Politecnico di Milano

Scuola di Ignegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Aeronautica

DESIGN AND IMPLEMENTATION OF FLIGHT TEST DATA PROCESSING SOFTWARE

Supervisor:
Prof. Alberto Rolando

Tesi di laurea di:
Matteo Grassi,
matr. 800974

AA 2014-2015

"If you think it's simple, then you have misunderstood the problem."

Bjarne Stroustrup (C++ creator)

SOMMARIO

La presente tesi tratta lo sviluppo di un nuovo software di elaborazione dati per prove di volo.
Il personale del corso Prove di Volo si serviva inizialmente di una versione superata del software. Questa versione presentava numerosi problemi; inoltre, poiché gli ingegneri di Prove di Volo necessitano regolarmente di nuove funzioni e non essendo tale software sufficiente a soddisfare le loro richieste, esso risultò essere mancante di numerosi strumenti.
Partendo dalla vecchia versione e da uno studio del suo contesto di uso, l'idea di un nuovo software prese forma nella mente dello sviluppatore durante la fase di pianificazione. Nella fase di progettazione l'idea iniziò ad essere più chiara. Sono state progettati nuovi schemi e strutture del software in modo da avere un piano di lavoro durante la fase di sviluppo. La fase di sviluppo riguarda la reale scrittura del codice. Durante questa fase il programmatore ha dovuto risolvere tutti i problemi che sorgevano mano a mano e soddisfare le nuove richieste espresse dai committenti. Durante tutta questa fase sono stati eseguiti test in modo da offrire all'utilizzatore finale un software privo di errori.
Lo stadio finale ha riguardato la creazione dell'eseguibile da distribuire al pubblico.
Il software così sviluppato, oltre che soddisfare tutte le esigenze, presenta un'estrema flessibilità, un'interfaccia grafica particolarmente user-friendly ed è già predisposto per accogliere nuovi miglioramenti sviluppati da eventuali futuri programmatori.

ABSTRACT

The subject of this thesis is the development of a new flight test data processing software.
Initially Flight Test course staff and students used an outdated version. It presented many problems; moreover, being the Flight Test Engineers always needing new features and, being it not able to satisfy their requirements, it started to lack many tools.
Starting from the old version and from a study of the context of use, the idea of a new software took shape into the developer's mind during the planning phase. In the design phase this idea started to become clearer. New software schemes and structures have been designed in order to have a plan to follow during the development phase. The development phase is the actual writing of the code. During this phase the developer had to solve all the problems encountered and to meet all the new requirements coming from the stakeholders. A testing has also been carried out throughout this phase to provide a bug safe software to the end user. The final stage has been the actual deployment of the software in order to build the executable to be distributed to the public.
The software developed, apart from satisfying all the requirements, presents a great flexibility, a extremely user-friendly GUI and it is predisposed for new improvements by future developers.

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor Alberto Rolando for the useful comments, remarks and engagement throughout the learning process of this thesis. There is no doubt for me that without his precious guidelines, dealing with common difficulties for a beginner would be not possible and this dissertation would have not seen light.
Furthermore I wish to express my sincere thanks to Professor Giovanni Lozza, Principal of the School, to Professor Giovanni Azzone, Dean of Politecnico di Milano, and to the whole university for providing me, during those years, all the facilities necessary for my studies.
I would like to thank my loved ones, who have supported me throughout the entire process, by keeping me harmonious and guiding me.

Milano, april 2015 M.G.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

CAS	Calibrated Air Speed
DLL	Dynamic-link Library
EAS	Equivalent Air Speed
ECEF	Earth-centered Earth-fixed
ENU	East North Up
FAT	File Allocation Table
FFT	Fast Fourier Transform
FTE	Flight Test Engineer
FTI	Flight Test Instrumentation
GNU	GNU's Not Unix
GNU GPL	GNU General Public License
GNU LGPL	GNU Lesser General Public License
GPS	Global Positioning System
GUI	Graphical User Interface
IEEE	Institute of Electrical and Electronics Engineers
ISA	International Standard Atmosphere
MinGW32	Minimalist GNU for Windows
MSL	Mean Sea Level
MSVC	Microsoft Visual C++
OOP	Object Oriented Programming
OS	Operating System
QML	Qt Modelling Language
Qwt	Qt Widgets for Technical Applications
RDMS	Relational Database Management System
SQL	Structured Query Language
TAS	True Air Speed

UAV	Unmanned Aerial Vehicle
UML	Unified Modeling Language
UTC	Universal Coordinated Time
WAL	Write Ahead Log

1 INTRODUCTION

This thesis presents the development of a new software for data management requested by Polimi XFlight, a university flight testing group, for its own Flight Test Instrumentation (FTI) system. Together with the hardware and software of the acquisition instrumentation, this software will be used in many occasions, like lessons, experimental activities and research.
The process will be presented only through the most significant parts, comparing the job done to the previous version and showing its main virtues; this approach highlights the new release itself shadowing the developer commitment necessary to reach it.
The software development can be divided into different phases:

PLANNING This is the initial part, that comprehends the gathering of information about the software to be developed and from them obtaining some requirements to be satisfied.

DESIGN During this phase some critical choices will be developed and they will affect the whole project and its execution. Moreover a structure of the software and its parts will be developed in order to organize the subsequent phase and at the end to have an organized, neat source code; this will guarantee an efficient software and will allow future developers to easily expand the code.

DEVELOPMENT AND TESTING The development phase comprehends the actual building of the software by writing the source code; the concept, developed in the design phase, starts to take shape. Any change made to it in that moment is going to involve a great amount of work. For this reason, problems and new requirements have been high-lightened. Also the debugging is carried out in order to avoid the appearance of error to the end user.

DEPLOYMENT The building of the real executable to be distributed to end-users.

In the end a global, complete view of the application is given through its main features, for more information about the real structure of the software see the appendices.
Also a short forecast of the possible improvements soon to be implemented is given in the final part of the thesis.

2 PLANNING

This is the initial part, that comprehends the gathering of information about the software to be developed. A knowledge of the actual software condition and usage, of its main lacks and failure, provides the obtaining of some of the requirements. This task has been eased by my attendance at the Flight Testing course, so I already knew a part of the requirements.

2.1 INITIAL ANALYSIS

2.1.1 Context of use

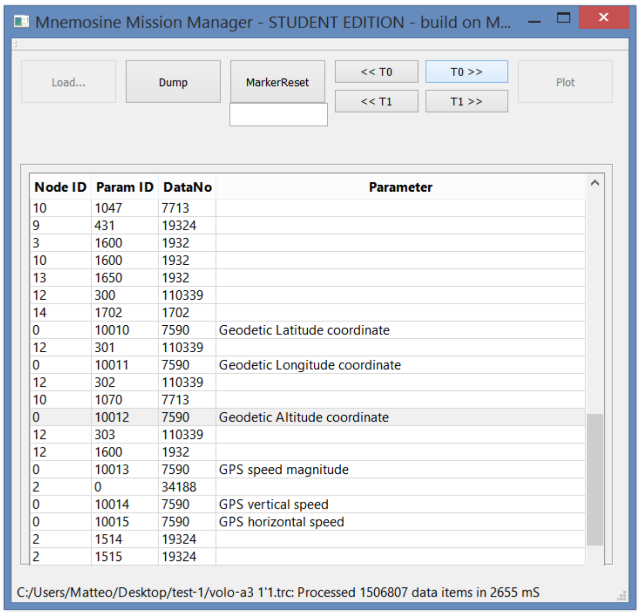
Mnemosine Mission Manager was created as a support to the FTI hardware system used to acquire flight data on ultralight aircraft. This system is used, as already said above, in the context of Flight Testing course at Politecnico di Milano in order to provide, in a usable format, flight data to course students; they, at a later stage, have to analyze them. Anyway those data may be useful for other purposes also to professors and stakeholders, in fact this system has been used to acquire data from new kind of ultralight aircraft and is going to be installed onto Unmanned Aerial Vehicle (UAV). Finally, there are expectations concerning a prospective use of the system by other universities.

2.1.2 Previous Release

The previous release of the software consists mainly of a table interface (fig. 1) to view data vectors and to plot them on a child window. It does not allow any kind of numerical operation on vector and there are no checks on the integrity of the imported data. The export is available only on comma separated value files (*.csv), one for each vector.

The plot window (fig. 2) contains only one plot with one curve at a time but many plot windows can be opened simultaneously; the user can move the plot but the movement is unlinked between the various windows and no zoom feature is available. There are two time markers located by default on initial and final time but they can be moved through the main window. Furthermore the x-axis displays nanoseconds and this is not very readable and useful.

In conclusion, even if the previous release had been able to import, export and show data, it was too basic to allow any kind of usage,



Node ID	Param ID	DataNo	Parameter
10	1047	7713	
9	431	19324	
3	1600	1932	
10	1600	1932	
13	1650	1932	
12	300	110339	
14	1702	1702	
0	10010	7590	Geodetic Latitude coordinate
12	301	110339	
0	10011	7590	Geodetic Longitude coordinate
12	302	110339	
10	1070	7713	
0	10012	7590	Geodetic Altitude coordinate
12	303	110339	
12	1600	1932	
0	10013	7590	GPS speed magnitude
2	0	34188	
0	10014	7590	GPS vertical speed
0	10015	7590	GPS horizontal speed
2	1514	19324	
2	1515	19324	

C:/Users/Matteo/Desktop/test-1/volo-a3 1'1.trc: Processed 1506807 data items in 2655 mS

Figure 1: Mainwindow of the previous software release

except for the conversion of the data to csv-file, which is nevertheless not versatile.

2.1.3 End-user analysis

Actually, the end-users of the software are students and professors from Politecnico di Milano, but the two categories make a different use of the software. Students require a fast, easy way to acquire flight data, viewing it quickly in the ground station and later exporting it in a file format compatible to the tools provided by the university; this means mainly MATLAB®. Even though professors share the same first two necessities, they do not require an export feature, but are more interested in saving the data into a long term ordered safe storage system to allow fast reliable data availability also in the future; moreover, professors need a way to calibrate data and to check the acquisition performance immediately after the flight.

2.1.4 Data analysis

The discussions that have been conducted till now relate to undefined data, but they are actually the core of all the activities related to flight testing. Therefore having a knowledge of the data type, dimensions and quantity acquired from the FTI is mandatory. Data are stored into a binary file (*.fti,*.trc) one structure at a time, even if the structure is different for each file format, the main characteristics are the same. Each structure contains (see figure 3):

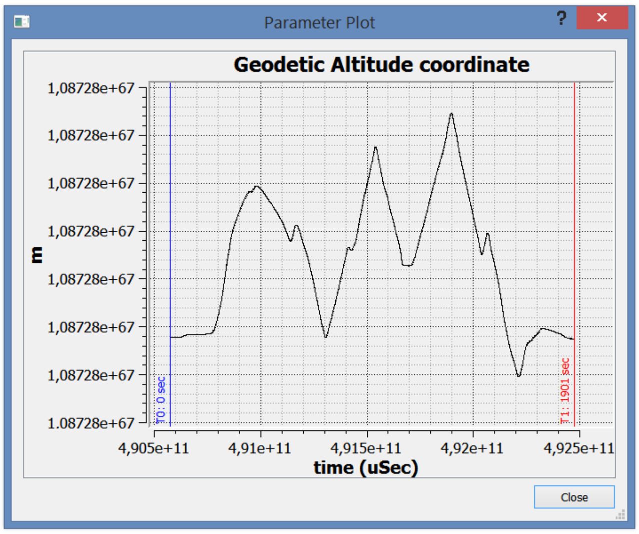


Figure 2: Plot window of the previous software release

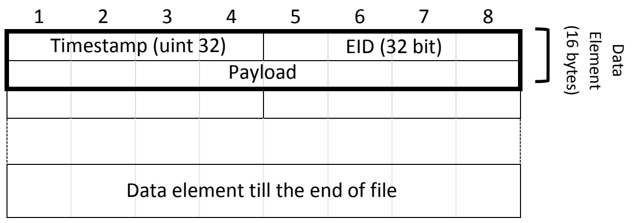


Figure 3: FTI file structure

TIMESTAMP A time datum in a local time reference system. Time is counted as a number of ticks from the starting of the FTI (a tick is a portion of time different for each acquisition system and file format).

EID 32 bit containing various information of which the most important are: an identification number that allows the classification of the datum into the correct vector (e.g. pitot boom static pressure) and the node ID that is used to divide the same kind of data into different vector depending on the acquisition system it came from (e.g voltage of a node).

DATATYPE The actual datatype used to correctly read the bytes of the payload (e.g. unsigned integer 32 bit, double).

PAYLOAD The actual datum.

When data are properly loaded from the binary file, they result into a group of vectors, each one containing a series of two dimensional points (time and datum). At the actual state, with flights lasting less than an hour and with some flight test instruments still missing, data files have a size between 20-40 MB, containing 50-60 vectors; vector size depends obviously on the data acquisition frequency, anyway it varies from around 1000 points to (in the worst case) 200,000 points.

2.1.5 Time Reference

A knowledge of the time formats to be used in the software is necessary to properly develop the software. The time formats are:

UTC Universal Coordinated Time can be stored as number of seconds (or smaller parts of it) from Jan 1 1970, like unix time, with the only difference given by leap seconds, a one second adjustment made occasionally to keep UTC aligned with mean solar time. It is usually shown as normal date and time(month-day-year hh:mm:ss).

LOCAL Local time is similar to UTC time format, the only difference is a variation of few hours accordingly to the time zone where the format is used.

GPS Global Positioning System time format [3] is composed by three different values: the week number (number of weeks from the start of the GPS, Jan 6 1980), the time of week (milliseconds from the start of the current week) and nanosecond reminder (number of nanoseconds, can be either positive or negative). The week number initially was 10 bit but, having reached the week 1024, its size has been increased to deal with it. GPS time does not have any connection with leap seconds and so there is no correction keeping it aligned with solar time.

IEEE The time precision protocol IEEE 1588 [5] contains rules to store time data in an accurate manner. The format prescribes an unsigned 32 bit integer for the number of seconds and a signed 32 bit integer for the number of nanoseconds. Epoch is not declared from rules so it can be chosen accordingly to peculiar needs.

2.2 REQUIREMENTS

Thanks to the previous analysis, I've been able to obtain requirements for the new version of the software which have been divided into mandatory or essential and optional.

2.2.1 Essential Requirements

IMPORT M^3 must be able to import every kind of file created by the actual acquisition systems (*.fti,*.trc).

EXPORT Comma separated value export tool is mandatory to allow student activities to be performed correctly.

STORAGE Software must provide a safe, reliable way to save the data for long term usage.

PLOT A plot feature is needed to have a fast way to view time-histories.

CALIBRATION There must be a way to calibrate the acquired data inside the application.

SANITY CHECK A sanity check is required by the Flight Test staff to be able to control the acquisition results immediately after the flight.

2.2.2 Optional Requirements

EXPORT Would be useful to students to be able to export into MAT-file format in order to make the usage of data in MATLAB® easier.

PLOT Having the plot not in the external window would make the software more appealing and less disorganized.

PLOT Allowing more curves on the same plot would permit easier and faster comparisons.

PLOT Synchronized time-axis (x-axis) movement between all plot could provide a way to make the user more easily aware of the situation of the flight at an exact segment of time.

PLOT Time axis showing current time would be more user friendly.

ZOOM A zoom feature would make the software more valuable and it would allow a wider usage.

ZOOM A synchronized zoom between curves would be an added value to the software.

MARKER A way to mark TOPs and moments of interest on all plot would be required but not mandatory, because its absence does not degrade software functionality.

PLATFORM Being able to use the software on different platforms (e.g Windows, Ubuntu, Android) would guarantee a greater flexibility to all users and quicker access to data also from mobile devices (e.g. smartphone, tablet) which are very common nowadays.

FREE If possible, the tools used during the development must be free of charge to avoid the purchase of them for each developer.

FREE The application will be probably sold. The usage of the software and the libraries distributed with free licenses that allow commercial third party use is recommended.

3 DESIGN

The design phase of the software comprehends all the initial decisions and planning made before the actual building of the application by writing the source code. The design phase took quite some time in order to allow me to acquire the needed knowledge regarding software design and programming language, but also to allow an initial, well developed scheme of the actual features in order to simplify the subsequent building phase and make it more neat. This will improve the source code, making it easier for future developers to expand and improve this software.

3.1 CRITICAL CHOICES

3.1.1 Programming Language

The decision regarding the programming language was taken quickly because of various reasons. First we needed to analyze the programming paradigms which are divided into 3 macro categories: procedural programming, structured programming and object oriented programming.

PROCEDURAL PROGRAMMING This programming paradigm was widely used in very old software. Procedures are executed in series and sometimes are groped into subroutines and functions. It is therefore possible to avoid some of them by using GOTO command. This kind of language is built with a top-down approach, it is easy and fast to construct simple applications, but with large applications and a complex code with many connections between the functions it is not the best choice.
Examples of procedural language are: Basic, Fortran.

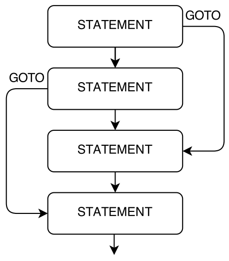


Figure 4: Procedural Programming, example scheme

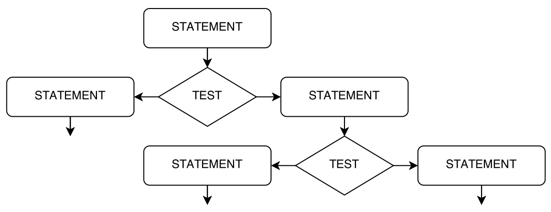


Figure 5: Structured Programming, example scheme

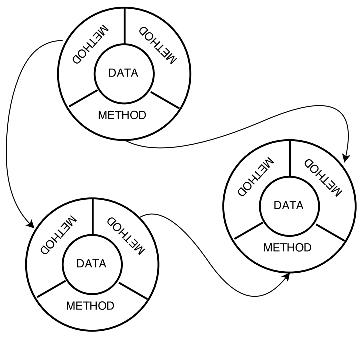


Figure 6: Object Oriented Programming, example scheme

STRUCTURED PROGRAMMING The structured style is an improvement of the procedural paradigm, because it allows a more flexible usage of functions thanks to block structures and a greater use of logical operators.
Examples of structured language: C, Pascal.

OBJECT ORIENTED PROGRAMMING The Object Oriented Programming (OOP) techniques are actually the most widely adopted and they are a powerful programming tool in the current software and hardware environment. They are based on the usage of object or class that contains both data and methods which are functions that allow access to data. This programming style is not top-down but bottom-up because, in order to build a software in this way, a developer must build classes first. OOP based applications are very flexible and reusable. They can be also very complex. Furthermore OOP provides a safer way to use data, while being usually accessible only through methods.
Examples of OOP language are: C++,java.

In the end the final choice regarding the paradigm itself ended up onto the OOP style due to the obvious advantages given by it, a simple, flexible software structure open to possible future developments and a safe data processing. The programming language has been easily guided, once chosen the paradigm, C++ [2] is one of the most common programming languages, this powerful language has been used to create operative system and a lot of software. Moreover the previous software was developed in a C++ environment allowing the usage of parts of its code.

3.1.2 Development Environment

The selection of the development environment has been Qt [6], a cross-platform C++, QML and javascript integrated development environment. This framework provides a large library of fully developed C++ classes. Those objects take care of almost every basic need of C++ developers, allowing them to focus only on the case specific part of the application. There are classes for input/output device management, file management, data storage and usage, database operation, almost every basic Graphical User Interface (GUI) element and many others. Also Qt Creator has also a GNU Lesser General Public License (GNU LGPL); this means we develop software in a totally free manner.
Qt Creator can use many different compilers. It is provided with a basic compiler for 32 bit x86 processor in Windows environment, Minimalist GNU for Windows (MinGW32). It is free and open source but in order to improve our software for Windows usage (it is still the most common Operating System (OS) used in our university), we decided to use Microsoft Visual C++ (MSVC) which has also a free redistributable version. MSVC usage will make our software already well optimized for Microsoft OS without having to do it manually.
Anyway, in case it is needed to have a software version working on other OS, Qt allows the developer to add other compilers and build the project with each of them; so choosing Qt as development environment can satisfy the requirement asking for a multi-platform application.
Even though Qt has many useful features, it lacks a fully developed pack of GUI components to plot graphs. Luckily there has been already built such a library, compatible with Qt environment, Qt Widgets for Technical Applications (Qwt) library [7]. Qwt has a GNU LGPL allowing free usage.

3.1.3 Database Environment

The need of a data storage system has been already underlined in the previous chapter. This system obviously must provide fast access both in upload and download to the needed part of data stored. This could not be achieved with just a big file or small files because such a system would lack of flexibility. A database is a reliable and flexible solution to such a problem. The database allows the storage of data into an organized manner permitting fast retrieval of every data recorded into it; moreover, if a relational database is used, the software will be able to achieve greater flexibility in association with it.
Initially the choice of the Relational Database Management System (RDMS) was going toward MySQL which is licensed as GNU General Public License (GNU GPL); it is fast, reliable and would have allowed

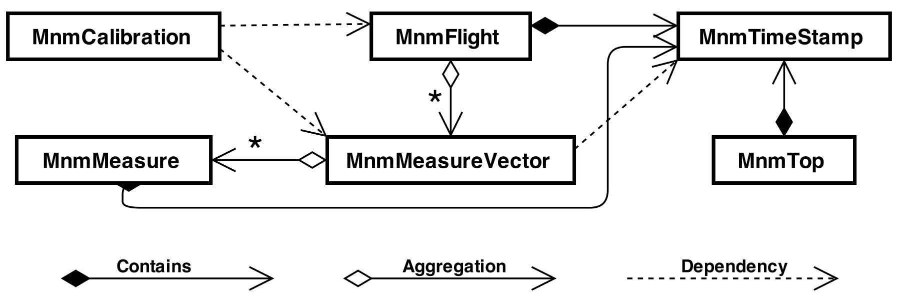


Figure 7: C++ class plan, UML model, dependency map

the creation of a dedicated database on the Politecnico di Milano server to guarantee access to data to everyone in the university who needed them. Obviously to connect the software to such a server an internet connection must be available. This is not always granted, especially in Flight Testing because the test aircraft could eventually fly in rural area without a stable internet connection or with a slow one; to avoid this problem we fell back on a portable solution. SQLite [8] is a self-contained, serverless, zero-configuration, transactional SQL database engine, once a SQLite database is created, it does not even require the software but just the database file itself. Anyway, SQLite has its own weaknesses; in fact it does not protect database integrity as values are weakly typed allowing the storage of a different datatype in a column. Even if MySQL is used in many famous applications and large websites like Facebook and Twitter, also SQLite is used in common software like web browsers and smartphone OSs, as a guarantee of its usefulness and safety.

3.2 C++ CLASS PLAN

The decision of the usage of an object oriented language like C++ determined the necessity to identify the needed classes and their own structures. The first ones that needed to be identified, designed and created are obviously classes related to data management, they are the subject of this section. For information regarding the GUI see sec. 3.4. In this section the initial plan for the C++ data classes is presented, for details on the actual C++ classes see appendix A. To have a rough idea of class plan, you must look at fig. 7 and 8; a short description is given below of the main characteristics of each class.

3.2.1 MnmTimeStamp

This class is the container of the time datum. It must be able to work with all the different time formats described in the section 2.1.5.

UTC Universal Coordinated Time can be stored using the existent QDateTime class. It provides a very good group of functions that can fulfil any kind of date and time requested. Unfortunately QDateTime does not take into account leap second. So, every time Universal Coordinated Time (UTC) time format is requested, a correction is applied to raw data, using an *.ini file containing all the leap seconds adjustments.

Y2015 = 16
M2015 = 0

These two lines are taken from the file, they show how leap second information is stored: a value of the actual leap seconds from 1980 stored as Y followed by the actual year, and an identifier to locate when the leap second has been exactly placed (it can be put at midnight of the 30 June or 31 December, in the file it is actually indicated on the next moth, so, respectively, July or January of the next year).

GPS In order to deal with Global Positioning System (GPS) time format [3] a new structure had to be built. QDateTime was not able to manage GPS time format.

Datatype	Identifier	Description
uint 16 bit	m_WeekNum	Week number: current number of weeks since GPS epoch (Jan 6 1980)
uint 32 bit	m_ToW	Time of week: number of milliseconds from the start of the week
int 16 bit	m_nSecRem	Nanoseconds reminder: number of ns (signed)

Table 1: GPS time data structure

The datatype of each value has been chosen accordingly to its boundaries and the week number it is not limited by the 1024 old limit. As said before the GPS time does not have a dependency from leap second, so no correction is needed.

IEEE A proper structure has been built to manage IEEE 1558 [5] time format. For details see table 2.

Datatype	Identifier	Description
uint 32 bit	m_seconds	Number of seconds from the epoch (it can be chosen accordingly to needs)
uint 32 bit	m_nSec	Nanoseconds from the current second

Table 2: IEEE 1558 time data structure

NS Nanosecond number from a variable epoch is often used to elaborate data.

Even if all those formats are required, only one of them can be effectively stored in the class in order to avoid an unnecessary memory usage. At the early start IEEE 1588 data format was chosen, but after few weeks, still during the planning, it has been changed to GPS time format to avoid too many numerical operations on data, as GPS time is almost the actual output given by the FTI (FTI return a timestamp that can be related directly to GPS time also recorded by the system). Regarding the functions of this class, they must be able to work from and to any kind of these formats, for export and visual output needs.

3.2.2 MnmMeasure

This object corresponds to the two dimensional point of a time history. It contains a time value (MnmTimeStamp) and one instrumentation output at that time. The measured value is stored like a QVariant, this is a Qt class that allows to manage many kind of different datatype without using the real format for every value. It must be able to set these values and give them back in case of need.

3.2.3 MnmMeasureVector

Each time-history corresponds, regarding data, to one MnmMeasureVector object. In fact it contains a QList composed of pointer to MnmMeasure objects, or a vector of points. Both QList and QVector could have been used but QVector stores data in the memory in a more rigid way (it stores data in adjacent bytes, QList stores just a pointers gaining also a lot of heap memory) slowing down access to data inside it. In this class, apart from the list of measure, also basic info are stored: label, measure unit, description and the number of nanosecond for one FTI tick.

3.2.4 MnmFlight

This class is the container for all the time-histories of a flight; pointers to all the vectors are stored into a Qhash with a key. The key is a quint32 value obtained from node ID and paramater ID of the vector.

```
quint8  NodeID = 2;
quint16 ParameterID = 321;
quint32 Hash_key = (NodeID  << 16) + ParameterID;
```

The epoch is also stored there to avoid too much memory usage by placing it into other classes. About the methods, this object needs functions to be able to guarantee access to vectors and also a function to provide fast appending of new points to the hash table.

3.2.5 MnmCalibration

One of the requirements was the ability to elaborate data. This class is at the base of this need. It is a bridge between raw data vectors and calibrated data vectors; vectors with parameter ID lower than 32768 are made of raw data, the other contains calibrated data. Apart from the obvious data (i.e. parent node ID, parent parameter ID, label, description, measure unit), this class contains also two boundaries, they are the way to fulfil also sanity check requirements; having no way to put reasonable boundaries on raw data (data can be acquired as voltages and voltage settings can vary every time), the problem has been solved by placing boundaries on calibrated data. When vectors go through the calibration, the software analyses if there are values outside of the prescribed boundaries of each vector and shows the result, providing a way to the user to know almost instantly if something has gone wrong during acquisition phase. About the curve, a new structure contains its information.

As it is possible to see in table 3, the structure contains 4 coefficients

Datatype	Identifier	Description
enum	t_curveClass	Curve class identifier: it define how to use the coefficients in the correct way
double	a0	First coefficient
double	a1	Second coefficient
double	a2	Third coefficient
double	a3	Last coefficient

Table 3: Calibration curve data structure

which are used in the proper way, depending on curve class enumerator value:

NONE $y = x$

LINEAR $y = a_0 + a_1\ x$

POLY $y = a_0 + a_1\ x + a_2\ x^2 + a_3\ x^3$

POW $y = a_0\ x^{a_1}$

EXP $y = a_0\ \exp\left(x^{a_1}\right)$

LOG $y = a_0\ \ln\left(x\right) + a_1$

Other than usual functions, as mentioned above, it must be able to perform a sanity check with boundaries and to compute the calibrated vector.

3.2.6 MnmTOP

The TOP Object it's simply made of a MnmTimeStamp object of a moment of interest, a label and info to identify that time instant and a TOP class identifier to know where the TOP has been acquired (i.e. in flight, on ground, post processing, unidentified). Functions must provide the usual access to the inside data.

3.3 DATABASE PLAN

The database structure can be easily understood by looking at fig. 9; the Unified Modeling Language (UML) diagram is self explaining. So, only main features and parts that require special attention will be described below.

The most important table of the database is the flight table. It correlates all the other tables; the flight table ID is the only one given by the software as an unsigned 64 bit integer and not composed of an automatically incremental integer. This is to avoid the insertion of the same flight in two different record of the table. The flight ID, from left to right contains:

- Flight Test Engineer (FTE) ID (up to 10000)
- pilot ID (up to 100)
- year of flight execution
- month of flight execution
- day of flight execution
- hour of start of the flight
- minute of start of the flight

This table contains 4 values from 4 different tables but only two of them are foreign key, pilot and FTE IDs. Foreign keys are a way that database itself use to protect its own integrity, but in order to do so, any INSERT or REPLACE query missing a foreign key is rejected; pilot and FTE assignment to a flight is mandatory in order to have a correct flight ID. On the other hand aircraft and calibration group may not be set when the flight is saved. So to avoid query failure they were not added as foreign key but just as normal integers. Another characteristic planned solution is the use of a different table for each kind of data type of the measured values (data type will be stored in a VECTOR table row, in order to know which table points are stored into). In each row of POINT tables there is also a column called serial number, this is a value assigned during data inserting procedure in order to guarantee the correct order of the points also when they are loaded into the software. One other feature requires attention: the calibration are not stored as vectors but as calibrations, only vectors with parameter ID below 32768 are actually saved into the database; remaining vectors are computed from calibration data when flight is loaded.

3.4 GRAPHICAL USER INTERFACE

In order to have a rough idea of the graphical user interface of the software, an initial mock-up has been done. The mock-up, even if it gives only an approximate view of the application (the GUI design can be easily changed even in the final phase of construction), will be helpful while building all the widgets composing the GUI in order to keep in mind the connections between them and their possible arrangement. The GUI presented here have many differences from the final one (Appendix B).

The main window (fig. 10) plan tries to fix some lacks of the previous release, especially the need of new windows for each plot, by placing them into tabs inside the main window; the first tab, instead, will contain the vectors table.

The GUI also have a menubar with drop down menus (fig. 11), an innovation from the previous release in order to avoid unnecessary space use with buttons (e.g. import, load).

Finally the plot tab (fig. 12) is improved by the possibility of placing more than one plot in it and with a small menu on the right for plot settings.

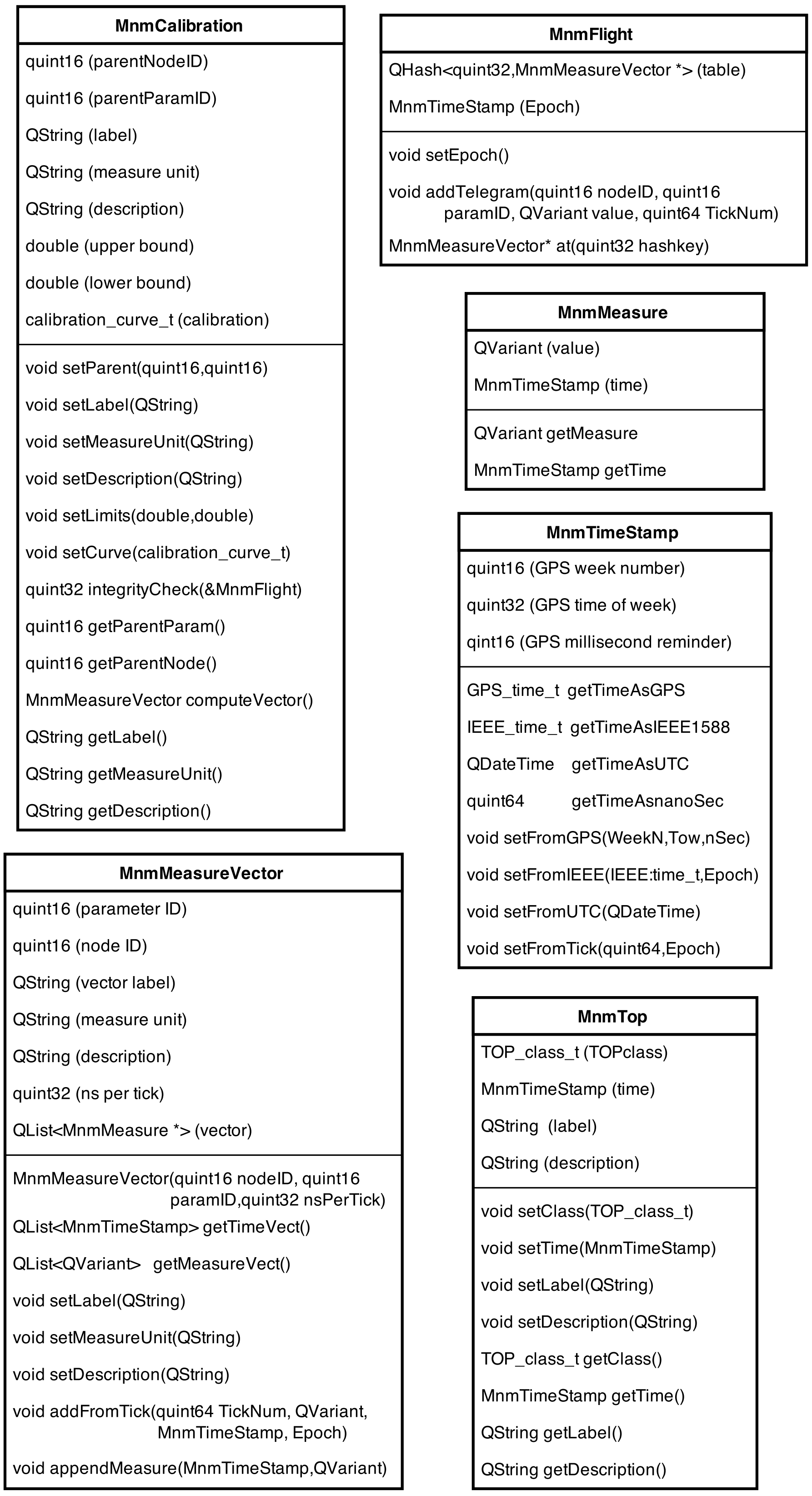

MnmCalibration
quint16 (parentNodeID)
quint16 (parentParamID)
QString (label)
QString (measure unit)
QString (description)
double (upper bound)
double (lower bound)
calibration_curve_t (calibration)
void setParent(quint16,quint16)
void setLabel(QString)
void setMeasureUnit(QString)
void setDescription(QString)
void setLimits(double,double)
void setCurve(calibration_curve_t)
quint32 integrityCheck(&MnmFlight)
quint16 getParentParam()
quint16 getParentNode()
MnmMeasureVector computeVector()
QString getLabel()
QString getMeasureUnit()
QString getDescription()
MnmFlight
QHash<quint32,MnmMeasureVector *> (table)
MnmTimeStamp (Epoch)
void setEpoch()
void addTelegram(quint16 nodeID, quint16 paramID, QVariant value, quint64 TickNum)
MnmMeasureVector* at(quint32 hashkey)
MnmMeasure
QVariant (value)
MnmTimeStamp (time)
QVariant getMeasure
MnmTimeStamp getTime
MnmTimeStamp
quint16 (GPS week number)
quint32 (GPS time of week)
qint16 (GPS millisecond reminder)
GPS_time_t getTimeAsGPS
IEEE_time_t getTimeAsIEEE1588
QDateTime getTimeAsUTC
quint64 getTimeAsnanoSec
void setFromGPS(WeekN,Tow,nSec)
void setFromIEEE(IEEE:time_t,Epoch)
void setFromUTC(QDateTime)
void setFromTick(quint64,Epoch)
MnmMeasureVector
quint16 (parameter ID)
quint16 (node ID)
QString (vector label)
QString (measure unit)
QString (description)
quint32 (ns per tick)
QList<MnmMeasure *> (vector)
MnmMeasureVector(quint16 nodeID, quint16 paramID,quint32 nsPerTick)
QList<MnmTimeStamp> getTimeVect()
QList<QVariant> getMeasureVect()
void setLabel(QString)
void setMeasureUnit(QString)
void setDescription(QString)
void addFromTick(quint64 TickNum, QVariant, MnmTimeStamp, Epoch)
void appendMeasure(MnmTimeStamp,QVariant)
MnmTop
TOP_class_t (TOPclass)
MnmTimeStamp (time)
QString (label)
QString (description)
void setClass(TOP_class_t)
void setTime(MnmTimeStamp)
void setLabel(QString)
void setDescription(QString)
TOP_class_t getClass()
MnmTimeStamp getTime()
QString getLabel()
QString getDescription()

Figure 8: C++ class plan, UML model, data and methods (name are not the real ones to be self explaining as much as possible)

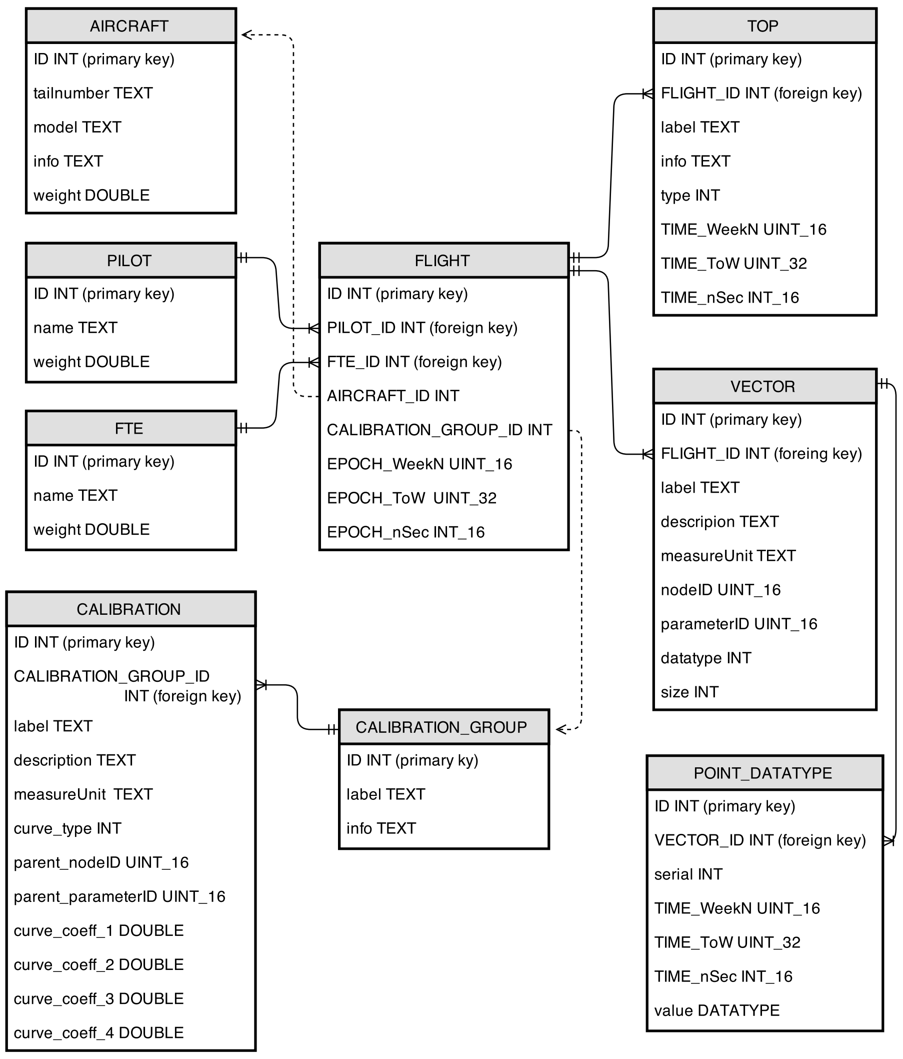
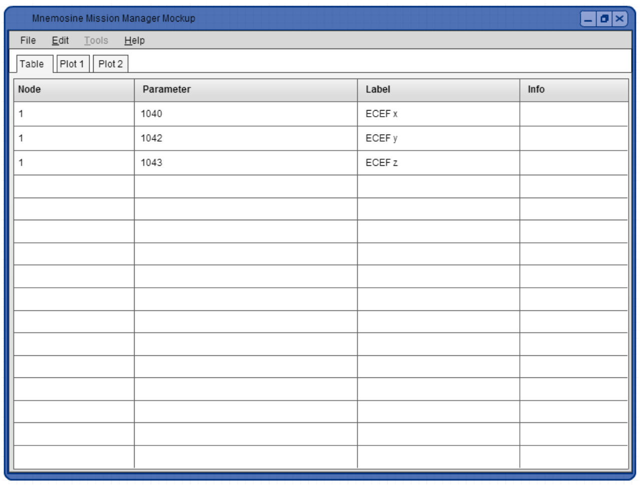


Figure 9: SQL tables plan, UML model (name are not the real ones to be self explaining as much as possible)



Node	Parameter	Label	Info
1	1040	ECEF x	
1	1042	ECEF y	
1	1043	ECEF z	

Figure 10: Mock-up of the main window

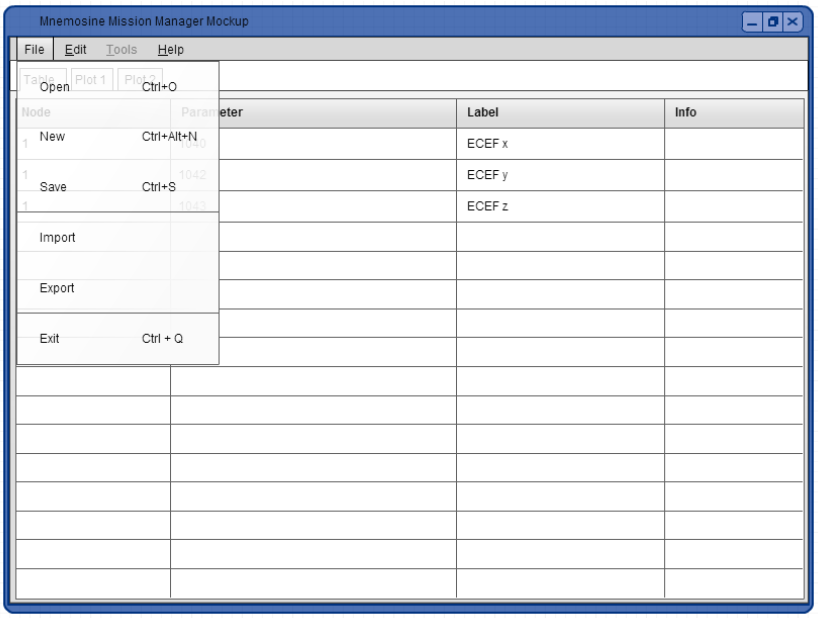


Figure 11: Mock-up of the main window, drop down menu

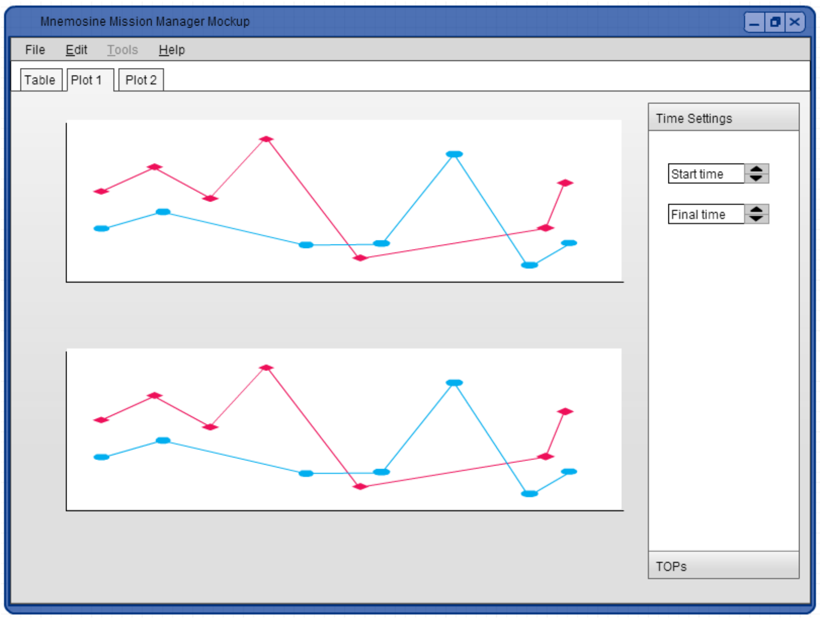


Figure 12: Mock-up of the main window, plot tab

4 DEVELOPMENT AND TESTING

The software development is a time consuming activity; all the features planned must be realized and, obviously, nothing goes exactly as planned. Problems arise during the building up and sometimes they require a redesign of some parts of the software. Moreover, the stakeholder may come up with new requirements for the software.
Regarding the testing/debugging, no particulars problems have been encountered, although this activity has been executed simultaneously with the development and a great amount of time has been taken by the resolution of many small bugs found in the software; half of the bugs were caused by imperfect use of C++ and Qt libraries due to inexperience and they were easy to solve. The other half, instead, requested an in-depth research to find a solution.

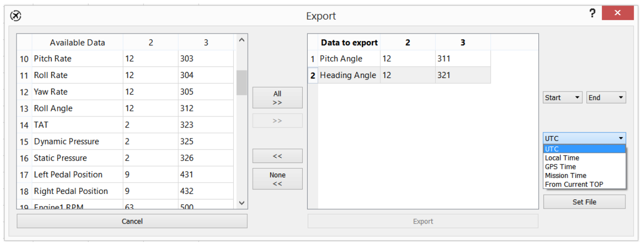
4.1 IMPROVED FEATURES, MAIN PROBLEMS AND LATE REQUIREMENTS

4.1.1 Inappropriate use management

The software can perform a lot of actions if used correctly; unluckily a wrong use may conduct to errors and software shut-down. Part of the time has been used to make inappropriate use impossible to the end-user, mostly by enabling and disabling actions and buttons when some conditions arise.
One of the most important protections disables pilot and FTE setting after their first selection; this ensures the correct recognition of the flight by the database and it defends the integrity of the latter.
Other corrections were applied: to time management to avoid the unnatural settings of the x-axis range, to file import tool to avoid loss of the current data management session and to check the import to be performed properly, to save-on-database tool to avoid integrity loss and unwanted overwriting of data, etc.

4.1.2 Data export improvements and MAT-file

The data export feature has been greatly improved from the previous release. All the lacks identified have been corrected as much as possible. The new export widget allows to export all or just some of the vectors from the flight by selecting them: the two tables allow a fast



	Available Data	2	3
10	Pitch Rate	12	303
11	Roll Rate	12	304
12	Yaw Rate	12	305
13	Roll Angle	12	312
14	TAT	2	323
15	Dynamic Pressure	2	325
16	Static Pressure	2	326
17	Left Pedal Position	9	431
18	Right Pedal Position	9	432
19	Engine1 RPM	63	500

	Data to export	2	3
1	Pitch Angle	12	311
2	Heading Angle	12	321

Figure 13: Export window

visual recognition of the vectors being exported (they can be moved from the left to the right table to export them or they can be removed in the opposite manner). Moreover, it is now possible to select only a short timepiece to export, between two existent TOPs; finally the time reference system can be selected between many possibilities:

- UTC reference system (nanoseconds from Jan 1 1970)
- GPS reference system (nanoseconds from Jan 6 1980)
- Mission Time reference system (nanoseconds from mission epoch)
- From Current TOP reference system (nanoseconds from selected TOP)

To further improve export feature and to simplify student's life, as suggested in the requirements, the MAT-file format [10] has been added to export formats; this was not an easy task, even if the MAT-file format adopted was level 5, the simplest one, because MATLAB® requires an exact structure.
First of all it is needed to know that MATLAB® reads the files 64 bit at a time. That is why the structure is presented in 8 bytes columns in fig. 14. This characteristic will have also some consequences on the following details. The 116 bytes of descriptive text at the top of the file contains few information on the file and on the flight data contained (example: MATLAB 5.0 MAT-file, Platform: Mnemosine, Created on: mer mar 11 02:00:10 2015, Flight on: ven mag 23 13:11:48 2014); all the following bytes are filled with null characters. Finally there are the version number, equal to 1 (from MathWorks documentation), and the endian indicator (IM indicating little endian encoding).
After the header, each data element is placed one after the other until the end of file is reached; also the data element have a rigid prefixed structure shown in fig. 15. The sub-element following the data element tag are in our case:

- the array flags (16 bytes: 8 tag, 8 data) to indicate the variable type (i.e. complex, global, logical) and the data class (e.g. double array, uint_16 array).
- dimensions element (16 bytes: 8 tag, 4 rows number, 4 column number) to indicate data element dimensions.

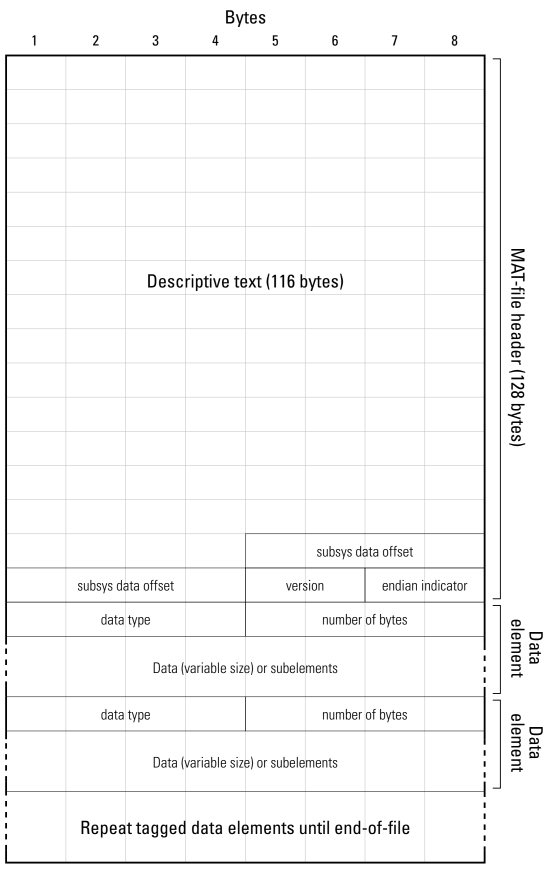

Bytes
1
2
3
4
5
6
7
8
Descriptive text (116 bytes)
subsys data offset
subsys data offset
version
endian indicator
MAT-file header (128 bytes)
data type
number of bytes
Data (variable size) or subelements
Data element
data type
number of bytes
Data (variable size) or subelements
Data element
Repeat tagged data elements until end-of-file

Figure 14: MAT-file level 5 structure

- name element containing the data element name that will appear in MATLAB® workspace (mnm(node ID)_(parameter ID)_(vector label)).
- the real data element, 8 bytes of element tag (containing the number of bytes of real data) and the data.

Attention must be paid to every sub-element length, because in case it does not occupy a multiple of 8 bytes, padding bytes must be put to restore it.

4.1.3 New import data file format: FTB

Approaching the end of the project, a new requirement has arisen from the FTI building division; a new FTI output file format will be adopted in this year flight test campaign.
Mnemosine Mission Manager must be able to import this file to be useful during the soon to come campaign; the file structure is a lot different from the older format. In fact a flight is not saved into a single file but into many 128 KB files, split between many folders each containing 128 files. This new method of saving improve data safety, because an accidental, sudden removal of a FAT formatted volume [9]

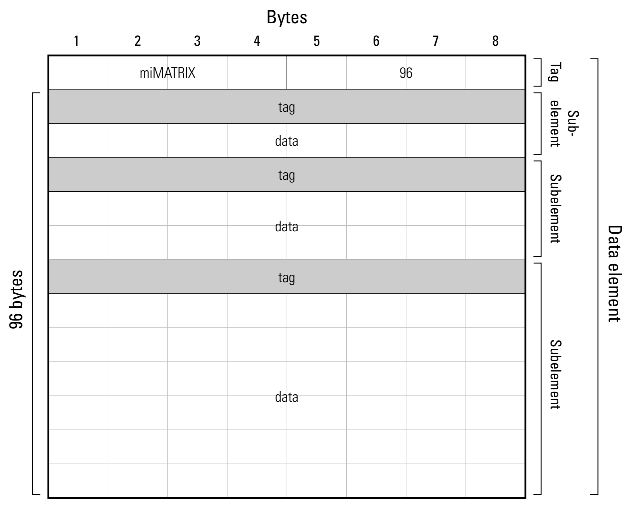


Figure 15: MAT-file level 5, data element structure

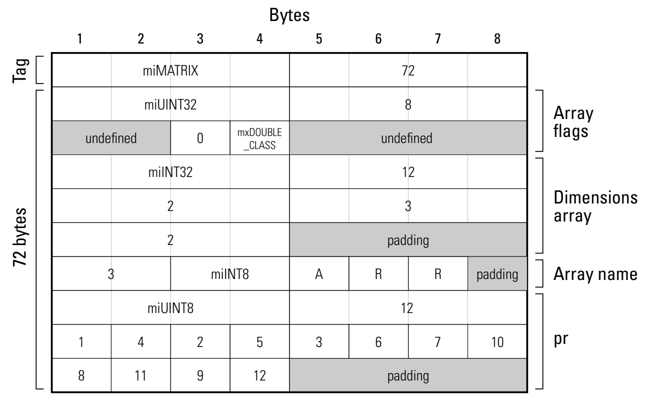


Figure 16: MAT-file level 5, example

that is having a file written on itself, would end up in the total loss of the data. With the new arrangement only a short stream of data is lost corresponding to the last 128 KB file being written. Moreover the choice of dividing the files into many folders has been done in order to improve file creation speed; every time a new file creation is requested on a FAT volume, a check on the name is performed in order to verify its uniqueness in the current folder. This makes file creation time proportionally increase with folder file number; having a limited file number in each folder guarantees a fixed file creation time. The FTI also creates a txt log file into the main directory that contains all the folder; in order to avoid various problems, the import feature of the software does not require to open all the .FTB file but, instead, it requires the selection of the .txt log file; from the log file content, the software extracts the actual number of file created and the missed file. With the information obtained from the .txt, it automatically opens all the .ftb files and imports them. Each .FTB file contains a sequence of messages; message integrity is not guaranteed in a single file because the data stream could be divided into two files. The import feature is

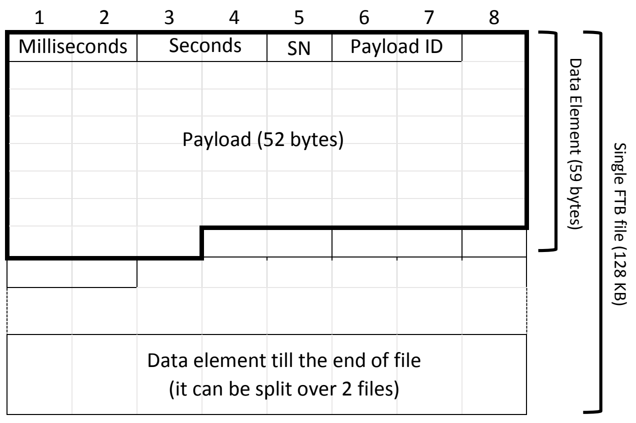


Figure 17: FTB file format

able to ensure a correct data flow by rebuilding also the messages split onto two files.
The message is different from the previous versions; instead of containing a single value it may contain more measures at the same timestamp. To import them, the application unpacks it into many instances.
As shown in the data element structure in table 4, the new message

Datatype	Identifier	Description
uint 32 bit	timestamp	It contains time information: it can be unpacked into two uint_16 one for milliseconds and one for seconds
uint 8 bit	SN	Sequence number: it is used during import phase to find missing data elements (it is incremental, 0-255)
uint 16 bit	payload_ID	Payload structure identifier: it used to recognize the payload structure to be applied while importing the data element
payload _union_t	payload	This contains the data measured by the FTI (to understand it's structure see tables 5)

Table 4: FTB file format, data element structure

type (59 Bytes) contains a payload ID that indicates the structure of the payload (it is also used as node ID when importing data), a sequence number (to find missing timestamp because it's always incremental, 0-255) and the timestamp, that, without any process, is a useless value because 16 bit indicate number of seconds and 16 bit the number of milliseconds (so bytes must be read separately).
As said above the payload structure to be used (table 5) is chosen according to the payload ID; the union has the dimension of the biggest structure (52 Bytes) and all the payload structures are packed forcing the software to read one byte each time because there is no padding

Datatype	Identifier	Description
caffe_data_irs_t	as_irs	Inertial measurement unit data (e.g. pitch rate, heading)
caffe_data_ads_t	as_ads	Air data (e.g. static pressure, AoA)
caffe_data_gpsSol_t	as_gpsSol	GPS information (e.g. ECEF data, GPS time)
xsens_pkt_t	as_xsens	AHRS data (e.g. magnetic field, yaw rate)
caffe_data_analog1_t	as_analog1	Analog channels data
char array 52 bytes	as_Bytes	This datatype is used to reconstruct the data element in the import phase

Table 5: FTB file format, payload union

left from FTI system. Every payload structure has a similar construction: first a status value is placed before data; this value must be then read one bit at a time, as every bit indicates the status of one of the measures in the structure (0 no measure, 1 measure is existent).
The analog1 payload structure shown in table 6, apart from being a

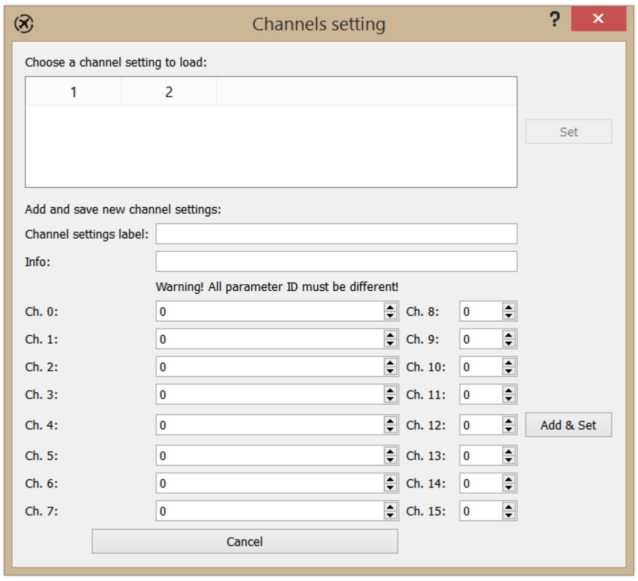
Datatype	Identifier	Description
uint 16 bit	status	Its bits indicates the presence (1) or the absence (0) of each measure contained
int 16 bit	ch_0	First channel measured value
int 16 bit	ch_1	Second channel measured value
...	...	...
int 16 bit	ch_7	Last channel measured value

Table 6: FTB file format, payload, analog1 structure

clear example of a payload structure, is also a singular one; this object in fact does not have fixed parameters assigned to its channels but they can be assigned during the import process. Obviously in order to help users, the setting of the channels is not to be performed every time, but when it is done one time, it is saved in a dedicated table into the database, ready to be loaded for the next use (if setting is the same), for details see Appendix B.

4.1.4 Local Frame Coordinate Calculation Tool

A feature to calculate the local frame coordinates was present in the previous release but it set automatically the local reference at the start of the recording; reaching the end of the construction the need for a similar tool appeared but in an improved version. The new local frame coordinate tool allows the selection of the reference either


Channels setting

Choose a channel setting to load:

1	2

Set

Add and save new channel settings:

Channel settings label:

Info:

Warning! All parameter ID must be different!

Ch. 0:	0	Ch. 8:	0
Ch. 1:	0	Ch. 9:	0
Ch. 2:	0	Ch. 10:	0
Ch. 3:	0	Ch. 11:	0
Ch. 4:	0	Ch. 12:	0
Ch. 5:	0	Ch. 13:	0
Ch. 6:	0	Ch. 14:	0
Ch. 7:	0	Ch. 15:	0

Add & Set

Cancel

Figure 18: Import tool, channels setting window

from the position of the aircraft at a selected time instant (from TOP list) or from the selection of ground station position (using GPS Earth-centered Earth-fixed (ECEF) ground coordinates).
The local frame coordinate vectors have fixed node and parameter IDs so they are stored in the database like every other vector; if a new reference is calculated it overwrites the existing one. The calculation [3] of geodetic coordinates (longitude λ, latitude ϕ, altitude h) from ECEF ones (x, y, z) is an iterative process because, except in the case of altitude equal to zero, there is no closed form; luckily few steps, less than four usually, provide a very accurate solution.

$$\lambda = \mathrm{atan2}\,(x, y) \tag{1}$$

First the longitude (eqn. 1) is computed using 4-quadrants arc-tangent.

$$p = \sqrt{x^2 + y^2} \tag{2}$$

$$\phi = \mathrm{atan2}\,(p, z) \tag{3}$$

This value (eqn. 3) for the latitude is just an initial estimation to be used as starting value for the iterating process (eqn. 4-5).

$$h_i = \frac{p}{\cos(\phi_i)} - R_N(\phi_i) \tag{4}$$

$$\phi_{i+1} = \mathrm{atan}\left(\frac{z}{p}\left(1 - e^2 \frac{R_N(\phi_i)}{R_N(\phi_i) + h_i}\right)^{-1}\right) \tag{5}$$

where (see fig. 19)

$R_N(\phi) =$ longitude radius of curvature
$e =$ ellipsoid eccentricity

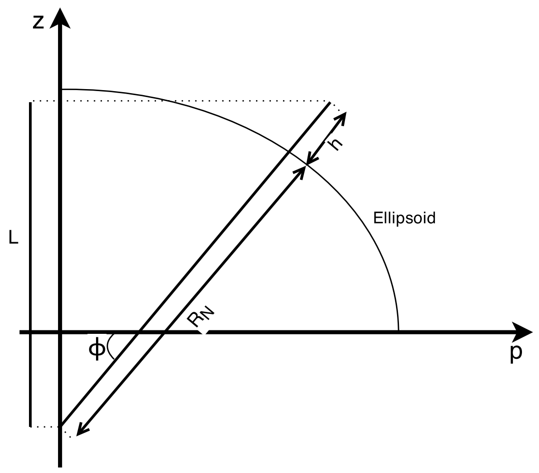
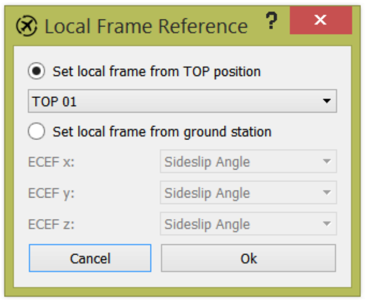


Figure 19: Geodetic coordinates, longitude radius of curvature


Local Frame Reference

(●) Set local frame from TOP position

TOP 01

( ) Set local frame from ground station

ECEF x:	Sideslip Angle
ECEF y:	Sideslip Angle
ECEF z:	Sideslip Angle

Cancel Ok

Figure 20: Local frame reference window

The equation for the altitude diverges at poles, so another one must be used (eqn. 6).

$$h = \frac{L}{\sin(\phi)} - R_N(\phi) \tag{6}$$

$$L = z + e^2 R_N(\phi)\sin(\phi) \tag{7}$$

After the calculation of the geodetic coordinates (i.e. latitude, longitude, altitude) and after the choice of a local reference point, it is possible to calculate [4] the local East North Up (ENU) coordinates (x_0, y_0, z_0).

$$\begin{bmatrix} E \\ N \\ U \end{bmatrix} = \begin{bmatrix} -\sin(\lambda_0) & \cos(\lambda_0) & 0 \\ -\sin(\phi_0)\cos(\lambda_0) & -\sin(\phi_0)\sin(\lambda_0) & \cos(\phi_0) \\ \cos(\phi_0)\cos(\lambda_0) & \cos(\phi_0)\sin(\lambda_0) & \sin(\phi_0) \end{bmatrix} \begin{bmatrix} x - x_0 \\ y - y_0 \\ z - z_0 \end{bmatrix} \tag{8}$$

4.1.5 Air Data Calculation Tool

A simple air data calculation tool was already implemented in the previous version but it was an automatic procedure done during file

import. After reaching a good point in the building phase, it has been requested to develop a new improved tool. The new tool is not automatically executed during import phase because data usually require a calibration to be used in the correct manner; for this reason the widget leaves to the user free choice to select the vectors to be used. Moreover, during the previous years it has been detected that sometime some of the needed information were missing or gathered manually by flight test engineers; the new tool is able to handle this kind of situation by allowing the user to insert manually constant values for QNE, QNH and static air temperature (static and dynamic pressure obviously cannot be inserted manually, otherwise everything would become a constant value).
The air data tool calculates those new vectors: Equivalent Air Speed (EAS),Calibrated Air Speed (CAS),True Air Speed (TAS), Mach number, rate of climb, QNE altitude, QNH altitude, QFE altitude. All the equations [1] are referred to altitudes lower than 11000 meters and subsonic speeds to reduce processing time by avoiding some if clauses; this is allowed because the tested aircraft reach speed lower than sound speed and stay below the tropopause.

Equation 9 is used to calculate QNE, QNH and QFE altitude depending on the value assigned to P_{ref} (respectively standard MSL pressure, barometric pressure adjusted to sea level and pressure at the airfield).

$$h = \frac{T_0}{a}\left(1 - \left(\frac{P_s}{P_{ref}}\right)^{\frac{-a\ R}{g_0}}\right) \tag{9}$$

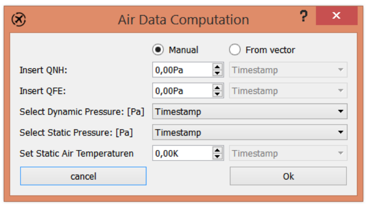
To calculate the rate of climb (eqn. 10) the knowledge of the pressure rate is needed. It is calculated with the forward difference method ($\dot{P} = \frac{P_{i+1}-P_i}{t_{i+1}-t_i}$) which is quick and simple. Obviously, the last point will be lost.

$$RoC_i = -\frac{R\ T_0}{g_0\ P_{s_i}} \frac{P_{s_{i+1}} - P_{s_i}}{t_{i+1} - t_i} \tag{10}$$

The mach number is evaluated before the speeds because it is directly used in the evaluation of the CAS and TAS.
It must be pointed out that these equations are not used as shown. In the software, to improve processing speed, all the numerical operations that can be done without the variables (static and dynamic pressures and static air temperature) have been already computed during the code writing phase and only the final numerical value is actually present.

$$M = \sqrt{\frac{2}{\gamma - 1}\left(\left(\frac{q_d}{P_s} + 1\right)^{\frac{\gamma-1}{\gamma}} - 1\right)} \tag{11}$$

$$CAS = \sqrt{\frac{2\,\gamma}{\gamma - 1}\frac{P_0}{\rho_0}\left(\left(\frac{q_d}{P_0} + 1\right)^{\frac{\gamma-1}{\gamma}} - 1\right)} \tag{12}$$



Air Data Computation

[✓ Manual] From vector

Insert QNH:	0,00Pa	Timestamp
Insert QFE:	0,00Pa	Timestamp
Select Dynamic Pressure: [Pa]	Timestamp	
Select Static Pressure: [Pa]	Timestamp	
Set Static Air Temperaturen	0,00K	Timestamp

Figure 21: Air data calculation, input window

$$EAS = \sqrt{2\frac{q_d}{\rho_0}} \tag{13}$$

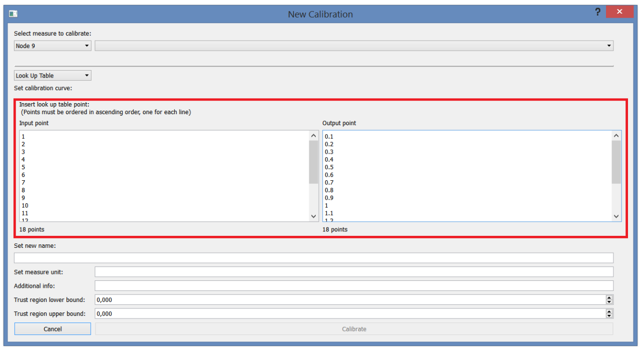
$$TAS = \sqrt{\frac{2\,\gamma\,R}{\gamma - 1}\,T\left(\left(1 + \frac{q_d}{P_s}\right)^{\frac{\gamma-1}{\gamma}} - 1\right)} \tag{14}$$

where

T_0 = ISA static air temperature at acsmsl (288.15 K)
P_S = static air pressure
a = lapse rate$(-6.5K/km)$
g_0 = gravitational acceleration at MSL
R = specific gas constant$(287.06J/kg\ K)$
P_{ref} = reference pressure
h = altitude
t_i = time at instant i
γ = heat capacity ratio
ρ_0 = air density at MSL according to ISA$(1.225kg/m^3)$
q_d = dynamic pressure
T = static air temperature
M = Mach number

4.1.6 Database Insert Problem

While implementing the save-to-database feature, a big problem came out. Initially the function was implemented like a single query for each point of each vector but soon this resulted in an awfully large amount of time to complete the user request; one and half hours were the approximate predicted time needed to perform the action. This was too much to allow a good use of the software. The first thoughts were about the SQLite real data writing approach: every time it receives a query it must write the rollback journal, perform it and then proceed to the next one. Obviously, this put a big limit on database performance given by hard disk data writing speed.
In order to solve this, the first try has been to change database journal mode into Write Ahead Log (WAL), this mode simply append to the journal the changes and the database remain intact (instead in normal



Select measure to calibrate: Node 9

Look Up Table

Set calibration curve:

Insert look up table point:
(Points must be ordered in ascending order, one for each line)

Input point	Output point
1	0.1
2	0.2
3	0.3
4	0.4
5	0.5
6	0.6
7	0.7
8	0.8
9	0.9
10	1
11	1.1
18 points	18 points

Set new name:

Set measure unit:

Additional info:

Trust region lower bound: 0,000

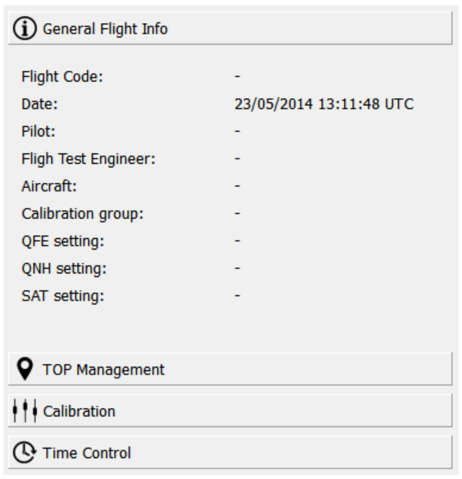
Trust region upper bound: 0,000

Figure 22: Look-up table insert window

mode it write the unchanged data in the rollback journal); this method can result in database errors in case of power loss or reboot, but it also can improve greatly writing speed. Indeed the time to save a flight was reduced to less than an hour with this method, but that is still too much for a useful software.
To further reduce dependency to hard disk write speed, a new try has been made by dividing the queries into transactions (one for each vector) so that a group of query will be performed like a single action from the database. This improved greatly the saving time up to few minutes. Even if the time was now reasonable, the hope of reducing it further and increase user comfort has led to a last attempt; points table have been removed and instead of them a file for each vector is placed into vector folder (in the same directory of the database), the file (file name: (FLIGHT_ID)_(VECTOR_ID).dat) contains various QLists, one for the measured value and one for every time attribute (i.e. week number, time of week, nanoseconds reminder). Writing directly the file allows the software to reach flight saving time of few seconds, so this is the practical solution adopted in the last version of the software.

4.1.7 Look-up table

Once the calibration features were fully developed, the Flight Test staff asked for a new option: the possibility of using look-up tables, instead of inserting the calibration curve. This new requirement presented two problems: the look-up table acquisition method and the table saving method; the actual way to use the data in fact does not represent a problem, as two QLists into the calibration class can easily manage this change. The calculation of values falling between two points of the look-up table is calculated using a linear approach. Being MATLAB® and Microsoft Excel the most common software in the department, the design of the look-up table acquisition method must provide a fast way to acquire points from them; the solution is to place a new choice in the add-calibration window. This new option, as shown into fig. 22 consists of two text-box that allow user to copy and past



General Flight Info

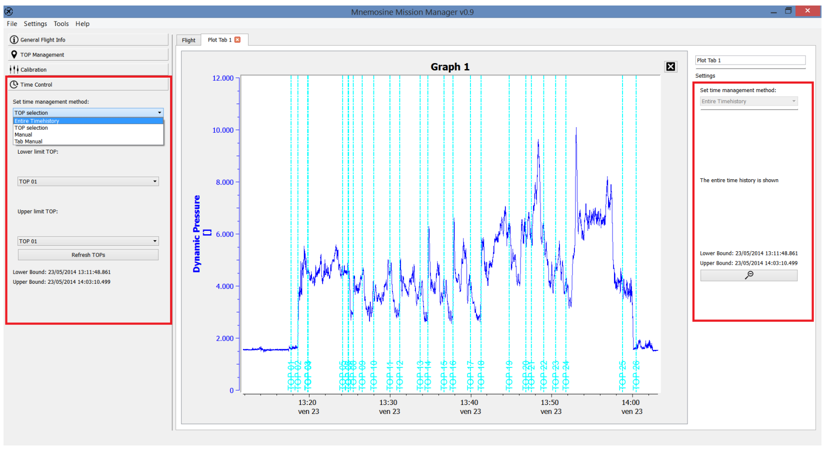
Flight Code:	-
Date:	23/05/2014 13:11:48 UTC
Pilot:	-
Fligh Test Engineer:	-
Aircraft:	-
Calibration group:	-
QFE setting:	-
QNH setting:	-
SAT setting:	-

Figure 23: Magnified stacked widget (placed on the left of the main window)

columns from a lot of software (including MATLAB® and Excel), one for x-coordinate and one for y-coordinate. The number of rows indicated below each text-box is to provide a quick visual aid to the user and to prevent the user from trying to use different number of rows (a check is also performed when calibration start).
About the look-up table saving, being the vector save feature already developed, the same kind of approach has been used, even if usually look-up tables would not contain as many points as the vectors, to prevent long processing time due to unnecessary detailed look-up tables insertion; each table is stored as a file (file name: (CALIBRATION_SET_ID)_(CALIBRATION_ID).dat) inside the folder calibration and contains a QList of x-axis a points and one of y-axis points.

4.1.8 Improved GUI

The graphic user interface is very user-friendly. All the improvement high-lightened in the design phase has been reached.
There are also some differences from the mock-up built in the design phase (sec. 3.4). The new stacked widgets on the left (see fig. 23) provide the user a way to have always main feature under his control and in a quick, intuitive way. In this part there are the main information about the flight (e.g. date and time, pilot, FTE, aircraft). They can always be helpful to the user providing to him feedback of the choices he has taken during the post-processing; both the calibration widget and their management can be done quickly without going through the main menu. The time control widget inserted in the stacked widget gives to the user a control over all the plots. For more info see sec. 4.1.9. Finally also the TOP management widget is here, giving access to the user to a TOP list through which choosing which one to shows on the plots.



Lower Bound: 23/05/2014 13:11:48.861
Upper Bound: 23/05/2014 14:03:10.499

The entire time history is shown

Lower Bound: 23/05/2014 13:11:48.861
Upper Bound: 23/05/2014 14:03:10.499

Figure 24: Time control management

4.1.9 Plot x-axis control

The control of the x-axis (time axis) has been greatly improved since the previous version. The new tool allows a very flexible usage of the software for data displaying; both synchronization and time range have various options. Regarding the synchronization, x-axes in the default mode are all connected and the time control widget (left red rectangle on fig. 24) is in charge of defining the visible range; when the appropriate option is selected in the time control widget (Tab manual), the synchronization is globally lost and only in each tab plots are connected, while controlled by the tab settings (right red rectangle on fig. 24). Moreover, also the zoom feature is synchronized, so zooming in on a part of one plot will result in a magnifying action on all plots (or tab plots if this option is selected); the zooming out can be performed with the right click or with the dedicated button on the tab settings.
The time range has many options (the same for main and tab control): default is all time history to be shown, but the user can also set range manually through two QDateTimeEdit boxes or by choosing the TOPs delimiting the desired time interval.
Attention must be payed when using the x-axis tools because only this axis is synchronized; y-axis is totally independent on each plot and only the zoom feature can vary its range. This is due to the speed improvement considerations; indeed, calculating for each x-axis variation the corresponding y-axis one, it would have required a lot of time, especially if there are many plots opened.

4.1.10 Ground Station Data Import

During the construction phase, the idea of a ground station developed into having a recording independent from the one done on the aircraft. This new recording contains data measured directly on ground (i.e. weather information, tracker data) and Mnemosine Mis-

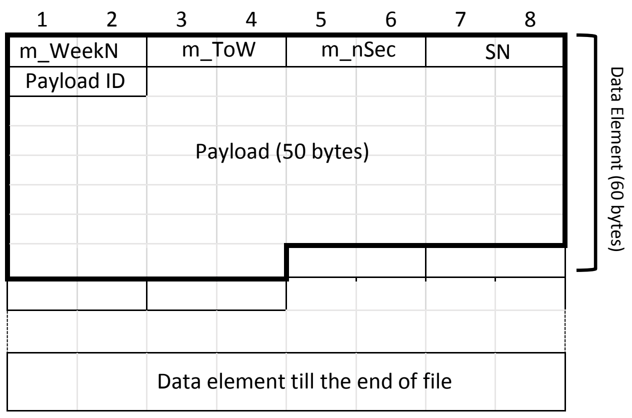


Figure 25: FTG file format

sion Manager must be able to import them. Ground data time span is not the same of the flight; in fact it can be longer, even all day, so that the software must cut the segment contained into flight time interval.

Datatype	Identifier	Description
GPS_timestamp_t	timestamp	It contains the time information: it stored directly in GPS format (see table 1)
uint 16 bit	SN	Sequence number: it is used during import phase to find missing data elements (it is incremental, 0-65535)
uint 16 bit	payload_ID	Payload structure identifier: it helps recognize the payload structure to be applied while importing the data element (i.e. weather station, tracker command, tracker measure)
GND_payload _union_t	payload	This contains the data acquired by weather station or by the tracker

Table 7: FTG file format, data element structure

As it is possible to see from figure 25 and table 7, the time format used by the ground station, it is not the same used in the data files (see sec. 2.1). The GPS format is directly used; this is due to the great amount of memory and processing power available from an actual personal computer. The use of the GPS time also allows a fast time alignment of the ground information with the flight data, which would have been difficult to overcome if a timestamp system would have been used.
The rest of the message is similar to .FTB message (see par. 4.1.3), there is a sequence number to find missing point and a payload ID to

identify payload structure. The payload structure can contain 3 different information type (i.e. weather station, tracker system command, tracker system measured) but all of them have a status datum to identify existent measures inside the payload itself.
Table 8 shows an example of a payload.

Datatype	Identifier	Description
uint 16 bit	status	Its bits indicates the presence (1) or the absence (0) of each measure contained
int 16 bit	air_temp	Air temperature
uint 16 bit	air_rh	Air relative humidity
uint 16 bit	true_speed	True wind speed
uint 16 bit	true_dir	True wind direction
uint 8 bit	icao_speed	Wind speed (ICAO Annex 6)
uint 8 bit	icao_gust	Wind gust (ICAO Annex 6)
uint 8 bit	icao_dir1	Wind direction 1 (ICAO Annex 6)
uint 8 bit	icao_dir1	Wind direction 2 (ICAO Annex 6)
uint 32 bit	qfe	Pressure at airfield
uint 32 bit	qnh	Pressure referred to MSL
int 32 bit	ecef_x	Ground station ECEF x coordinates
int 32 bit	ecef_y	Ground station ECEF y coordinates
int 32 bit	ecef_z	Ground station ECEF z coordinates
int 32 bit	lat	Ground station latitude
int 32 bit	lat	Ground station longitude
int 32 bit	lat	Ground station altitude

Table 8: FTG file format, data element, weather station payload structure

4.2 SOFTWARE DEPLOYMENT

The deployment of the software has been the last part to be performed, as it comprehended the building of the executable to be installed on the students' computers. Software deployment has been made using MSVC 32 bit version, the 64 bit release has been avoided because of problems during the compiling phase with Qwt libraries. Unluckily using 32 bit release to complete deployment generated new problems on computers with 64 bit architecture, making pc unable to recognize missing DLLs; even Dependency Walker (it displays modules imported and exported by portable executable) was not able to resolve dependency correctly. To avoid such problems the use of a x86 architecture computer is recommended during the libraries completion. Deployment phases:

- Build the release using Qt Creator (change building settings from Debug to Release)

- Copy into the folder containing the release all the icons, .ini files and the splashscreen.
- Open Qt from prompt (corresponding version, e.g. MSVC 32 bit).
- Go to bin directory of the selected version.
- Run windeployqt tool (windeployqt [filepath]), this will deploy most of the necessary DLLs into release folder.
- Load the release executable with Dependency Walker and identify the missing libraries (alternatively try to run the executable, but Windows will show only one missing Dynamic-link Library (DLL) at a time).
- Copy and paste missing libraries into executable folder.

In order to run the executable, the end-user could need to install the Microsoft Visual C++ Redistributable Package; this can be freely downloaded from microsoft.com.

5 CONCLUSION

The initial version of the software presented many problems, lacks and weaknesses. It was barely able to satisfy the needs of the users. Once new requirements appeared, it was clear that the old version was not able to satisfy them even with few improvements. So a totally new software needed to be developed.
During the planning phase the requirements were clearly pointed out, and, even if, sometimes with a great amount of work, all of them have been satisfied. Moreover, the software is also able to satisfy all the additional needs that came out during the Development phase, despite the difficulties encountered.
Furthermore, because the users are always in need for new and improved feature, the software considers that possibility and it has been built already predisposed for the new expected features (see chapter 6).

In conclusion, a great improvement since the previous version has been made; both students and staff will benefit from the new graphic user interface, the flexibility and the usability of the new release. This software will start to be used by them and not just for a quick check or just to import/export, but to really view and manipulate data.
This is just the first step for this software to become a powerful tool into the students and researchers computer; from now on it will become more and more diffused every time a new improved version will be released.

Figure 26: Mnemosine Mission Manager logo

6 FUTURE DEVELOPMENTS

The new Mnemosine Mission Manager, even if it satisfy all the requirements, leaves space to new improvements. A lot of new features could be useful to both kind of end-user.

6.1 SIGNAL FILTERING

A digital signal filtering tool could be useful in case time-histories are particularly degraded by noise, especially if aggregated to calibration widget which is already prepared for it. To avoid data changes, the filter must be zero phase. This can be easily achieved by performing two filtering forward and backward; it also provides an actual filter with double the order of the used filter.

6.2 SIGNAL SPECTRUM

To help signal filtering it would be needed a tool displaying the frequency spectrum of a signal, calculated through Fast Fourier Transform (FFT).

6.3 OUTLIERS DETECTION

Apart from the sanity check performed through the calibration tool, an outliers detection function could be useful. An adequate algorithm must be developed to find outliers in a quick, reliable way, without using bounds. It is a complex, time consuming task because there are many outliers detection algorithms that can be used depending on the context to be used; moreover, a great reliability require a lot of computational time. A good balance between reliability and computational time must be found.

6.4 PLOT IMAGE

An option to save an image of one or many plots could be interesting.

6.5 REAL TIME DATA STREAMING

There are thoughts about a possible real time data stream from the aircraft to the ground station; if this will be developed, a tool to show data in real-time would be needed. The widget could show data on digital instruments, giving the user a cockpit deck like view. More flexibility can be provided by allowing the user to choose which instruments to have on display and where.

6.6 NEW EXPORT FORMATS

Other export file-formats could be implemented to augment software flexibility (e.g. Microsoft Excel .xlsx)

6.7 PERFORMANCE EVALUATION TOOLS

Automatic performance evaluation tool could be implemented to support the Flight Test staff's job. Some performance could be:

TAKE-OFF Take off performance widget could automatically create a window showing all the parameters significant for such a phase; moreover, an automatic evaluation of take-off distance and time could be implemented.

LANDING Landing phase is similar to take off, so same feature would be needed.

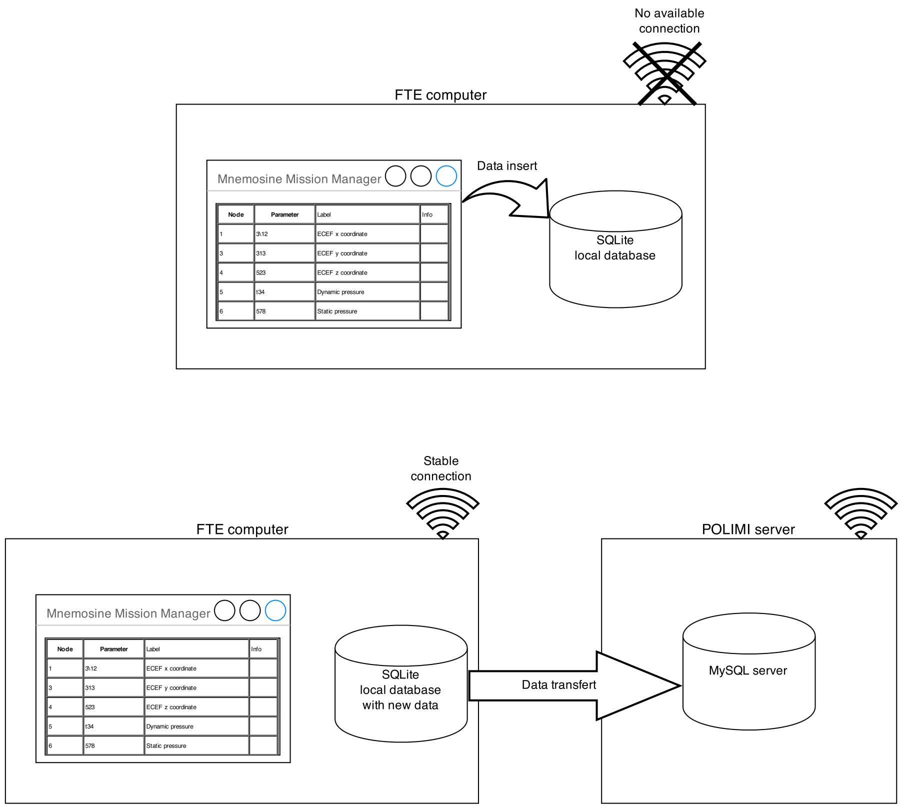
STALL Stall widget, apart from showing relevant data, could automatically (or semi-automatically) calculate stall entry rate.

PHUGOID A tool to calculate the phugoid period could be interesting maybe with some help from the user by selecting each peak.

Those examples are only here to give an idea of the tool. There are many kind of flight tests and for many of them a widget could be built. In the already mentioned features the window with significant data is a common one but another option could be shared by all of them, the automatic export of the relevant vectors data within the test time segment into a single performance data file.

6.8 SERVER DATABASE

Having a MySQL server collecting all flight data would be an added value to the department, having a big amount of information properly classified and ready to be used in no time. Software could use smaller



Node	Parameter	Label	Info
1	3\12	ECEF x coordinate	
3	313	ECEF y coordinate	
4	523	ECEF z coordinate	
5	t34	Dynamic pressure	
6	578	Static pressure	

Figure 27: Explanation of local database to server communication

SQLite database as a temporary storage until a stable connection is provided to update the server with the new data (see figure 27).
This solution, technically, can be reached by allowing the software to run in the background and periodically check the connection. Once a stable connection is detected, the server database is updated. In order to know which rows must be updated, a boolean variable could be added to each row so, every time a row is modified in the local database, its boolean variable change state signalling to the software that, when server update is being performed, this row must be replaced.
The main problem of such a solution is represented by the possibility of having many users connected at the same time to the main database; so, a way to avoid conflicts in the server database and data loss due to overwriting of a row from two different users is needed. SQL database default users conflict solution are not enough because they don't avoid the possibility of two different users of replacing both the same row one after the other.

A C++ CLASSES

A.1 DATA CLASSES

In this section are presented the C++ data classes implemented for the software. In the tables only data and the most important public methods are shown; all the method to access data (both to set or get them), if they don't present any particularity, they are omitted. Also all the private methods are avoided because they are not useful to the reader to understand the class structure.

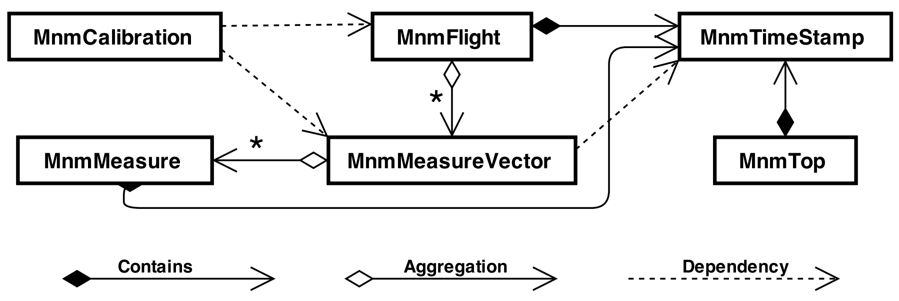


Figure 28: C++ class plan, UML model, dependency map

A.1.1 MnmTimeStamp

Datatype	Label	Description
quint16	m_WeekNum	GPS week number
quint32	m_ToW	GPS time of week
qint16	m_nSecRem	GPS milliseconds reminder

Table 9: MnmTimeStamp class, data

The timestamp class contains the time datum, in GPS format. The public methods can be divided into three categories:

- one containing all the functions to set the data inside from different inputs (the most important are *setTimeStamp* used during file import and *setTimeStampFromGPS* used during data loading from database).
- one regarding the time information return, in different time formats (*getTimeStampAsNanoS* is one of the most important because it is used every time complex mathematical operations must be

Output	Label	Arguments
void	setTimeStampFromIeee1588	t_Ieee1588 timestamp MnmTimestamp* p_Epoch
void	setTimeStamp	qint64 nanoSeconds MnmTimestamp* p_Epoch
void	setTimeStampFromGPS	quint16 WeekNum quint32 ToW qint16 nSecRem
void	setTimeStampFromUTC	QDateTime timestamp
const qint64	getTimeStampAsNanoS	
const qint64	getTimeStampAsNanoS	MnmTimestamp* p_Epoch
const Ieee1588_t	getTimeStampAsIeee	MnmTimestamp* p_Epoch
const QDateTime	getTimeStampAsUTC	
const GPStime_t	getTimeStampAsGPS	MnmTimestamp* p_Epoch
const qint64	getTimeAs	MnmTimestamp Epoch timeClass_t timeType

Table 10: MnmTimeStamp class, methods

performed with the time datum; also *getTimeStampAsGPS* is relevant due to its use during the saving to database. Finally also *getTimeStampAsUTC* has an important role in the plotting feature).

- last one containing the definition of all the logical and basic mathematical operators.

A.1.2 MnmMeasure

Datatype	Label	Description
MnmTimestamp	m_time	Time information
QVariant	m_value	Actual measure

Table 11: MnmMeasure class, data

The measure class contains the time datum and the measured value (as QVariant). The public functions, apart from logical operators and methods to get private data, provide also a way to set data inside. Particular attention must be paid to *setEpoch* which is a function used modify the time value in order to align it with the epoch during file loading.

Output	Label	Arguments
void	addWithNsTicks	quint64 nsTicks QVariant value MnmTimestamp* p_Epoch
void	setMeasure	MnmTimestamp time QVariant measure
void	setEpoch	qint64 nanoSec
const MnmTimestamp	time	
const QVariant	value	

Table 12: MnmMeasure class, methods

A.1.3 MnmMeasureVector

Datatype	Label	Description
quint16	m_parameter	Parameter ID
quint16	m_node	Node ID
quint32	m_nanoSecPerTick	Time length of one FTI tick (in nanoseconds)
QList<MnmMeasure*>	m_measureList	Vector of pointers to each measure composing the time-history
QString	m_label	Vector name
QString	m_unit	Measure unit
QString	m_description	Vector info
t_datatype	m_datatype	Measure value datatype
bool	m_onServer	Saved-to-database indicator

Table 13: MnmMeasureVector class, data

Output	Label	Arguments
void	addMeasureFromTicks	quint64 ticks QVariant value MnmTimestamp* p_Epoch t_measureType type
void	appendMeasure	MnmTimestamp time QVariant measure
MnmMeasure*	at	int i
QList <MnmMeasure*>	getList	
const QVector<double>	getMeasureVect	
const QList<quint8>	getMeasureUCHAR	
const QList<quint8>	getMeasureUCHAR	MnmTimestamp initialT MnmTimestamp finalT
const QList<type>	getMeasureTYPE	
const QList<type>	getMeasureTYPE	MnmTimestamp initialT MnmTimestamp finalT
const QList<double>	getTimeVect	
const QList<qint64>	getTimeAs	t_timeClass timeClass MnmTimestamp initialT MnmTimestamp finalT MnmTimestamp Epoch
const QList<quint16>	getTimeAsGPSweek	
const QList<quint32>	getTimeAsGPStow	
const QList<qint16>	getTimeAsGPSnsec	

Table 14: MnmMeasureVector class, methods

The Measure Vector class contains all the time-history of a single datum. Moreover, it contains also all the information about it (i.e. label, description, node, parameter, measure unit, datatype). There are also two special info inside it: the nanoseconds per tick (it is used during data appending to correctly use the tick number) and the *m_onServer* boolean (it indicates if the vector has been already saved on database and if it is changed from the last saving).
Regarding the methods, there are various functions to add measures to the list, depending on the data origin format. Then there are the usual functions to set information and to get them; particular attention must be paid on how to get the list, as there are a lot of different functions

depending on what it is needed. It is possible to get the single measure or the whole QList (attention must be paid when using those functions because they allow direct access to data members) or a QVector of the measured values (used to plot the data). There are also various functions to get a list of values in different datatypes (overloaded, without argument all the list, with arguments the selected time segment). In order to get the time information, there are also various functions: one returning a QVector (for the plot tool), one depending on the time segment and time format requested and 3 other function to return each datum contained into the GPS time format (used during the saving into the database).

A.1.4 MnmFlight

Datatype	Label	Description
QHash<quint32, MnmMeasureVector*>	m_hash	Collection of all the MnmMeasureVector of a flight
MnmTimestamp	m_epoch	Flight starting time
MnmTimestamp	m_epochEnd	Flight end time
fileType	m_fileType	File format (*.fti,*.trc,*.ftb,*.ftg)
int	m_pilotID	Pilot primary key (SQL)
int	m_FTEID	FTE primary key (SQL)
int	m_aircraftID	Aircraft primary key (SQL)
int	m_calibration	Calibration group primary key (SQL)
quint64	m_flightID	Flight primary key (SQL)
bool	m_onServer	Saved-to-database indicator

Table 15: MnmFlight class, data

The Flight class contains all the time-histories into a QHash. Their key is the composed by the node ID left shifted by 16 (nodeID «16) and the parameter ID. It contains also the Epoch (first time instant of recorded) and the Epoch end (last time instant recorded) because they are useful in many functions inside and outside this class. The other information stored are related to database and become useful when a database saving is requested in order to have all the information necessary to insert or replace a row into flight table. Also here a *m_onServer* boolean exist because it provides a quicker way, in some cases, to check flight status instead of inspecting each vector.
Looking at the methods, special attention must be paid to some of them; the functions *at* and *getTable* allow direct access to the data mem-

Output	Label	Arguments
MnmMeasureVector*	at	quint32 key
QHash<quint32, MnmMeasureVector*>*	getTable	
void	toServer	
void	addTelegram	t_telegram* telegram t_loggedItem* logItem
void	addNewTelegram	t_newtelegram* telegram QList<quint16>* channel
void	addGNDTelegram	t_udptelegram* telegram
void	addVector	quint32 key MnmMeasureVector* newVect QTableWidget* table bool check
void	fillTable	QTableWidget* table

Table 16: MnmFlight class, methods

bers, so that the user can end up modifying the information stored (this is a desired behavior because sometimes it speeds up the process). The function *toServer* is used when the flight is saved into the database to set all the vectors boolean variables to true. All the *addTelegram* functions provide methods to add a new single measure to a vector stored into QHash, depending on import file structure and measure parameter.
fillTable is a function that fills the table (GUI) of the main window with a row for each vector containing its data; *addVector*, after adding the vector to the QHash, does it automatically. Moreover, if the boolean variable *check* is set to true, it verifies if the vector parameter was already existent and overwrites the existing vector.

A.1.5 MnmTOP

Datatype	Label	Description
MnmTimestamp	m_time	TOP time
t_TOPclass	m_class	TOP class (in flight, on ground post-processing, undefined)
QString	m_label	TOP label
QString	m_text	TOP brief description
QString	m_ID	TOP ID (assigned by TOP widget)
QCheckBox	p_checkBox	CheckBox for the TOP widget (public member)

Table 17: MnmTOP class, data

The TOP class is very simple, it is mainly a container for the TOP information waiting to be used either in the plot or in the saving and exporting features. The only characteristic that is worth an explanation is the pointer to a QCheckBox inside the public members; this check-box is used in the TOP widget to allow the user to choose which TOPs will be shown on the plots.

A.1.6 MnmCalibration

Datatype	Label	Description
quint16	m_parent_node	Calibration parent node ID
quint16	m_parent_param	Calibration parent parameter ID
quint16	m_child_param	Calibration child parameter ID
QString	m_label	Calibration label
QString	m_unit	Calibration child measure unit
QString	m_unit	Calibration information
double	m_maxValue	Sanity check upper boundary
double	m_minValue	Sanity check lower boundary
QList<double>	m_xvec	Look-up table input points
QList<double>	m_xvec	Look-up table output points
t_calCurve	m_curve	Calibration curve (coefficients and identifier)
MnmFlight*	p_flight	Pointer to main flight element

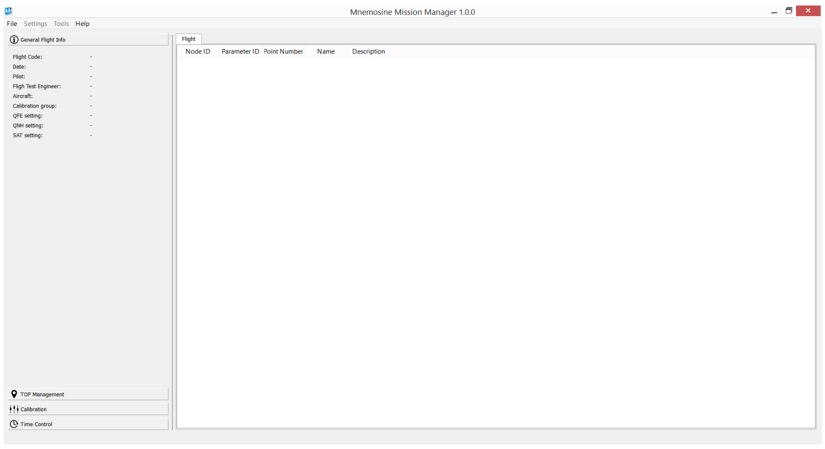
Table 18: MnmCalibration class, data

The calibration class, as for TOP class, is mainly just a data container which is used by the calibration widget or by save and export tools. The methods present no particular aspects and just provide a way to insert and get data from the private members. The data stored are clearly described in section 3.2.5, with the only difference, as said in the Development and Testing chapter, of the presence of two QList used to store the look up table if necessary.
A noteworthy characteristic is the presence of a pointer to the main MnmFlight element, this is needed in order to execute a lot of functions that request the access to the flight to add, remove or access MnmVector elements from the hash list.

A.2 GRAPHICAL USER INTERFACE CLASSES

This section presents briefly the GUI classes; for each one an image is presented and its main useful functions, because pasting all the header would result in too many methods, signals and slots which are not always very self explaining and they are sometimes only used to perform a very specific feature or resulting in a GUI output.

A.2.1 Main Window



Flight Code:	-
Date:	-
Pilot:	-
Fligh Test Engineer:	-
Aircraft:	-
Calibration group:	-
QFE setting:	-
QNH setting:	-
SAT setting:	-

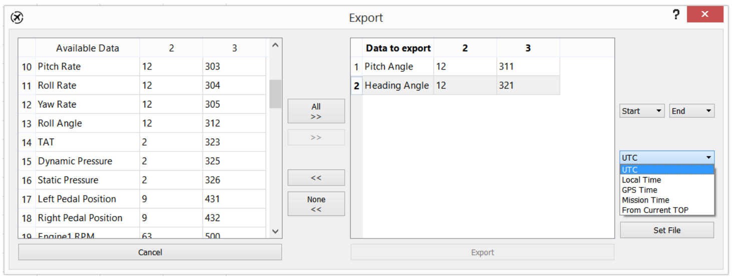
Node ID	Parameter ID	Point Number	Name	Description

Figure 29: Mnemosine Mission Manager, main window

The main window is a container for all the other widgets. It connects them and provides a way to the user to access to them. Moreover the main window contains a mnmFlight element which is the core of the whole software. The main window also have a significant role into avoiding incorrect use of the software by managing the status (enabled or disabled) of every widget inside it.
Furthermore this object includes some very important methods; the functions to import files and to load and save the flight from/to database are inside this class and the connection to the database itself is created inside it.

A.2.2 Export Window

The export window obviously contains the function to export data in one of the possible file formats (*.mat; *.csv). Moreover it creates everytime a *txt file in the same folder with the details of the exported data (file date and time creation, flight date and time, vectors exported, time segment choosen and time format selected).

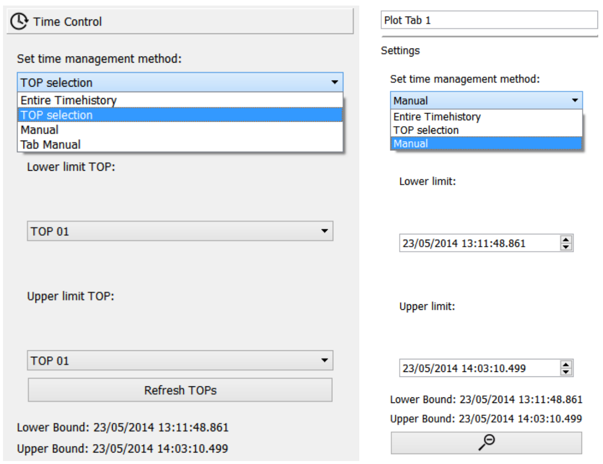


	Available Data	2	3
10	Pitch Rate	12	303
11	Roll Rate	12	304
12	Yaw Rate	12	305
13	Roll Angle	12	312
14	TAT	2	323
15	Dynamic Pressure	2	325
16	Static Pressure	2	326
17	Left Pedal Position	9	431
18	Right Pedal Position	9	432
19	Engine1 RPM	63	500

	Data to export	**2**	**3**
1	Pitch Angle	12	311
2	Heading Angle	12	321

UTC
UTC
Local Time
GPS Time
Mission Time
From Current TOP

Figure 30: Mnemosine Mission Manager, export dialog



Set time management method:
TOP selection
Entire Timehistory
TOP selection
Manual
Tab Manual
Lower limit TOP:
TOP 01
Upper limit TOP:
TOP 01
Lower Bound: 23/05/2014 13:11:48.861
Upper Bound: 23/05/2014 14:03:10.499

Set time management method:
Manual
Entire Timehistory
TOP selection
Manual
Lower limit:
23/05/2014 13:11:48.861
Upper limit:
23/05/2014 14:03:10.499
Lower Bound: 23/05/2014 13:11:48.861
Upper Bound: 23/05/2014 14:03:10.499

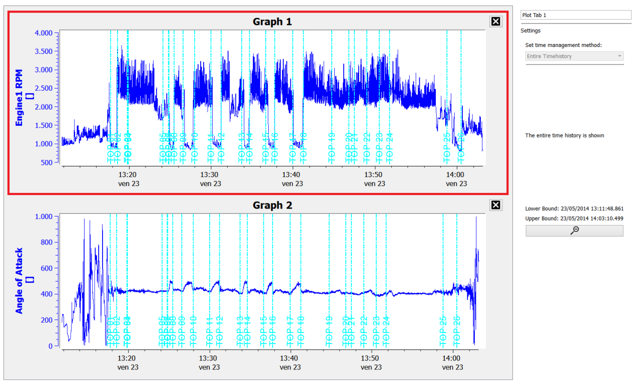
(a) (b)

Figure 31: Mnemosine Mission Manager, main time control widget (a) and tab time control widget (b)

A.2.3 Time Control Widgets

The main time control widget is placed on the left side of the main window inside a stacked layout. The main time control widget has an essential role in the graphs management as it controls the x-axis range of all of them. It provides a signal to all the tab with the time range to be shown. Moreover, it can enable or disable the tab time control widget, allowing each tab to independently select a x-axis interval. On the lower part a label shows the actual time range selected in order to provide to the user an additional feedback.
Particular attention must be paid when TOP selection is used because it does not automatically refresh the TOP list in the combo boxes, therefore explained the presence of the *Refresh TOPs* button.
The tab time control widget is placed on the right side of the plot tab; when enabled it allows direct control on the plot tab graphs x-axis. The tab time control widget receives the main time control widget signals and filter them depending on its own status (enabled or disabled); if it is enabled the signal is re-transmitted to each plot space. The widget also contains a button to zoom out (alternatively it can be used the right mouse click).

A.2.4 Plot Tab and plot space



Plot Tab 1

Settings

Set time management method: Entire Timehistory

The entire time history is shown

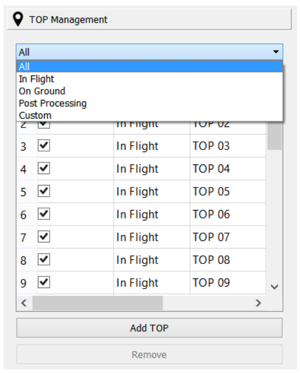
Lower Bound: 23/05/2014 13:11:48.861

Upper Bound: 23/05/2014 14:03:10.499

Figure 32: Mnemosine Mission Manager, plot tab and plot space (in the red rectangle)

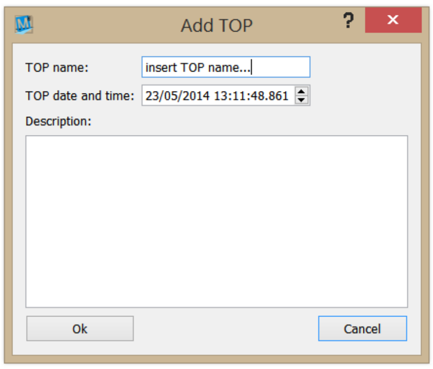
The plot tab is a container for the graphs. Every time a graph is added all of them resize to properly fit the available space. The plot tab has also an important role in managing the actions in the main table contextual menu, as it is the plot tab that contains the tab submenus and the add-plot-to-tab actions.
The plot space is a widget containing the Qwt plot element and a button to close itself; the plot space contains also the add-curve-to-plot action of the main table contextual menu. The plot space manages all the signals either from the time control widgets or from the mouse, as the zooming action, and runs the correct function.

A.2.5 TOP management widget



3	✓	In Flight	TOP 03
4	✓	In Flight	TOP 04
5	✓	In Flight	TOP 05
6	✓	In Flight	TOP 06
7	✓	In Flight	TOP 07
8	✓	In Flight	TOP 08
9	✓	In Flight	TOP 09

(a)



TOP name: insert TOP name...

TOP date and time: 23/05/2014 13:11:48.861

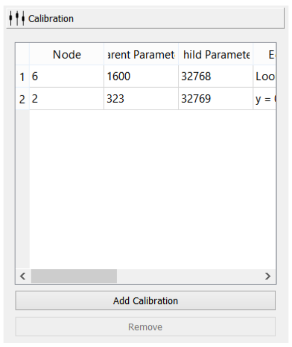
Description:

(b)

Figure 33: Mnemosine Mission Manager, TOP management widget (a) and add-TOP window (b)

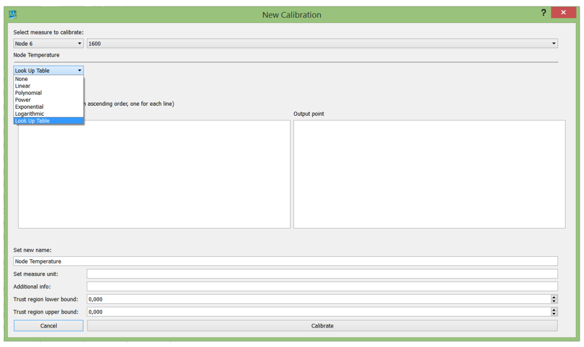
The TOP management widget contains all the TOPs of the current flight. It can get the TOP from the database or directly find them from the TOP counter vector. More TOPs can be added from the software and they will be directly classified as post-processing. The remove button is only enabled when a post-processing file is selected, the other kind of TOPs cannot be deleted in order to preserve original data.
Every request for the TOPs either to save them, use them to select a time segment or show them on plot must go through this widget.

A.2.6 Calibration Widget



	Node	arent Paramet	hild Paramete	E
1	6	1600	32768	Loo
2	2	323	32769	y = (

(a)



Select measure to calibrate: Node 6 | 1600
Node Temperature
n ascending order, one for each line)
Output point
Set new name: Node Temperature
Set measure unit:
Additional info:
Trust region lower bound: 0,000
Trust region upper bound: 0,000

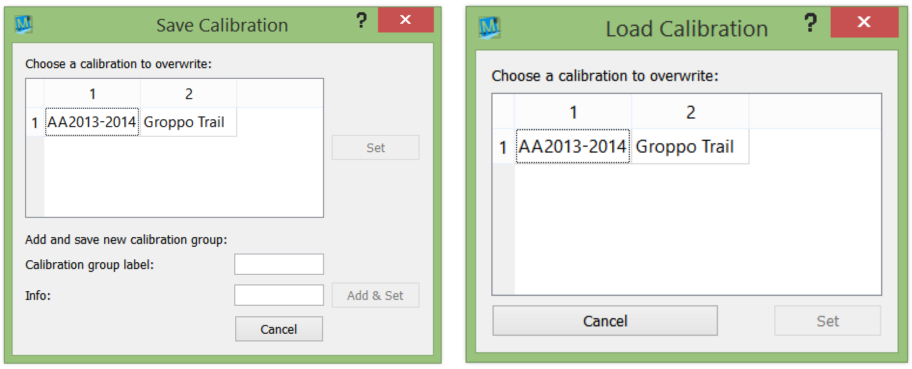
(b)

Figure 34: Mnemosine Mission Manager, Calibration management widget (a) and add-calibration window (b)

The calibration widget contains and manages all the calibrations. The add button opens the add calibration window which is a flexible, quick way to manipulate data; a parameter ID is automatically assigned to every calibration performed starting from 32768 (so every vector shown in the main table having a parameter ID equal or greater than this value contains a calibrated vector). The calibration widget is also in charge of performing a sanity check to all vectors that go through it by checking, for every datum, if it is inside the range selected by the user.
The calibration widget also performs the loading from the database of a calibration group and every time a calibration is loaded it checks if the parent vector exists.

A.2.7 Save and load calibration windows

To access to these windows the user must go into menu bar, submenu calibration. They allow the user to respectively save, as new or by overwriting another one, the current calibration group load and load calibration group from the database and send it to the calibration management widget to process it.


Save Calibration

Choose a calibration to overwrite:

	1	2
1	AA2013-2014	Groppo Trail

Add and save new calibration group:

Calibration group label:

Info:

Load Calibration

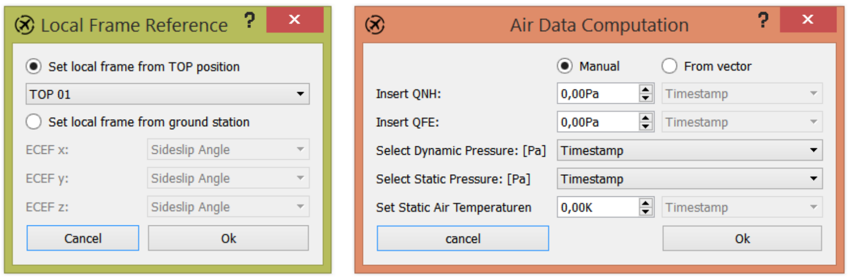
Choose a calibration to overwrite:

	1	2
1	AA2013-2014	Groppo Trail

(a) (b)

Figure 35: Mnemosine Mission Manager, Save-calibration-group window (a) and Load-calibration-group window (b)

A.2.8 Local frame coordinates and air data windows


Local Frame Reference

● Set local frame from TOP position

TOP 01

○ Set local frame from ground station

ECEF x: Sideslip Angle

ECEF y: Sideslip Angle

ECEF z: Sideslip Angle

Air Data Computation

● Manual ○ From vector

Insert QNH: 0,00Pa Timestamp

Insert QFE: 0,00Pa Timestamp

Select Dynamic Pressure: [Pa] Timestamp

Select Static Pressure: [Pa] Timestamp

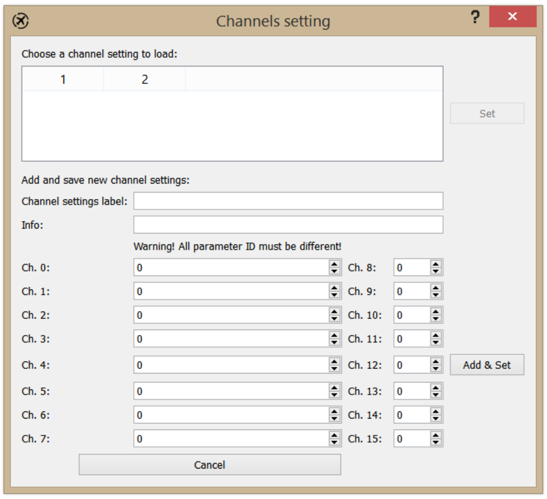
Set Static Air Temperaturen 0,00K Timestamp

(a) (b)

Figure 36: Mnemosine Mission Manager, local frame reference window (a) and air data calculation input window (b)

The local frame reference window allows the user to select the origin for the ENU coordinates; the air data window instead allows the user to enter or select the data to use for the calculation of QNE, QNH, QFE, rate of climb, CAS, EAS, TAS and mach number. They both have been widely discussed in the Development and Testing chapter, for more info see sections 4.1.4 and 4.1.5.

A.2.9 Channels setting window



Choose a channel setting to load:

1	2

Add and save new channel settings:

Channel settings label:

Info:

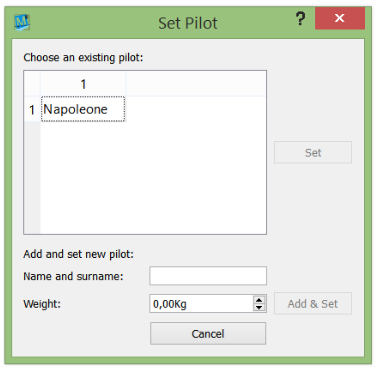
Warning! All parameter ID must be different!

Ch. 0:	0	Ch. 8:	0
Ch. 1:	0	Ch. 9:	0
Ch. 2:	0	Ch. 10:	0
Ch. 3:	0	Ch. 11:	0
Ch. 4:	0	Ch. 12:	0
Ch. 5:	0	Ch. 13:	0
Ch. 6:	0	Ch. 14:	0
Ch. 7:	0	Ch. 15:	0

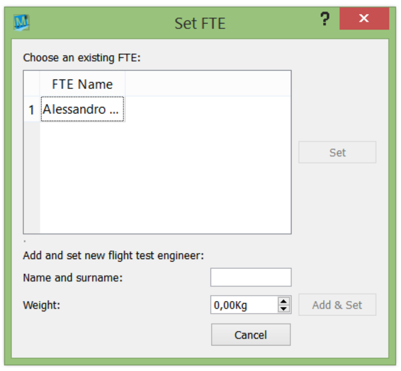
Figure 37: Mnemosine Mission Manager, channel setting window

The channel setting window appears to the user only in case a .FTB file is being imported (actually a .txt is the main file). The channel window allows the user to assign parameters to the channels of the analog structure of the FTB file format data element. The .FTB fle format and the current windows has been already discussed in the Development and Testing chapter, for more information see section 4.1.3.

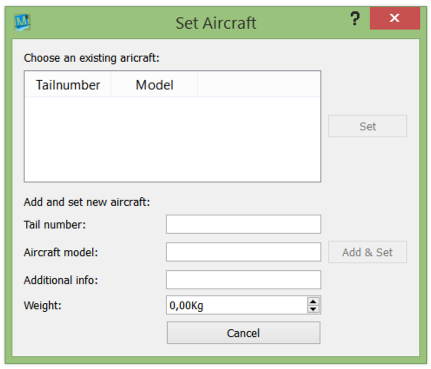
A.2.10 Other utility windows



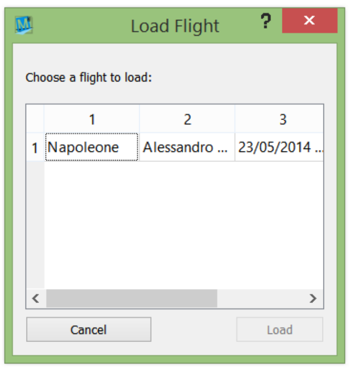
(a)



(b)



(c)



(d)

Figure 38: Mnemosine Mission Manager, set pilot window (a), set FTE window (b), set aircraft window (c) and select flight to load window (d)

The set-pilot, set-FTE, set-aircraft and load-flight windows are just a way to access to database to FTE, pilot, aircraft and flight tables and to allow the user to select an existing row or add a new row into them (not the load flight one). Their layout is simple and intuitive, avoiding useless buttons that could confuse the user.

B SQLITE DATABASE

B.1 FLIGHT TABLE

```
CREATE TABLE FLIGHT (
   ID UNSIGNED BIGINT PRIMARY KEY NOT NULL UNIQUE,
   FTE_ID INTEGER REFERENCES FTE (ID) MATCH SIMPLE,
   PILOT_ID INTEGER REFERENCES PILOT (ID) MATCH SIMPLE,
   AIRCRAFT_ID INTEGER, CALIBRATION_SET_ID INTEGER,
   QFE DOUBLE, QNH DOUBLE, SAT DOUBLE,
   START_WeekN UNSIGNED SMALLINT, START_ToW UNSIGNED INT,
   START_nSec SMALLINT, END_WeekN UNSIGNED SMALLINT,
   END_ToW UNSIGNED INT, END_nSec SMALLINT)
```

uint_64 ID = (PILOT_ID)$1e^{14}$+(FTE_ID)$1e^{12}$+(flight year)$1e^{8}$ +(flight month)$1e^{6}$+(flight day)$1e^{4}$+(flight start hour)100+(flight start minute)

ID is automatically generated once the user set FTE and pilot, otherwise flight cannot be saved. It is the primary key and must be unique and not null.

B.2 PILOT TABLE

```
CREATE TABLE PILOT (
   ID INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
   name TEXT, weight DOUBLE)
```

B.3 FTE TABLE

```
CREATE TABLE FTE (
   ID INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
   name TEXT, weight DOUBLE)
```

B.4 AIRCRAFT TABLE

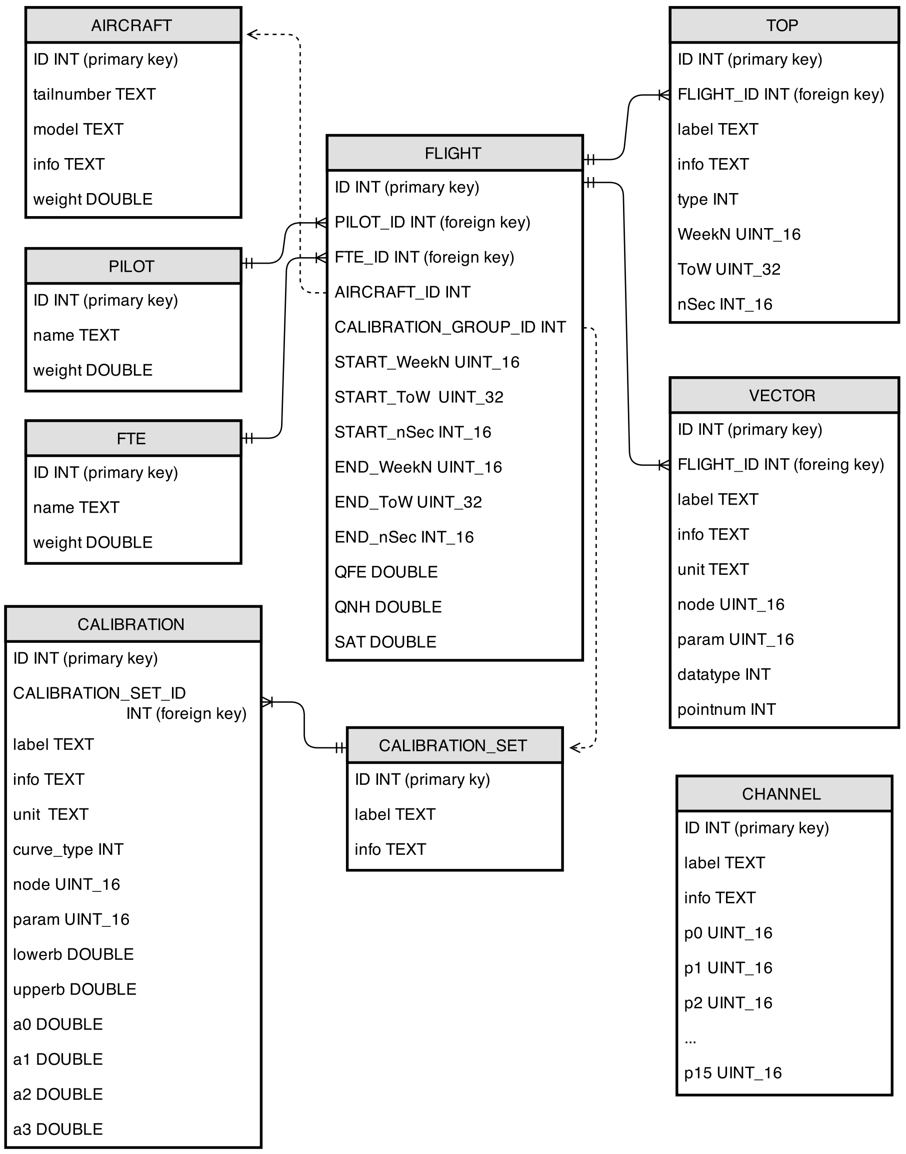


Figure 39: SQLite database structure

```
CREATE TABLE AIRCRAFT (
   ID INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
   tailnumber TEXT, model TEXT,
   info TEXT, weight DOUBLE)
```

B.5 VECTOR TABLE

```
CREATE TABLE VECTOR (
   ID INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
   FLIGHT_ID UNSIGNED BIGINT REFERENCES FLIGHT (ID)
   ON DELETE CASCADE ON UPDATE CASCADE,
   label TEXT, info TEXT,
   unit TEXT, node UNSIGNED SMALLINT,
   param UNSIGNED SMALLINT,
   pointnum UNSIGNED INT, datatype INT)
```

FLIGHT_ID has ON_CASCDE condition in order to allow automatic update of ID if FLIGHT.ID is updated and if a flight is deleted also all related vectors are deleted avoiding memory unnecessary memory employment with useless rows.

B.6 TOP TABLE

```
CREATE TABLE TOP (
   ID INTEGER PRIMARY KEY AUTOINCREMENT,
   FLIGHT_ID INTEGER REFERENCES FLIGHT (ID)
   ON DELETE CASCADE ON UPDATE CASCADE,
   label TEXT, info TEXT,
   type INT, WeekN UNSIGNED SMALLINT,
   ToW UNSIGNED INT, nSec SMALLINT)
```

ON_CASCADE option is present also in this table to be able to quickly update and remove unnecessary TOPs (in case of changes in FLIGHT table).

B.7 CALIBRATION_SET TABLE

```
CREATE TABLE CALIBRATION_SET (
   ID INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL,
   label TEXT, info TEXT)
```

B.8 CALIBRATION TABLE

```
CREATE TABLE CALIBRATION (
   ID INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
   CALIBRATION_SET_ID INTEGER REFERENCES CALIBRATION_SET (ID)
   ON DELETE CASCADE ON UPDATE CASCADE,
   node UNSIGNED SMALLINT, param UNSIGNED SMALLINT,
   label TEXT, unit TEXT, info TEXT,
   lowerb DOUBLE, upperb DOUBLE, type INT,
   a0 DOUBLE, a1 DOUBLE, a2 DOUBLE, a3 DOUBLE)
```

CALIBRATION table rows are connected with a foreign key to CALIBRATION_SET table. The ON_CASCADE option allows to update or remove the unnecessary rows here when a calibration set is modified.

B.9 CHANNEL TABLE

```
CREATE TABLE CHANNEL (
   ID INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
   label TEXT, info TEXT,
   p0 UNSIGNED SMALLINT, p1 UNSIGNED SMALLINT,
   p2 UNSIGNED SMALLINT, p3 UNSIGNED SMALLINT,
   p4 UNSIGNED SMALLINT, p5 UNSIGNED SMALLINT,
   p6 UNSIGNED SMALLINT, p7 UNSIGNED SMALLINT,
   p8 UNSIGNED SMALLINT, p9 UNSIGNED SMALLINT,
   p10 UNSIGNED SMALLINT, p11 UNSIGNED SMALLINT,
   p12 UNSIGNED SMALLINT, p13 UNSIGNED SMALLINT,
   p14 UNSIGNED SMALLINT, p15 UNSIGNED SMALLINT)
```

This table is totally independent from the other tables, because it is questioned only during file import and only if the new .FTB files are to be loaded, in order to set the free channels.

BIBLIOGRAPHY

[1] Collinson, Richard P.G., *Introduction to Avionics Systems*, Maidstone, Kent (United Kingdom) 3rd edition, 2011.

[2] Stanley B. Lippman, Josée Lajoie, *C++ Corso di programmazione*, 3rd edition, 2000.

[3] J.Farrell, M.Barth, *The Global Positioning System & Inertial Navigation*, 2nd edition, 1999.

[4] James R. Clynch, *Geodetic Coordinate Conversion*, Surveying and Geospatial Engineering, University of New South Wales, February 2006.

[5] *Precision clock synchronization protocol for networked measurement and control systems*, IEEE Standard 1588, 2004.

[6] *Qt Documentation*, `http://doc.qt.io/`, visited November 2014.

[7] *Qwt Classes Documentation*, `http://qwt.sourceforge.net/annotated.html`, visited December 2014.

[8] *SQLite Documentation*, `https://sqlite.org/docs.html`, visited January 2015.

[9] *Microsoft File System Documentation*, `https://technet.microsoft.com/en-us/library/cc938437.aspx`, visited March 2015.

[10] *MATLAB® MAT-File Format*, The MathWorks Inc., Natick, Massachusetts (USA), October 2014.