() > 45)
    {
      buffer[i].resize(45);   //cut the line if it's too long
    }
    while (buffer[i].length() < 45) buffer[i].append(" "); //add trailing
spaces
  }

  for (int i = 0; i < 13; i++)
  {
    buffer[i].copy(container[i], 45, 0); //copy contents onto the array
passed in argument
    container[i][45] = '\0';
  }
  delete[] buffer;
}
```

GetLine(int line) – works similarly to the previous method except that it returns a pointer to a single array of chars with the selected line formatted for display

UpdatePageAmount(int current, int number) – updates the page number in the top right of the screen to match the screen index (see *Page* class: 3.4.11) given in *current* and the total amount of pages given in *number*

ReadLine, RReadLine, LReadLine (int line) – returns the string with, respectively, the entire line, the right portion and the left portion of the row pointed by the *line* argument

Additionally, a number of constructors for initialization of the object with char arrays, vectors for strings etc. had been created, as well as an enumeration for easier identification of individual rows:

```
  enum
  {
    _lt = 0,
    _s1 = 1,
    _l1 = 2,
    _s2 = 3,
    _l2 = 4,
    _s3 = 5,
    _l3 = 6,
    _s4 = 7,
    _l4 = 8,
    _s5 = 9,
    _l5 = 10,
    _s6 = 11,
    _l6 = 12,
  };
```

3.4.10. Scratchpad

The Scratchpad class represents the scratchpad area of the screen, and is the primary class governing input into the CDU. It is designed to simulate the behaviour of the scratchpad in 737 series CDUs, which means that besides data entry it also allows for display of error messages.

Data members

_*errstatus* – private, Boolean member, set to true if the scratchpad is currently displaying an error message

_*delmode* – set to true when the scratchpad is in delete mode (after pressing the DEL button on the virtual keyboard

_*change* – set to true whenever changes are made to the scratchpad content *_output* – a 46-element array of chars containing the contents of the scratchpad formatted for display

_buffer – private, a string that the intermediate operations are performed on

cont – public member, stores the current content of the scratchpad that can be accessed by other functions.

Methods

Add(string input) – adds characters to the scratchpad. Can be anything from a single character to an entire word

Clear() – if an error message is displayed or the scratchpad is in delete mode, it clears the message from the scratchpad and sets respective flags to false. Otherwise it removes the last character

Clear_All() – clears _buffer and _output

FlipSign() – adds or removes leading minus sign, used for integer input

Del() – puts the object into "delete" mode: a "DELETE" message is displayed and *_delmode* flag is set to true; other functions can use this information to determine whether they should input or delete data

Deleted() – resets the DELETE message and *_delmode* flag

isDel() – returns true if the object is in delete mode, false otherwise

Error(string code) – allows to display an arbitrary error code in the scratchpad field

GetSP() – returns a pointer to a 46-element array (the *_output* member, to be more precise) with the scratchpad contents formatted for display:

```
char* Scratchpad::GetSP()
{
  if (_change)
  {
    _buffer.copy(&_output[1], 42, 0);
    for (int i = _buffer.length() + 1; i < 44; i++)
    {
      _output[i] = ' ';
    }
    _change = false;
    return _output;
  }
  else
  {
    return _output;
  }
}
```

3.4.11. Page

This is the class that covers the core UI functionality of the software. It represents the actual UI pages accessible with the CDU, in form of the textual contents and the functions accessible within the page by the means of text input/output and line selection keys.

Data members

_listStartLine – protected member, marks the first line of a list

_listStartPage – protected, first page of a list

_listPerPage – protected, number of items displayed per-screen

cont – a vector of *Screen* class objects representing UI subpages of a given page, every page has at least one *Screen*

functions – a vector of vectors of function pointers of signature:

```
typedef Page*(*tempf)(int select, void* input, Page* output);
```

where *select* is the LSK number, *input* is a pointer to data (usually the scratchpad), *output* is a pointer to the *Page* class object that the function is operating on. Returns a pointer to a *Page* when necessary, NULL otherwise; used to store the callback functions accessed with line selection keys, can be either member or non-member functions

callmap – a vector of vectors of pointers to other Page type objects, used for UI navigation using line selection keys

_currpage – stores the index of the currently selected screen

_modstatus – Boolean value, set to true whenever the contents of the page are modified

Methods

AddPage(), AddPage(Screen init) - two overloaded methods, one adding a new empty subpage to the page, one adding a subpage and initializing it with a Screen class object:

DelPage() – deletes the last subpage

Callback(int select, void input)* – calls a function from the *functions* vector, pointed by “select”:

```
Page* Page::Callback(int select, void* input)
{
   return (*functions[_currpage][select])(select, input, this);
}
```

AddCb(int select, int page, Page(*func)(int, void*, Page*))* – adds a callback function to *functions*, using the parameters provided as arguments.

AddRef(int select, int page, Page target)* – sets another *Page* (pointed by target) to be accessible with a line selection key.

GetSubpage() – returns currently displayed subpage

ListIndex(int select) – converts the LSK index to the corresponding line index

ListLSK(int select) – converts a line index to the corresponding LSK index

MarkList(int select, int page, int perpage) – sets the beginning of a list. A list is a series of entries associated with line selection keys, e.g. the flight plan

RefreshList(deque<string>data, int side, int line) – takes data listed in a form of a double-ended queue of strings, the side they are supposed to be displayed on (0 for left, 1 for right) and whether they are supposed to be displayed on top (subtitle) or in the main body of the line:

NextPage(), PrevPage() – switches subpage by iterating the *_currpage* variable

Additionally, the following callback functions are defined as static methods of Page class:

invalid_entry(int select, void input, Page* output)* – a placeholder callback function that is inserted into the "functions" vector at object construction, displays an error message "INVALID_ENTRY" on the scratchpad

call_page(int select, void input, Page* output)* – returns the pointer to another page stored in *callmap* under "select" at current subpage

copy_paste(int select, void input, Page* output)* – a simple copy and paste operation – copies the content from the page into the scratchpad if the scratchpad is empty, pastes it into selected line otherwise.

3.4.12. LegsPage

This is a class derived from *Page* class that features a variant of *RefreshList* prepared specifically for the LEGS page, that takes a vector of Leg class object as an argument, and prepares a leg list based on that.

3.4.13. RoutePage

A class derived from *Page* for accommodation of a variant of *RefreshList* specific to the Route page, similarly to the *LegsPage* class.

3.5. Non-member functions and data

Due to the nature of the X-Plane 11 SDK, which has been designed primarily with simple C plugins in mind, a number of functions had to be implemented as non-member. These functions are primarily responsible for communication with X-Plane and tasks that are timed in relation to simulation's physics cycles.

3.5.1. Data management

The following functions have been prepared for management of input/output operations, accessing simulator's internal data and interface for the Page class in form of non-member callback functions.

navdata_parser *(int mode, string search, Navaid* ret, ifstream* navFile)* – a multi-purpose function developed for X-Plane navigation database parsing. While X-Plane SDK features native functionality for navigation data access, its usage is limited in v11 as it has not been updated to reflect the changes in database structure. Therefore, to take full advantage of native navigation data available in X-Plane 11 a custom parser had to be prepared.

The function can operate in several different mode, accessed through the *mode* parameter setting:

0 (default) – parses the entirety of earth_nav.txt, earth_fix.txt and earth_awy.txt loading data into memory. Currently the only option used, as despite being suboptimal in terms of memory management (since the same data, albeit formatted differently, is already loaded into memory by X-Plane itself) it allows for fastest data access assuming enough memory is available, which there should be on any hardware capable of supporting the platform itself. The data extracted from these files is then accessible from *nodes_db*, *fixes_db* and *segments_db* vectors, which are declared in global scope.

1, 2 – parses exclusively through earth_nav and earth_fix respectively, searching for a navaid/fix which identifier matches with the "search" parameter, then loads it into memory.

3 – searches earth_awy, currently unimplemented

4 – searches both earth_nav and earth_fix for an ID matching the *search* parameter, if found stores it in *Navaid* pointed by the *ret* parameter. The parameter can be left as NULL in other modes

Additionally, the function can operate on an *ifstream* object defined elsewhere, as long as a pointer is provided by *navFile* parameter. If *navFile* is NULL, the function creates a new *ifstream* object instead.

In modes 1-4 the function returns 1 if the query has been found and 0 if it has not. In mode 0 the return value is always 1.

disp_byte *(Page* output, XPLMDataRef data)* – displays information retrieved from *data* argument (an X-Plane dataref) on the *output* page (line by line from the top), used for debugging.

disp_float – same as above, but exclusively for datarefs including floating point values

disp_byte_l *(int line, int subpage, Page* output, XPLMDataRef data, int length)* – allows to display a single line of arbitrary data on the *output* page. *line* and *subpage* specify where it should be displayed on that page, *data* is the X-Plane dataref with the source of data, *length* specifies how many characters should be displayed

engine_rating – this function accesses the appropriate X-Plane datarefs and updates the IDENT page with engine rating. The rating is given in force pounds for jet aircraft and HP in propellers.

select_origin, select_destination – a pair of functions of *tempf* signature (see *functions* in *Page* section for details) that search for airports in database based on scratchpad input and update the ROUTE page accordingly. Uses X-Plane SDK to access data on airfields, as CIFP are not implemented.

select_waypoint – this *tempf* function handles the interactions between the navaid/fix database, the *Page* and *Scratchpad* classes and the *Flightplan* class. It takes input from *Scratchpad*, searches the database to find the waypoint based on ID, calls the *AddWaypoint* method for the buffer flight plan stored in *current_flightplan* globally declared *Flighplan* class object and handles the returns. The exact functionality has been shown in Figure 9.

select_airway – a *tempf* function similar to *select_waypoint* that handles addition of airways to flight plan. Analogically to *select_waypoint*, this function handles interactions between *segments_db* holding the airway data, the *Page* and *Scratchpad* classes and the *Flighplan* class. The function searches the database for all segments based on the name inserted into scratchpad, assembles them into an *AirRoute* class object, calls *AddAirway* and handles return values. A detailed flowchart is shown in Figure 10.

activate_flightplan – this function copies the current buffer flight plan (*current_flightplan*) into the active flight plan (*active_flightplan*) and schedules the several flight loop functions related to guidance to run. It is expected to be accessed with the EXEC key on virtual keyboard.

read_route – reads a pre-made route from a file stored in X-Plane 11\Output\FMS plans. The route is entered by automatically executing the same sequence of function calls as the user would normally perform. The txt file containing the route must be formatted as a standard ICAO flight plan string, e.g.:

```
EPWA DIRECT LOZ DIRECT POLON T355 KURAV DIRECT EPKK
```

write_route – writes the currently entered flight plan into a txt file in X-Plane 11\Output\FMS plans, using the same formatting as above

route_storage – accessed by entering a file name into the scratchpad and selecting CO ROUTE on the ROUTE page. The function searches the folder for the file name entered by the user and calls *read_route* if found, otherwise displays error

save_file – accessed by selecting SAVE FILE> on the ROUTE page. Creates a new file with the name inserted into scratchpad (or sets to overwrite if the file already exists) and calls *write_route*.

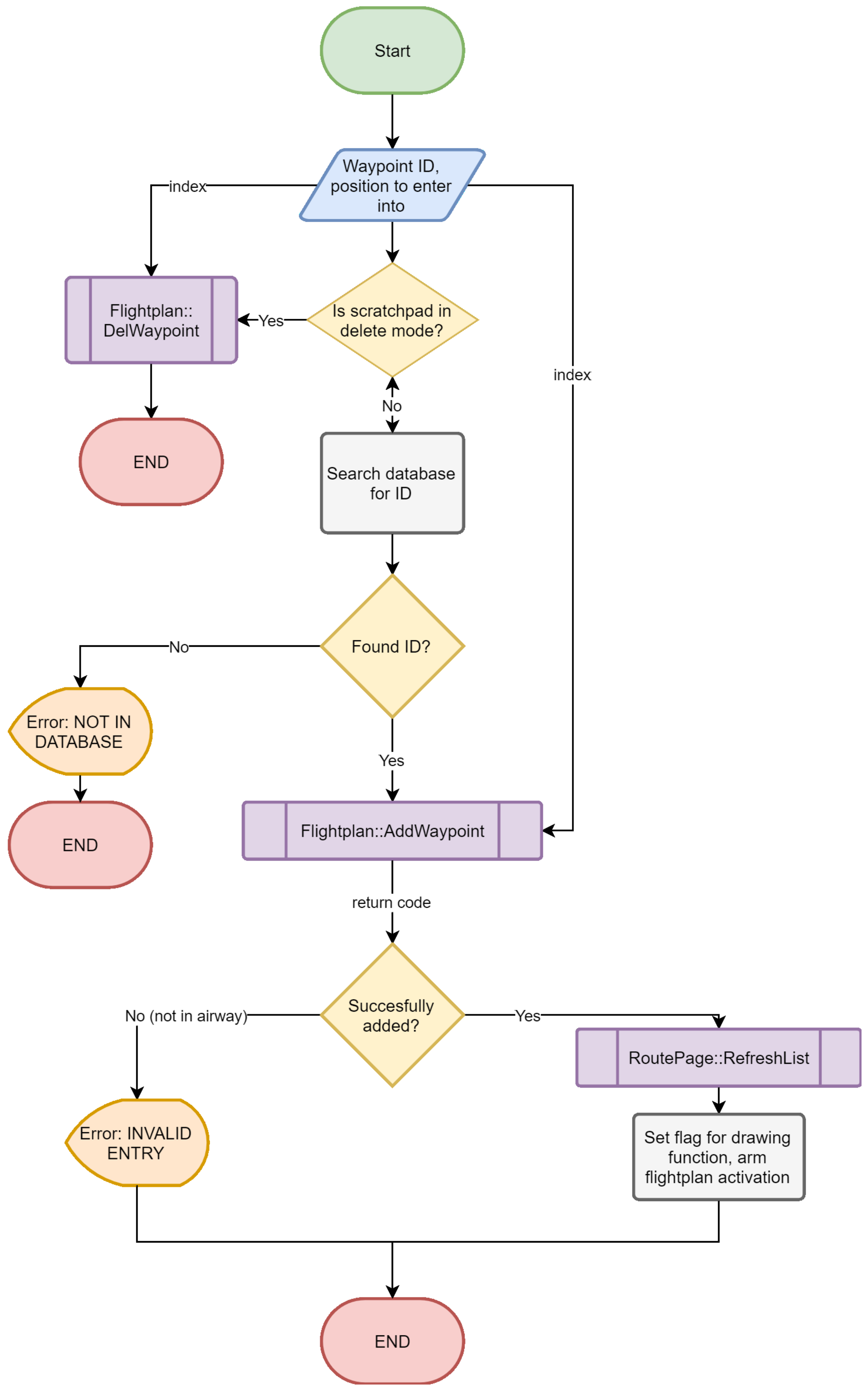


Figure 9: *select_waypoint* flowchart

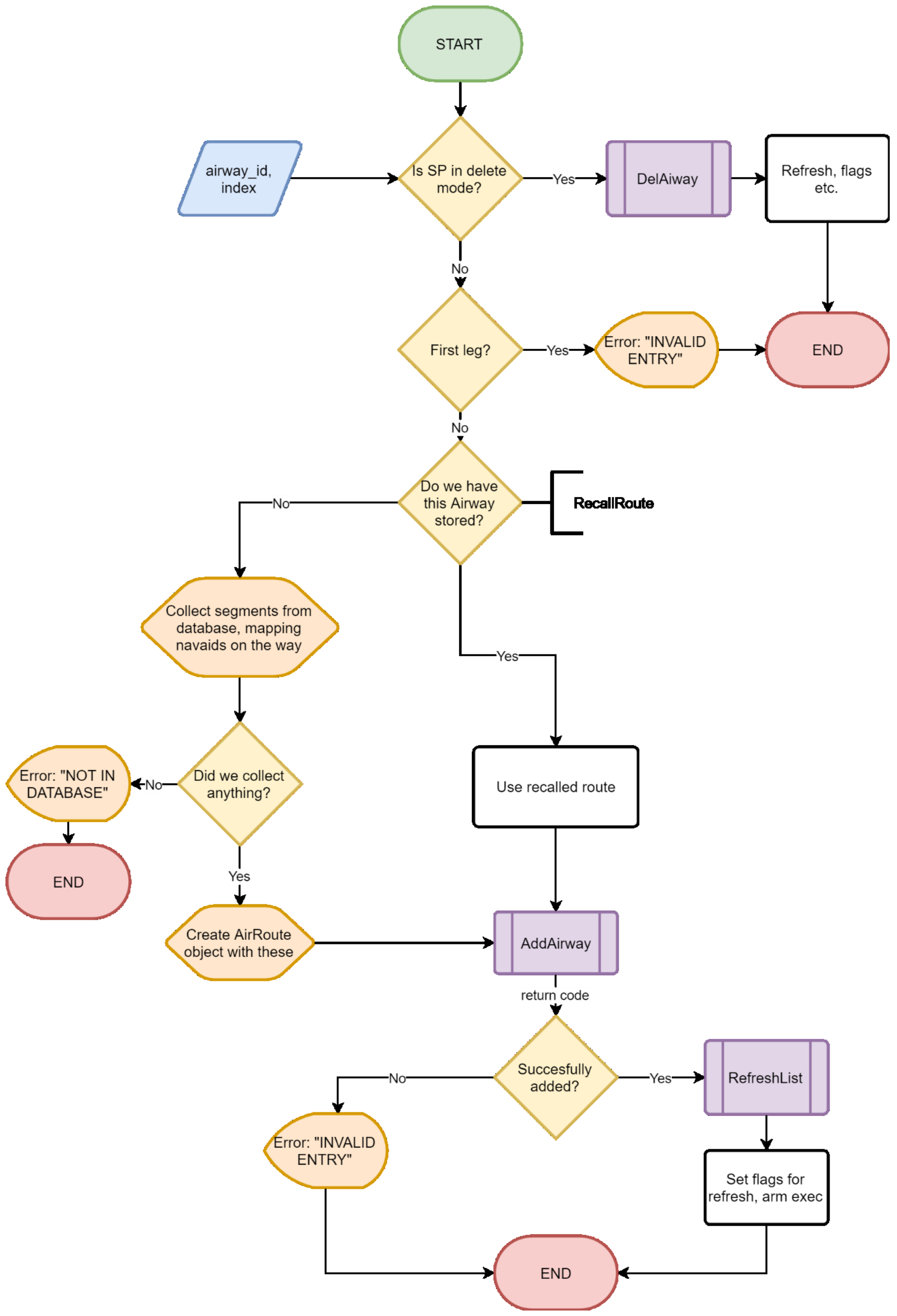


Figure 10: *select_airway* flowchart

3.5.2. User Interface functions:

The following functions have been developed to handle the UI. Most of these have signatures precisely defined by the X-Plane SDK and are called by the simulator based on user input. Not all related functions have been described in this section, as some of them are purely formal, required by SDK.

FMScreate – this function defines the entire layout of the UI and is called by the program on plugin initialization. The implementation is included in the appendix, <Figure> shows the resultant graphical layout.

initialize_interface – this function defines the layout of all the UI pages of the CDU. Their exact layout is shown in the next chapter.

FMSdraw – retrieves formatted data from relevant Page and Scratchpad class objects and displays it on the screen

FMShand – defines functionality of all keys on the virtual CDU keyboard

3.5.3. "Flight loop" functions

Flight loops are the C functions with a precise signature allowing them to be called by the simulation in a scheduled, time-based manner. These functions handle all the tasks that have to be performed constantly thought the flight, such as physics calculations, distance calculations and guidance. For organizational purposes, they are all defined within *floop* namespace:

GetPostion – this function updates the aircraft geographical coordinates and altitude, stored within globally declared *Aircraft* class object *aircraft*. It also updates the information on POS INIT page accordingly. The position is currently pin-point accurate, but IRS drift and other inaccuracies can be coded within this function.

LNav – performs necessary calculations for lateral guidance. Updates great circle heading and distance to next waypoint, and if L-Nav is engaged writes course guidance cues into relevant autopilot dataref.

updatenextleg – calculates cumulative distances and headings on the legs down the route. Updates a single leg every time it is called.

legsUpdate – updates the LEGS page with new data

turn – calculates current turn radius based on current airspeed and the autopilot bank angle setting, switches to next waypoint once the distance to next waypoint falls below this value. For turns tighter than 150 degrees a planned overshot is performed.

3.5.4. Other functions

The following three functions have been created for great circle calculations:

great_circle_in *(Coord p1, Coord p2)* – calculates the incoming (i.e. at the final point) heading for a great circle path between two points

great_circle_out *(Coord p1, Coord p2)* – calculates the outgoing (i.e. at the initial point) heading for a great circle path between two points

great_circle_distance *(Coord p1, Coord p2)* – calculates the great circle distance between two points

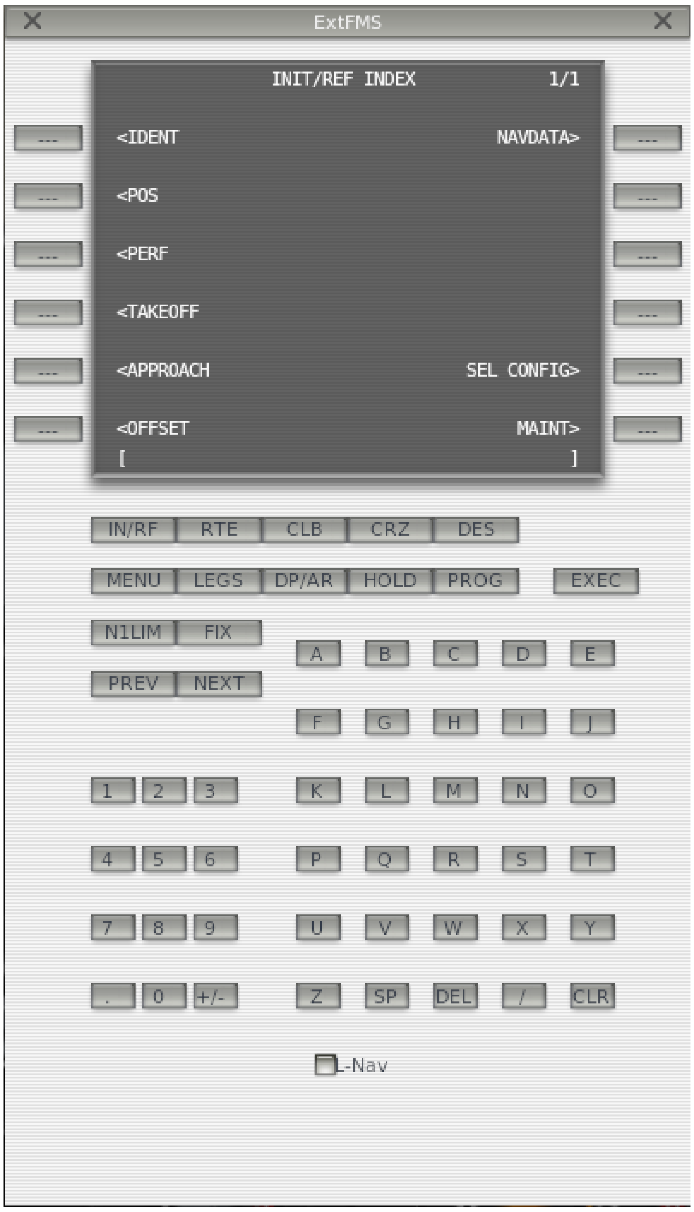
All calculations are done as described in chapter 2.4.1.

4. User Interface and operation

4.1. General User Interface layout

The UI of the plugin is loosely based on the Control Display Units featured in Boeing aircraft, particularly the 737 series. The UI can be divided into display portion – the screen – and the interface portion – the virtual keyboard (Figure 11).

The keyboard consists of 3 main sections: the alphanumeric keyboard used for entering the characters into the scratchpad, the page keys used to quickly invoke select pages of the FMC and the Line Selection Keys (LSK) used for context-based input. The screen is a monospace display fitting 14 lines, 45 characters each. These include the title line, 6 primary lines where data is displayed, 6 respective descriptor lines (displayed in blue above each primary lines) for text-based descriptions and the scratchpad line, displaying current contents of the scratchpad. Each primary line has two corresponding LSKs – left and right. While they are intended to invoke functionality based on screen context, the software permits completely arbitrary setup (see chapter 3.4.11).

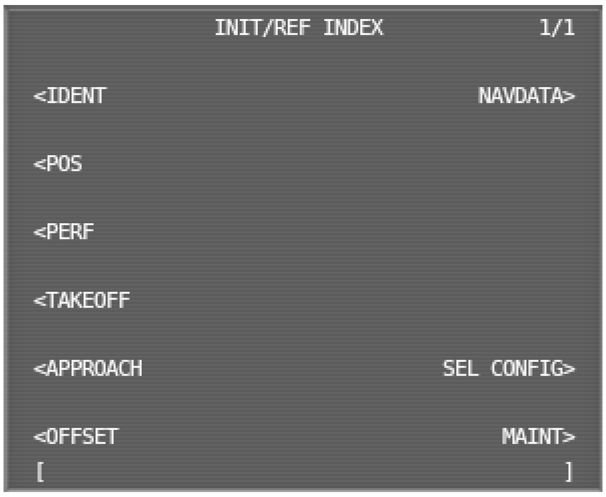


```
                 INIT/REF INDEX              1/1

<IDENT                                  NAVDATA>

<POS

<PERF

<TAKEOFF

<APPROACH                            SEL CONFIG>

<OFFSET                                   MAINT>
[                                              ]
```

Figure 11: UI layout

4.2. Information pages

4.2.1. INIT/REF INDEX

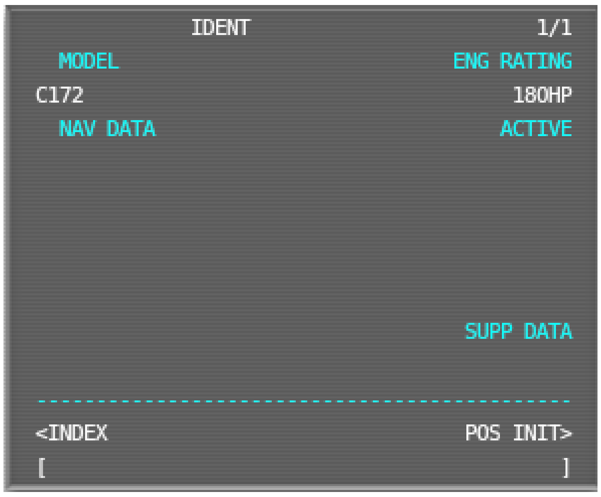


```
             INIT/REF INDEX          1/1

<IDENT                           NAVDATA>

<POS

<PERF

<TAKEOFF

<APPROACH                     SEL CONFIG>

<OFFSET                            MAINT>
[                                       ]
```

Figure 12: INIT/REF INDEX page

The Index page (Figure 12) is used to directly invoke pages used in flight preparation sequence and are not directly invokable with the page keys. While the page mimics the layout seen in 737NG, only the IDENT and POS INIT items are active here at this time.

4.2.2. IDENT

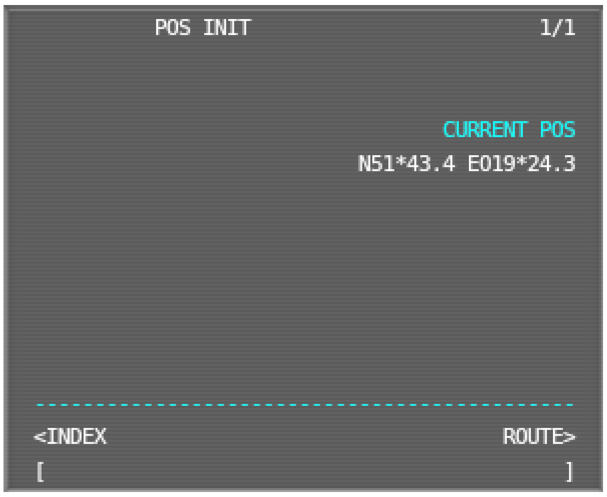


```
               IDENT                 1/1
  MODEL                       ENG RATING
C172                               180HP
  NAV DATA                        ACTIVE






                               SUPP DATA

----------------------------------------
<INDEX                         POS INIT>
[                                       ]
```

Figure 13: IDENT page

The IDENT page (Figure 13Figure 1) is accessed in an analogous manner to the 737 FMS i.e. through the INIT REF menu. The ident page summarizes some basic that such as the aircraft model, engine rating. The model and rating data are taken directly from datarefs, and they should display the ICAO code and the actual engine rating (thrust or power depending on propulsion type, nothing in case of gliders or basic models without proper engine simulation) for the currently selected aircraft.

4.2.3. POS INIT



```
          POS INIT              1/1

                         CURRENT POS
                  N51*43.4 E019*24.3





 ----------------------------------------
<INDEX                           ROUTE>
[                                     ]
```

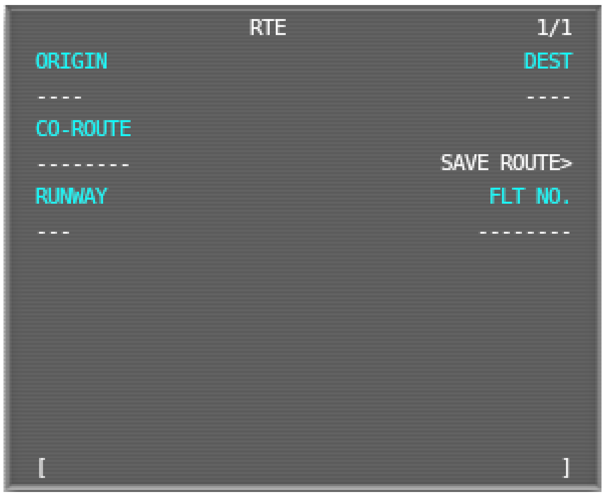
Figure 14: POS INIT page

The POS INIT page (Figure 14) in 737 features the necessary information and tools needed for initialization of the INS system. Those include direct readouts from aircraft GPS receivers as well as gate coordinates taken from the database. Since this information is aircraft-specific, the functionality of this page is, as of writing this thesis, limited to displaying the aircraft position taken from the simulation in LAT/LON format.

4.3. Flight planning

4.3.1. ROUTE

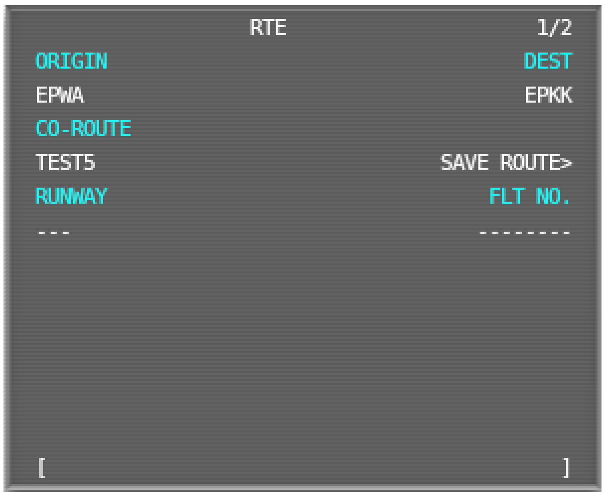
This is the core feature of this software. The Route page enables the user to enter the flight plan based on the information in the navigation database. The software accepts both direct paths between waypoints as well as airway following.



```
                    RTE                 1/1
ORIGIN                                 DEST
----                                   ----
CO-ROUTE
--------                        SAVE ROUTE>
RUNWAY                              FLT NO.
---                                --------




[                                         ]
```

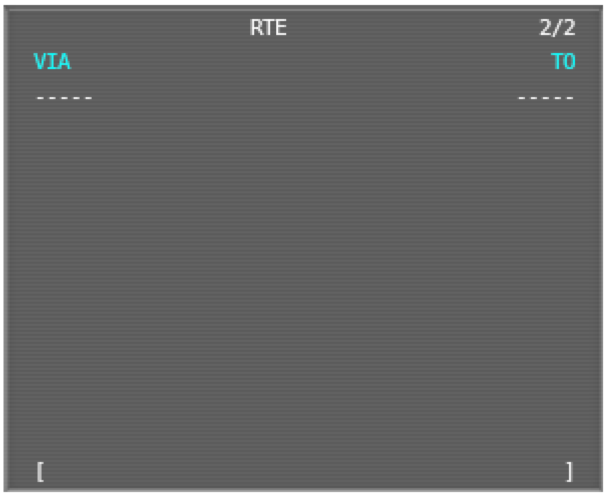
Figure 15: ROUTE subpage 1

The first page (Figure 15) allows the user to enter the origin and destination airports as well as load/save the route. Unlike the actual route, here the core SDK functions are used to verify entry (see 3.5.1), since no custom CIFP support is implemented at this time. The user can also load an existing flight plan file by entering the file name into scratchpad and selecting CO-ROUTE: this will load both the origin and destination, as well as the rest of the route. Upon entry (if the route was not loaded from a file), the origin and destination must be confirmed using EXEC button, enabling the user to enter page 2 (Figure 16).



```
                    RTE                 1/2
ORIGIN                                 DEST
EPWA                                   EPKK
CO-ROUTE
TEST5                           SAVE ROUTE>
RUNWAY                              FLT NO.
---                                --------




[                                         ]
```

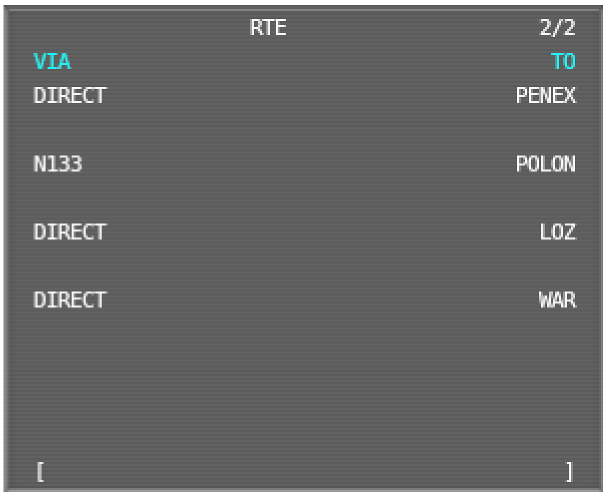
Figure 16: ROUTE subpage 1 with data entered and executed



```
               RTE              2/2
VIA                              TO
-----                         -----









[                                 ]
```

Figure 17: ROUTE subpage 2 and subsequent

The pages 2 and onwards (Figure 17) are used for flight plan entry. The entries are inserted with the line selection keys (left for airways and right for waypoints). If the waypoint is entered without specifying the airway, the "DIRECT" keyword is displayed in its place, indicating the system is going to plot a direct route to this waypoint with no regards for airways. Otherwise, the software will find a route via the selected airway. If the airway is entered after the waypoint has been selected, the system will change the plotted direct course into one that follows the selected airway. (Figure 18) If an airway is entered without specifying the waypoint, the system will display discontinuity (XXXXX in place of the required entry) if an attempt is made to activate it, until a waypoint to navigate to is specified.

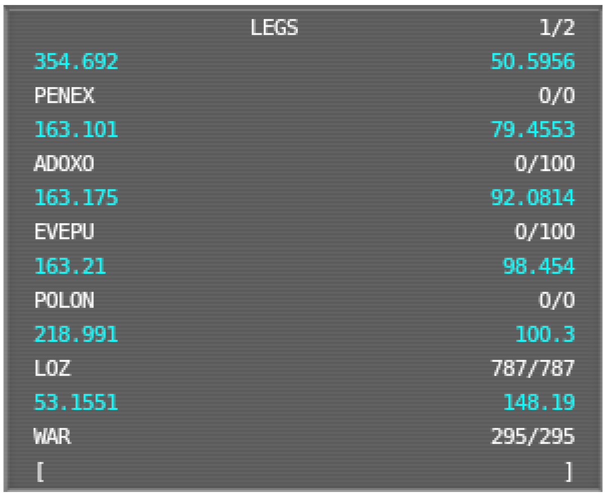


```
               RTE              2/2
VIA                              TO
DIRECT                        PENEX

N133                          POLON

DIRECT                          LOZ

DIRECT                          WAR



[                                 ]
```

Figure 18: ROUTE subpages with entered flight plan

Once the end of the page is reached, the system adds another page. There is no upper limit for the number of pages right now, but since the flight plan is stored in memory and operated on, excessive length may theoretically cause performance degradation, although it should be minimal on modern systems. Pressing EXEC at any moment will activate the current buffer flight plan, displaying the detailed path on LEGS page. This step is also required if the flight plan was loaded from a file. When finished, the flight plan can be saved into a file by returning to the first subpage, entering the desired file name into scratchpad and selecting SAVE ROUTE>.

4.3.2. LEGS



```
                  LEGS              1/2
354.692                         50.5956
PENEX                               0/0
163.101                         79.4553
ADOXO                             0/100
163.175                         92.0814
EVEPU                             0/100
163.21                           98.454
POLON                               0/0
218.991                           100.3
LOZ                             787/787
53.1551                          148.19
WAR                             295/295
[                                     ]
```

Figure 19: LEGS page

The LEGS page (Figure 19) displays all sections of the route, together with courses, distances and altitude limits. The default altitude restrictions/elevations are based on database, since no VNAV is simulated at this time. The distances and courses are calculated along the great circle. As subsequent waypoints are reached, the top waypoint is removed and the list cycles to the next item. Further information on the pathfinding and course calculation is given in the relevant section (chapter 2).

5. Conclusions

While the functionality of the resultant FMS simulation is limited compared to a full Flight Management System, a lot of work went into backend of the simulation and the base structure. Since the X Plane SDK is C-based, very low level and very limited in its access to resources, at least for the purpose of this project, most of the underlying functionality, including parsing of database files, had to be programmed from scratch. While the future versions of SDK might provide native functionality to access these features, the individual development of this functionality for the plugin permits further development without reliance on the SDK.

One major challenge when working with the X Plane SDK was the development of the user interface – the limitations of provided high level functionality required use of workarounds, while low level functions require comprehensive knowledge of OpenGL to draw graphics from scratch. This could be averted if the plugin was prepared for particular aircraft and the UI was prepared in a form of 3D cockpit elements scripted within the X Plane's graphics engine using lua.

Further development potential is limited unless plugin is adapted to power a selected aircraft, largely due to complexity of X Plane flight modelling. Since the flight model is dynamic, based on application of Blade Element Theory to the aircraft geometry, rather than a stability derivative based approach, the development of performance database would require either a very accurate flight model – in which case a real PDB can be used – or experimental generation of data in a manner similar to real FMS development but more automated owing to position and input manipulation abilities within the SDK.

5.1. Conformity with regulations

While there seem to be no flight simulation regulations specific to how software-based FMS implementations should perform, there are general requirements towards simulation of avionics in training devices, such as being able to accurately simulate the behaviour of the represented system, unaffected by occurrences specific to simulation environment, such as freezing flight or relocation of the aircraft, as well as a correct navaid environment [8]. If the functionality is extended to VNAV, in which case integration might be performed in calculations over discrete timesteps, additional steps might have to be taken to ensure that the data does not get corrupted as a result of such actions. As it stands, the calculations performed in flight loops are time independent (the turn radius and waypoint switch calculations, for instance, being based on distance and speed rather than time, partially in order to avoid this very issue), thus the stability of flight loop functions build with the X Plane SDK is ensured by the platform. The same applies to the navaid environment, as the database is provided by the platform.

There are however some regulations specific to real Flight Management Systems, that can proof useful in modelling of the behaviour. For example, the FAA Advisory Circular No. 25-15 [9] provided the requirements for airworthiness of FMS. The points that are of importance to this project are system integrity, accuracy criteria and North reference effects. The system integrity in this case is ensured by the fact, that the architecture of this software (external plugin) separates it from any other avionics simulation, save for limited interface that can be turned off. If this was a part of a full simulation of an aircraft, the integrity would have to be considered and fail safes similar to their real equivalents would have to be implemented. Similarly, while the core precision of navigation should be much higher than required, as calculations are performed on accurate data provided by the simulation, any simulation of noise and errors would have to be made in such a way as to ensure the Required Navigation Performance is appropriate despite the introduced noise. Finally, the North reference effect is taken care of by performing calculations on real data and only outputting magnetic headings when necessary by applying declination to the results, with declination data being provided by the platform and being identical to what is simulated within the environment.

5.2. Further development

While the intent for this project was to develop a generic FMS simulation that runs separately from core simulation, as a plugin, there is nothing preventing the computational core functionality from being adapted to work as a part of a full system simulation of a particular aircraft. Implementation in C++ provides certain performance advantages and greater possibilities than implementing the entire system as a lua script. However, significant work would have to be done on integration with other aircraft systems (which would often be defined through lua scripts) and the 3D cockpit, which is the current standard for X Plane 11 (which would typically be done through scripts, although there are plugins for XP11 that implement custom graphics that are projected within the 3D cockpit).

The interface itself is a potential subject to improvements as well. While the possibilities directly provided by the X Plane 11 SDK are limited, with some (but not very helpful to this particular project) being provided in recent SDK 3.0 release, the SDK allows direct drawing using OpenGL. This would however require either detailed knowledge of OpenGL (as everything would have to be programmed from scratch) or reliance on 3rd-party libraries to draw the UI.

Finally, a potential goal for future projects is to develop a vertical navigation engine. The main challenge in this case comes from the nature of the implementation of flight models in X-Plane. Instead of relying on stability derivatives and lookup tables, the flight characteristics are derived dynamically from general geometry and airfoil characteristics using an implementation of Blade Element Theory. While this allows to simulate subtle aerodynamic properties of the airframe that might not even have been tested on or documented, or even evaluation of potential performance of conceptual aircraft designs, it provides a challenge in this case, since no actual performance data exists anywhere in the software's assets. Even engine simulation is dynamic to a certain extent. Thus, while the performance calculations themselves would be easy to implement as the calculation methods are known, the capture of performance data to generate database from might even require a separate plugin to be written and could potentially provide enough challenge to be a subject of a thesis on its own.

Bibliography

1. **Brady, Chris.** *The Boeing 737 Technical Guide.* s.l. : Lulu, 2017. 9781447532736.

2. **Spitzer, Cary R.** *Avionics: Elements, Software and Functions.* Williamsburg, Virginia, U.S.A. : CRC Press, 2007. 0-8493-8438-9.

3. **Laminar Research.** XP-FIX1101. *X-Plane Developer.* [Online] 3 January 2017. [Cited: 27 February 2018.] http://developer.x-plane.com/wp-content/uploads/2016/10/XP-FIX1101-Spec.pdf.

4. —. XP-NAV1100. *X-Plane Developer.* [Online] 10 October 2016. [Cited: 27 February 2018.] http://developer.x-plane.com/wp-content/uploads/2016/10/XP-NAV1100-Spec.pdf.

5. —. XP-AWY1101. *X-Plane Developer.* [Online] 11 April 2017. [Cited: 27 February 2018.] http://developer.x-plane.com/wp-content/uploads/2016/10/XP-AWY1101-Spec.pdf.

6. —. XP-CIFP1101. *X-Plane Developer.* [Online] 5 February 2017. [Cited: 27 February 2018.] http://developer.x-plane.com/wp-content/uploads/2016/10/XP-CIFP1101-Spec.pdf.

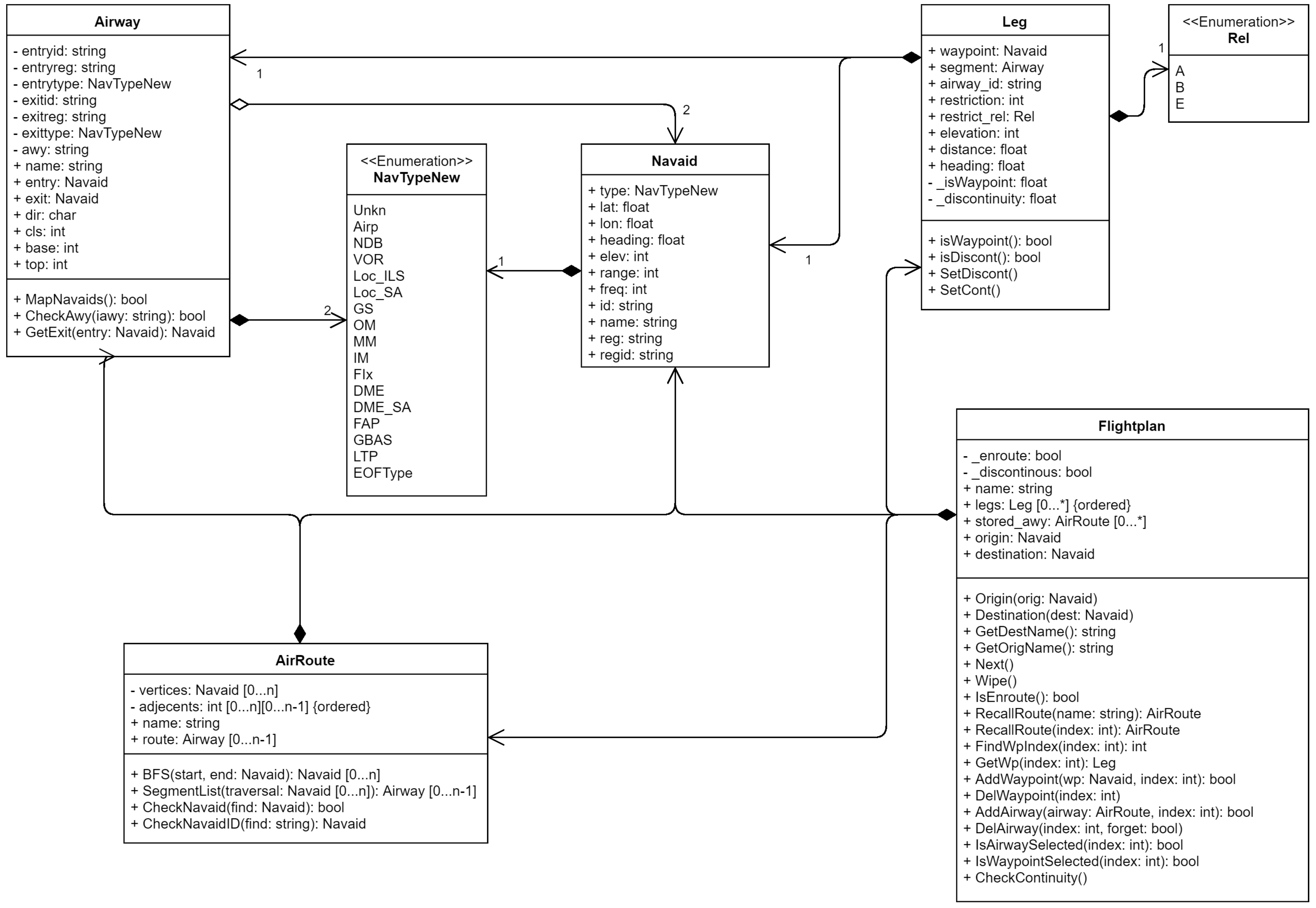
7. **Vincenty, Thaddeus.** Direct and Inverse Solutions of Geodesics on the Ellipsoid with Application of Nested Equations. *Survey Review.* April, 1975, Vol. XXIII, 176.

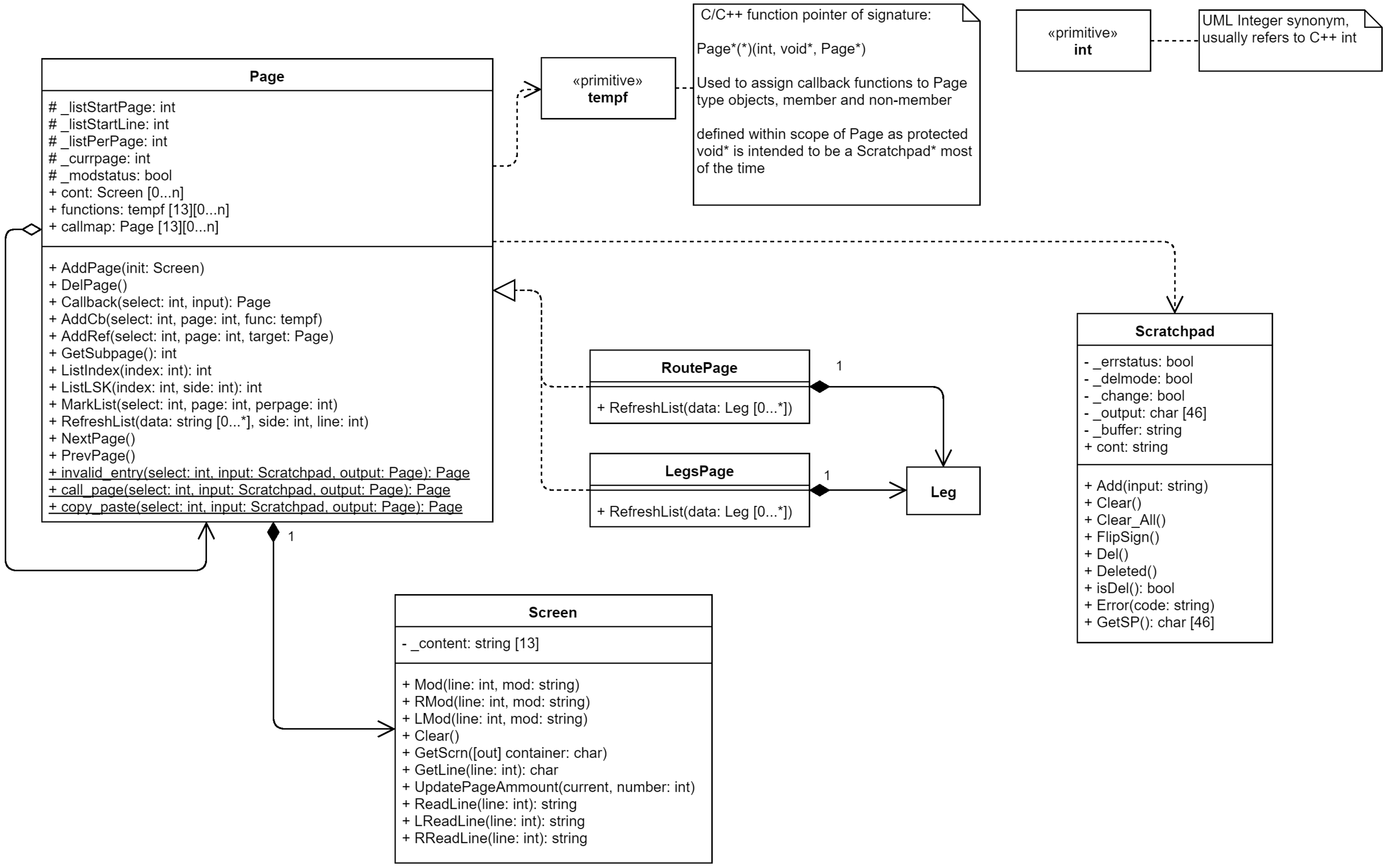
8. **European Aviation Safety Agency.** CS-FSTD(A). *Certification Specifications for Aeroplane Flight Simulation Training Devices.* 4 July 2012.

9. **US Department of Transportation, Federal Aviation Administration.** Advisory Circular No. 25-15. *Aproval of Flight Management Systems in Transport Category airplanes.* 1989.

Table of Figures

Appendix A Class relationship diagrams




Page
# _listStartPage: int
# _listStartLine: int
# _listPerPage: int
# _currpage: int
# _modstatus: bool
+ cont: Screen [0...n]
+ functions: tempf [13][0...n]
+ callmap: Page [13][0...n]
+ AddPage(init: Screen)
+ DelPage()
+ Callback(select: int, input): Page
+ AddCb(select: int, page: int, func: tempf)
+ AddRef(select: int, page: int, target: Page)
+ GetSubpage(): int
+ ListIndex(index: int): int
+ ListLSK(index: int, side: int): int
+ MarkList(select: int, page: int, perpage: int)
+ RefreshList(data: string [0...*], side: int, line: int)
+ NextPage()
+ PrevPage()
+ invalid_entry(select: int, input: Scratchpad, output: Page): Page
+ call_page(select: int, input: Scratchpad, output: Page): Page
+ copy_paste(select: int, input: Scratchpad, output: Page): Page
1
«primitive»
tempf
C/C++ function pointer of signature:
Page*(*)(int, void*, Page*)
Used to assign callback functions to Page type objects, member and non-member
defined within scope of Page as protected void* is intended to be a Scratchpad* most of the time
«primitive»
int
UML Integer synonym, usually refers to C++ int
RoutePage
+ RefreshList(data: Leg [0...*])
1
LegsPage
+ RefreshList(data: Leg [0...*])
1
Leg
Scratchpad
- _errstatus: bool
- _delmode: bool
- _change: bool
- _output: char [46]
- _buffer: string
+ cont: string
+ Add(input: string)
+ Clear()
+ Clear_All()
+ FlipSign()
+ Del()
+ Deleted()
+ isDel(): bool
+ Error(code: string)
+ GetSP(): char [46]
Screen
- _content: string [13]
+ Mod(line: int, mod: string)
+ RMod(line: int, mod: string)
+ LMod(line: int, mod: string)
+ Clear()
+ GetScrn([out] container: char)
+ GetLine(line: int): char
+ UpdatePageAmmount(current, number: int)
+ ReadLine(line: int): string
+ LReadLine(line: int): string
+ RReadLine(line: int): string