A Distributed System Interface for a Flight Simulator

A DISTRIBUTED SYSTEM INTERFACE FOR A FLIGHT SIMULATOR

BY

OMAR ZEITOUN, B.Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2014) McMaster University

(Computer & Software Engineering) Hamilton, Ontario, Canada

TITLE: A Distributed System Interface for a Flight Simulator

AUTHOR: Omar Zeitoun

B.Eng., (Mechatronics Engineering)

McMaster University, Hamilton, Canada

SUPERVISOR: Dr. Martin von Mohrenschildt

NUMBER OF PAGES: xiv, 136

بسم الله الرحمن الرحيم

"Here's To The Crazy Ones. The misfits. The rebels. The trouble-makers. The round pegs in the square holes....Because the people who are crazy enough to think they can change the world - are the ones who DO !" Steve Jobs

Abstract

The importance of flight training has been realized since the inception of manned flight. In this thesis, a project about the interfacing of hardware cockpit instruments with a flight simulation software over a distributed system is to be described. A TRC472 Flight Cockpit was to be used while linked with Presagis FlightSIM to fully simulate a Cessna 172 Skyhawk aircraft. The TRC 472 contains flight input gauges (Airspeed Indicator, RPM indicator... etc.), pilot control devices (Rudder, Yoke...etc.) and navigation systems (VOR,ADF...etc.) all connected to computer through separate USBs and identified as HID's (Human Interface Devices). These devices required real-time interaction with FlightSIM software; in total 21 devices communicating at the same time. The TRC472 Flight Cockpit and the FlightSIM software were to be running on a distributed system of computers and to be communicating together through Ethernet. Serialization was to be used for the data transfer across the connection link so objects can be reproduced seamlessly on the different computers. Some of the TRC472 devices were straight forward in writing and reading from, but some of them required some calibrations of raw I/O data and buffers. The project also required making plugins to overwrite and extend FlightSIM software to communicate with the TRC472 Flight Cockpit. The final product is to be a full fledged flight experience with complete environment and physics of the Cessna 172.

Acknowledgements

I would like to express my sincere gratitude to my advisor Professor Martin v. Mohrenschildt for his support, and patience with me at all times.

My sincere thanks also goes to my friend and professional pilot Abdalla Abdel-Rahman for always answering my questions about flight.

Last but not the least, I would like to thank my family for a lot of things that wouldn't fit in this tiny page.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

The cockpit of the plane is the area of the plane in which the pilot controls the aircraft. It contains the flight instruments and the controls that give the pilot full control over the plane. The flight cockpit used in this project is based on the cockpit of the Cessna 172 (C172). First produced in 1955, The Cessna 172 is one of the most popular flight training aircrafts in the world and the most successful aircraft in history.

In this project, the TRC472 Flight Cockpit produced by TRC Simulators was to be used with Presagis FlightSIM to fully simulate a Cessna 172 Skyhawk aircraft cockpit. The TRC472 simulator cockpit is precisely reproduced according to the exact dimensions of the original panel and the instruments can be found in exactly the same positions. The TRC472's gauges are modeled after the original gauges found in the Cessna 172 Skyhawk. They are not projected on an LCD screen like other flight simulators, but instead real instruments which behave and move like the original instruments. The TRC472 is built up to FAA/Canadian Transport specifications.

The software tool used in this project for the simulation is FlightSIM from Presagis. Presagis FlightSIM is a professional flight simulation software. It is the industry standard solution for creating high-fidelity fixed wing flight dynamic simulation. Using the developer package, you can build or modify components of the simulated flight using C/C++ such as: specifying subsystems behavior, including flight management systems, autopilot, and flight controls; and integrating virtual and/or real hardware devices and user-development simulation modules.

The virtual devices (inputs and outputs) that exist in FlightSIM will be overridden to map the data to their equivalent real USB-counterpart for both inputs and outputs. So each virtual simulator input will be mapped to its meaningful control input, and each output will be mapped to it's meaningful gauge. This will also be built over separate computers to employ distributed computing techniques to make the simulator more robust. The system is designed to be easy to extend and add devices in the future and to integrate with various other devices.

Another product worth mentioning here is VAPS XT by Presagis. VAPS is a flexible C++ object oriented Human-Machine Interface (HMI) modeling tool. It is used for rapid development of interactive graphical interfaces with unparalleled control and flexibility in the design of dynamic, interactive, real-time HMI. It can be used to expand and add virtual hardware instruments for FlightSim (Liu, 2014).

1.2 Rationale

The rationale for carrying out this research project lies in the following:

To be Used in Experiments; The system will be used in advanced experiments

examining reactions, turbulence and training abilities under different scenarios and situations in joint efforts with the psychology department at McMaster University.

Lower Cost; Flight simulators are extremely expensive to buy and maintain. Putting together our own simulator lowers the present cost and any future costs that can occur.

Advanced Simulation Software; Using Presagis FlightSIM, one of the most advanced flight simulation softwares out there, gives the system extra software capabilities.

Expandable Design; Having our own clean modular design helps making the system open to expansions and additions by having a design that makes it relatively easy to add new instruments and user plugins.

Maintainable Design; Having a modular design running on a distributed system makes the system easier to maintain as there is no single point of failure either in hardware or software.

Chapter 2

Background

2.1 History of Flight Simulation

Over the years, planes and the controls behind them have been advancing and so has been the training required to fly a plane. Flight simulation has been a major way of training a pilot before him taking a real plane off the ground. A flight simulator is a machine that simulates flight motion in parallel with the plane controls and instruments, and the environment the flight takes place in. The major reasons for using flight simulators to train pilots are: the cost of is usually lower to operate a simulator than an actual plane, and second the safety of the trainee where a pilot can be trained for emergency situations without actually being at risk.

One of the very first known flight training devices can be seen Figure 2.1. Developed around 1910, Pilot were trained on it to operate the control wheels before they would fly an actual aircraft. It consisted of a seat mounted in a two half-sections of a barrel mounted and moved manually to represent the pitch and roll of an aeroplane as training assistants would stand outside and rotate device in accordance with the

pilot's use of the wheels (Moore, 2008).

Figure 2.1: Early flight training device

The next step in the evolution of the flight training devices was to replace the need of human assistant for simulating motion with mechanical and/or electrical actuators linked to the trainer controls, so that automatically the device would be able to rotate and move the pilot's attitude corresponding to that of the real aircraft in response to his control inputs (Moore, 2008).

The real beginning of flight simulators was with Ed Link, who wanted to take flight lessons but they proved to be very expensive and Link believed the attitude of his instructor hindered his ability to follow flight instructions. Thus, he introduced one of the best known early flight simulators known, the Link Trainer(Figure 2.2), in 1929 which is considered to be the beginning of the flight simulation industry (Gabbai, 2001). It used an electrically driven suction pump mounted in the base that simulated the motion operated by the controls as in the stick and the rudder, and it also included a motor that simulated a repeated sequence of altitude disturbance. The

Link trainer was later adopted by the US army, marking the creation of the multi-billion dollar simulator industry. The first full aircraft simulator to be owned by an airline was developed by Curtiss Wright in 1948. It was full simulator of a Boeing 377 Stratocruiser developed for Pan American (Page, 2000).

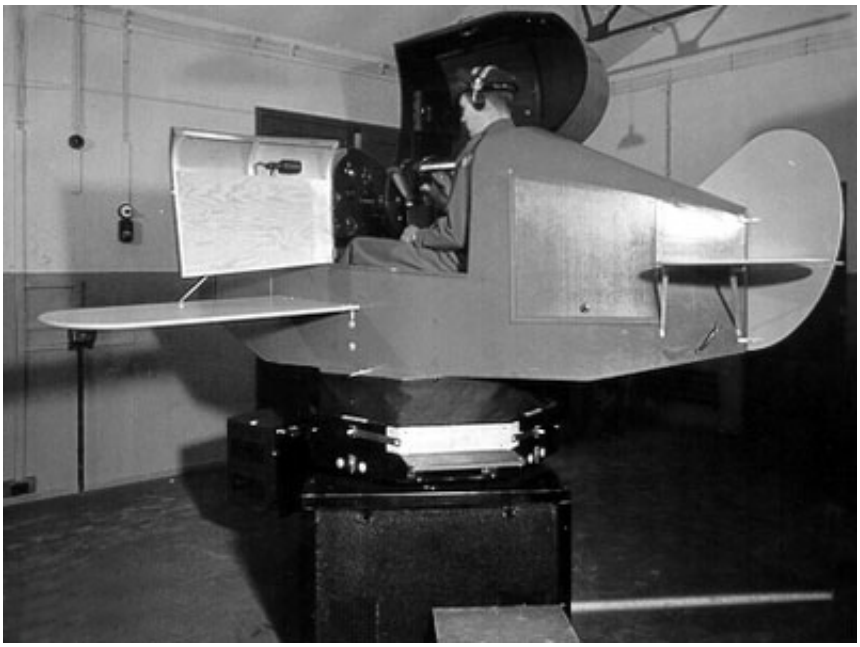


Figure 2.2: Link Trainer

Up to this point, one of the biggest challenges was having a real-time visual environment imagery. Early methods for the visual environment imagery were simulated by pre-filmed movies or a moving camera over a map. One of the first visual systems that was popular in the 1950's was the point-light source projection, where a model map was illuminated and a small camera was moved over the map terrain and projected in front of the pilot in accordance to his flight controls as can be seen in Figure 2.3. Then came the closed circuit television systems in which the scenery was on a moving belt and viewed by a servo-driven camera through an optical probe allowing for pitch, roll and yaw movements. The first color system system was produced by

Redifon in 1962 (Page, 2000).

Figure 2.3: The Mock Up Visual Terrain System

The next step in visual systems evolution was Computer Generated Images(CGI). The first use for a real-time computer generated images for simulation was the GE Computer Image Generation systems in the mid 1960s (Reisman, 1990). At the time, real time computer graphics couldn't compete with the earlier methods on the basis of complexity and realism, but with the performance of digital hardware increasing and price decreasing, it was seen that CGI is a good substitute for the other display methods staring the modern simulation era (Yan, 1985).

In 1975, Bruce Artwick developed a thesis project titled "A versatile computer generated dynamic flight display." in which he displayed a model of a three dimensional simulation of flight on the monochrome Apple II (Figure 2.4) giving birth to the concept of a flight simulator running on a what was known then as a microcomputer (Bonner, 2012).

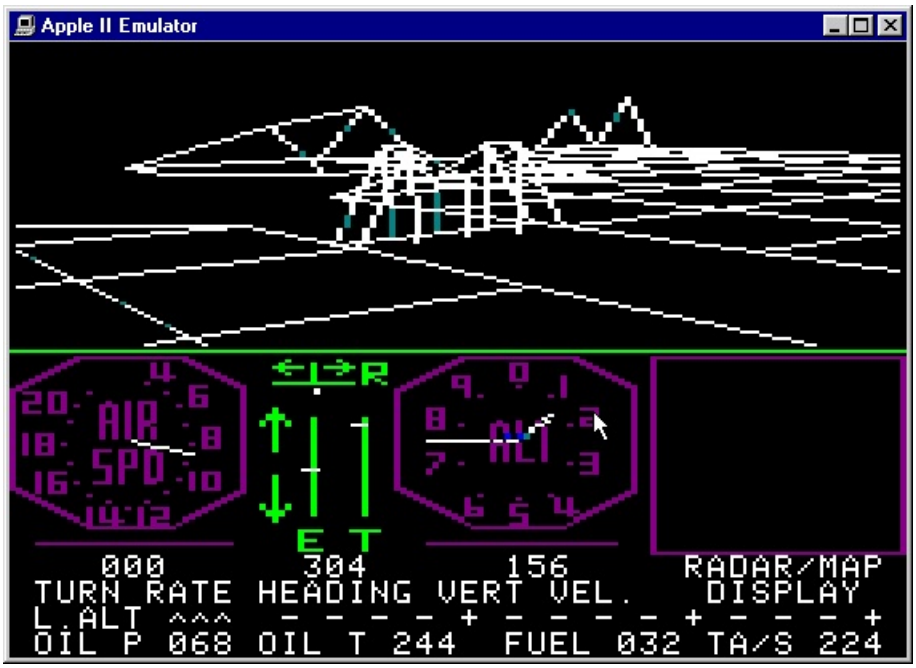


Figure 2.4: Flight simulator running on Apple II

2.2 Different Approaches of building a Flight Simulator

There are different approaches and designs to build flight simulators based on how professional and immersive is the simulation needed to be.

The first approach for a flight simulator is the all in screen approach. In this approach, all the scenery and the flight instruments lie on the monitor and you can interact with them with a mouse cursor. It's the simplest and most flexible option as you can simulate different cockpit looks for different planes. An example can be seen in Figure 2.5 of a Cessna 172 cockpit in Microsoft Flight Simulator and most of the instruments are visible. The yoke and the pedals and some of the controls here can

be simulated either using a keyboard and/or a joystick.

Figure 2.5: Cessna 172 in Microsoft Flight Simulator

The second approach is be close to the first one but more professional where you can simulate separately on different monitors the different instruments in a way that would be closer to reality. One of the best tools to do so is VAPS XT. VAPS is a flexible C++ object oriented Human-Machine Interface (HMI) modeling tool used for rapid development of interactive graphical interfaces with unparalleled control and flexibility in the design of dynamic, interactive, real-time HMI. Interaction then can be also more natural using touch screens. Using VAPS XT, one can build an object-oriented display architecture that can run on PC-based systems and can emulate production flight displays. Using the object-oriented architecture, different frames can be programmed, tested, and implemented rapidly and the multiple displays can be driven from a single computer (Liu, 2014). VAPS XT can also be used with touch-screens to develop on-display controllers with virtual buttons to emulate the same behavior as the real aircraft (Presagis, 2010). A project of this kind was developed

Jian Liu et al. and is discussed in their paper "Cockpit Display System Simulation of General Aviation Aircraft Based on VAPS XT". A similar project using VAPS XT can be seen in Figure 2.6 developed by GulfStream using VAPS (Presagis, 2010).

Figure 2.6: VAP XT flight simulator by Gulfstream

The third approach is using a hardware cockpit with real modeled instruments and real feel just like in a real plane for the most immersive environment. These instruments would be closely modeled after the real cockpit and can interact with a computer over USB. This approach gives the best simulation experience and is the most professional one. An example of these would be the TRC472 as can be seen in Figure 2.7 and which is also used in this project.

Figure 2.7: TRC472 with Microsoft Flight Simulator

2.3 Requirements for a Full Flight Simulator

There are different levels and kinds of flight simulators. According to the FAA(Federal Aviation Administration) which is generally followed by Transport Canada, there are 4 levels of full flight simulators(FFS): A, B, C, and D, D being the highest, and 3 Levels of Flight Training Devices(FTD) from 4 to 6, 6 being the highest. levels 1,2,3 are considered generic devices to be evaluated by the general aviation division not the national simulator program, and Level 7 airplane FTDs were never really used in practice and were removed from the standards.

Typically, an FTD is a training module usually used for either generic or aircraft-specific flight training. Comprehensive flight, systems, and environmental models are required. High level FTDs require visual systems but not to the level of details of a Full Flight Simulator.A Full Flight Simulator provides aircraft-specific flight training under rules of the appropriate national civil aviation regulatory authority. Under these rules, relevant aircraft systems must be fully simulated, and a comprehensive aerodynamic model is required. All FFS require outside-world visual systems and a motion platform

The following are the general categories that the a flight simulator is judged on :

(1) General flight deck configuration.

(2) Simulator programming.

(3) Equipment operation.

(4) Equipment and facilities for instructor/ evaluator functions.

(5) Motion system.

(6) Visual system.

(7) Sound system.

Based on these criterias and how advanced is the simulator, it is judged if it passes as a FTD or FFS and whether so which level does it belong to. The difference between the levels can be generally be listed as follows:

FAA FTD Level 4: This level does not require an aerodynamic model, but accurate systems modeling is required.

FAA FTD Level 5: Aerodynamic programming and systems modeling is required, but it may represent a family of aircraft rather than only one specific model.

FAA FTD Level 6: Aircraft-model-specific aerodynamic programming, control

feel, and physical cockpit are required. The FTD must simulate significant flight deck sounds resulting from pilot actions that correspond to those heard in the airplane.

It should be noted that FTD's require neither motion systems nor visual systems.

FAA FFS Level A: A motion system is required with at least three degrees of freedom.

FAA FFS Level B: Requires three axis motion and a higher-fidelity aerodynamic model than does Level A. The lowest level of helicopter flight simulator.

FAA FFS Level C: Requires a motion platform with all six degrees of freedom. Also lower transport delay (latency) over levels A and B. The visual system must have an outside-world horizontal field of view of at least 75 degrees for each pilot.

FAA FFS Level D: The highest level of FFS qualification currently available. Requirements are for Level C with additions. The motion platform must have all six degrees of freedom, and the visual system must have an outside-world horizontal field of view of at least 150 degrees, with a Collimated (distant focus) display. Realistic sounds in the cockpit are required, as well as a number of special motion and visual effects.

(More detailed information about the qualifications and requirements can be accessed at ELECTRONIC CODE OF FEDERAL REGULATIONS: Title 14 - Aeronautics and Space : PART 60 - FLIGHT SIMULATION TRAINING DEVICE INITIAL AND CONTINUING QUALIFICATION AND USE at 'Appendix A to Part 60 - Qualification Performance Standards for Airplane Full Flight Simulators' & 'Appendix B to Part 60-Qualification Performance Standards for Airplane Flight Training Devices'.)

2.4 Flight Simulator Instruments

The TRC 472, Cessna 172 Cockpit Panel, as can be seen in Figure 2.8, has a lot of Instruments, from various gauges to controls, to navigation instruments. In this chapter, More detailed look of the instruments that are functional in the simulator is to be explored to explain what are their functionalities in the simulation which in turn parallels their real functionalities.

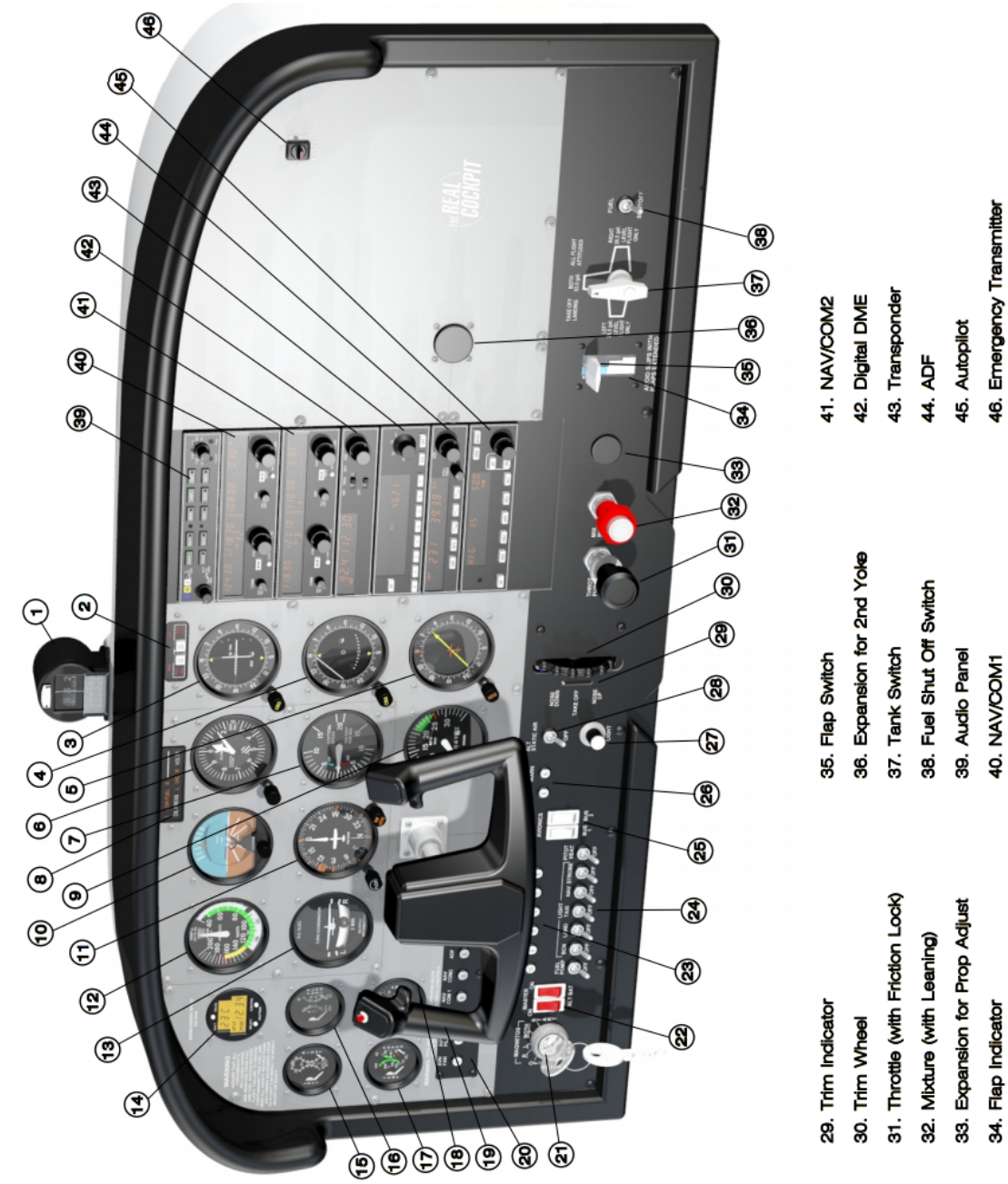


Figure 2.8: Cessna 172 Instruments

2.4.1 Cockpit Gauges Instruments

These are the gauges that will feed the status of the plane to the pilot.

Wet Compass -1

The wet or liquid compass is a compass in which the magnetized needle is damped by fluid.

Unit: degrees

Range: 0 to 36 degrees(*10)

Scale: 5 degrees

Figure 2.9: Wet Compass

Airspeed Indicator -12

The airspeed indicator is an instrument used in an aircraft to display the craft's airspeed, typically in knots, to the pilot.

Unit: knots

Range: 40 to 200

Scale: 10 knots

Figure 2.10: Airspeed Indicator

Tachometer -9

A tachometer is an instrument measuring the rotation speed of a shaft or disk, as in a motor or other machine.The device usually displays the revolutions per minute (RPM).

Unit: RPM (Rotation per Minute)

Range: 0 to 40 (x100)

Scale: 1 RPM (x100)

Figure 2.11: Tachometer

Vertical Speed Indicator -7

A vertical speed indicator is the flight instrument that informs the pilot of the near instantaneous rate of descent or climb.

Unit: feet per minute

Range: -20 to 20 (*100)

Scale: 1 Feet per Min (x100)

Figure 2.12: Vertical Speed Indicator

Heading Indicator -11

A flight instrument used in an aircraft to inform the pilot of the aircraft's heading. The magnetic compass faces errors and reads incorrectly whenever the aircraft is in a bank, or during acceleration, making it difficult to use in any flight condition other than perfectly straight and level; thus the heading indicator is used for maneuvering the plane as the gyroscopic heading indicator is unaffected by dip and acceleration errors.

Unit: degrees

Range: 0 to 36 degrees(*10)

Scale: 5 degrees

Figure 2.13: Heading Indicator

Attitude Indicator -10

An instrument that informs the pilot of the orientation of the aircraft relative to Earth's horizon. It indicates pitch (fore and aft tilt) and bank or roll (side to side tilt).

Unit: degrees

Range & Scale: The outer ring shows increasing bank angles of 10, 20, 30, 60 and 90 degrees, while the two white diagonal lines show bank angles of 15 and 45 degrees. The central scale indicates nose pitch above and below the horizon in 5 degree increments.

Figure 2.14: Attitude Indicator

Turn and Bank Indicator -13

This Instrument has two parts - a rate-of-turn indicator (the miniature airplane) that displays the rate the aircraft heading is changing. The turn coordinator only indicates rate of turn, not bank angle. the inclinometer, the ball in the tube, acts as the bank Indicator displaying sideslip indicating of whether the aircraft is slipping, skidding or in balanced flight.

Rang & Scale: The 2 MIN refers to the rate-of-turn indicator portion. When the wing tip of the miniature airplane is aligned with the tick mark in a coordinated turn (ball centered), a 360 degree turn will take 2 minutes.

Inclinometer Range: -10 to 10 degrees

Figure 2.15: Turn Coordinator

Fuel Indicator Left & Right -15

Indicates the fuel quantity left in the left and right tanks.

Unit: Gallons Quantity

Range: 0 to 26 (left & right)

Scale: 5 Gallons

Figure 2.16: Fuel Indicator

EGT/Fuel Flow Indicator -16

This indicator has 2 gauges. An exhaust gas temperature gauge (EGT gauge) is a meter used to monitor the exhaust gas temperature of an internal combustion engine in conjunction with a thermocouple-type pyrometer. And the other is the fuel flow indicator which indicates the fuel flow of the engine in the plane.

EGT Unit, range and Scale: a series of unlabeled tick-marks spaced 25 degree Fahreneit apart

Fuel Flow unit: Gallon per hour

Range: 0 to 19

Scale: 1 GAL/HR

Figure 2.17: EGT/Fuel Flow Indicator

VOR 2 Indicator -4

This gauge can work as VOR, VHF Omnidirectional Range navigation, a type of short-range radio navigation system for aircraft, enabling aircraft with a receiving unit to determine their position and stay on course by receiving radio signals transmitted by a network of fixed ground radio beacons.

Figure 2.18: VOR 2 Indicator

The VOR display has four elements:

I. Rotating Course Card

Calibrated from 0 to 360, which indicates the VOR bearing chosen as the reference to fly TO or FROM. Here, the 345 radial has been set into the display. This VOR gauge also digitally displays the VOR bearing, which simplifies setting the desired navigation track.

II. OBS knob

The Omni Bearing Selector, used to manually rotate the course card.

III. The CDI

Course Deviation Indicator. This needle swings left or right indicating the direction to turn to return to course. When the needle is to the left, turn left and when the needle is to the right, turn right, When centered, the aircraft is on course. Each dot in the arc under the needle represents a 2 degrees deviation from the desired course. This needle is more-frequently called the left-right needle, with the CDI term quickly forgotten after taking the FAA written exams. Here, the pilot is doing well, and is dead-on courseor maybe lazy and with the autopilot activated in the "NAV" mode.

IV. The TO-FROM indicator

The TO-FROM indicator. This arrow will point up, or towards the nose of the aircraft, when flying TO the VOR station. The arrow reverses direction, points downward, when flying away FROM the VOR station. A red flag replaces these TO-FROM arrows when the VOR is beyond reception range, has not been properly tuned in, or the VOR receiver is turned off. Similarly, the flag appears if the VOR station itself is inoperative, or down for maintenance. Here, the aircraft is flying TO the station.

VOR 1/ILS Indicator -3

This gauge can work as either VOR ,VHF Omnidirectional Range navigation, or ILS, Instrument Landing System, depending on the frequency set in NAV/COM1.

It has all the four elements mentioned in VOR 2 to be used as a VOR gauge. It can also be used as an ILS representing Localizer signal and Glide scope signal The ILS reacts differently from a VOR in several ways:

Figure 2.19: VOR 1/ILS Indicator

I. The localizer course needle is four times as sensitive as a VOR needle. Heading adjustments must be much smaller because of the increased sensitivity of the indicator. For VOR work, each dot under the needle represents 2 degrees deviation from course while for the localizer each dot under the needle represents 0.5 degrees deviation from course.

II. Because the localizer provides information for only one radial, the runway heading, the Nav. receiver automatically cuts out the OBS, the Omni Bearing Selector knob. Rotating the OBS still rotates the course ring on the instrument, but has no affect on the needle.

III. It also has a horizontal needle that represents Glide Slope which is the signal that provides vertical guidance to the aircraft during the ILS approach. The standard glide-slope path is 3 downhill to the approach-end of the runway. The glidescope signal has a full scale of 1.4 degrees, or plus or minus 0.7 degrees from the centerline.

ADF -5

ADF or Automatic Direction Finder is used to detect and follow NDB's or non-directional (radio) beacons. The needle of the gauge always points to the navigation beacon. There is also a Rotatable Compass Card which the pilot can rotate with the knob to display the aircrafts magnetic heading read from the compass, and then the ADF needle will directly indicate the magnetic bearing to the NDB.

Figure 2.20: ADF

Altimeter -6

An altimeter or an altitude meter is an instrument used to measure the altitude of the plane. Altitude can be determined based on the measurement of atmospheric pressure. The greater the altitude the lower the pressure. When a barometer is supplied with a nonlinear calibration so as to indicate altitude, the instrument is called a pressure altimeter or barometric altimeter.

Figure 2.21: Altimeter

Unit: feet

Range: 0 to 99,999 foot.

Scale: The Altimeter has 3 indicators: a 100-feet pointer, a 1000-feet pointer and a 10,000-feet indicator disc.

Oil Temperature and Pressure Indicator -17

This indicator has 2 gauges. First, the oil pressure gauge provides a direct indication of how the oil system is operating. It ensures the pressure in pounds per square inch (psi) of the oil supplied to the engine with the green range indicating the normal operating range. Second, The oil temperature gauge measures the temperature of oil.

Oil pressure Unit: pound per square inch

Range: 0 to 115 psi

Oil temp. Unit: Fahrenheit degrees

Range: 75 to 245

Figure 2.22: Oil Temp. & Pressure Ind.

Suction Guage and Ammeter -18

The suction gage is calibrated in inches of mercury and indicates suction available for operation of the attitude and directional indicators. The desired suction range is 4.5 to 5.4 inches of mercury.

The ammeter shows if the alternator/-generator is producing an adequate supply of electrical power. It also indicates whether or not the battery is receiving an electrical charge. When the pointer of the ammeter is on the plus side, it shows the charging rate of the battery. A minus indication means more current is being drawn from the battery than is being replaced.

Figure 2.23: Suction Guage/Ammeter

2.4.2 Radio Stack Instruments

The radio stack includes a set of radios that are used for navigation and communication while flight. The radio receivers that are functional for the simulation here are:

KX155A NAV1/COM1 -40

Figure 2.24: KX155A NAV1/COM1

The NAV1 is used to set the frequency for the VOR1 gauge. depending on the frequency set(ending in a odd or even number), The VOR1 gauge either works as an ILS (instrument landing system) or a VOR (VHF Omni Directional Radio Range) gauge. The frequency wanted can be input through a 2-piece knob. It allows you to switch between pre-selected standby and active frequencies with the touch of a button.

Unit: MHz

Range: 108.00 to 117.95

Scale: Outer Knob:1 MHz increments; Inner Knob: pushed in: 0.1 MHz increments, pulled out: 0.05 MHz increments

KX155A NAV2/COM2 -41

Figure 2.25: KX155A NAV2/COM2

NAV2 has the same functionality and look as NAV1 but instead it is used to set the frequency for the VOR2 gauge.

Unit: MHz

Range: 108.00 to 117.95

Scale: Outer Knob:1 MHz increments; Inner Knob: pushed in: 0.1 MHz increments, pulled out: 0.05 MHz increments

KN62A DME -42

The DME reciever shows the slant range distance from origin of the radio signal in nautical mile. It can be channeled remotely through the NAV2 receiver or tuned directly with its own frequency selection knob.

Figure 2.26: KN62A DME

Unit: MHz

Range: 108.00 to 117.95

Scale: Outer Knob:1 MHz increments; Inner Knob: pushed in: 0.1 MHz increments, pulled out: 0.05 MHz increments

KR87 ADF -44

The ADF reciever gives you accurate bearing-to-station that shows up on the ADF gauge. It allows you to switch between pre-selected standby and active frequencies with the touch of a button.

Unit: kHz

Range: 199 to 1799

Scale: Outer Knob: 100 kHz increments; Inner Knob: pushed in: 1 kHz increments, pulled out: 10 kHz increments

Figure 2.27: KR87 ADF

2.4.3 Cockpit Controls Instruments

These are the controls that the pilot will use to control plane on ground and in air.

Yoke -19

The yoke is used to control the attitude of the plane in both pitch (up and down) and roll (left and right). The Yoke can turn left and right 90 degrees for a total of 180 degrees! (Most commercially available yokes are only capable of turning 90 degrees.) The maximum movement in- and out is 170 mm. (Most commercial available yokes can move in and out only 100 mm. maximum, while some do even only 70 mm.)

Figure 2.28: Yoke

Rudder Pedals

The rudder pedals are used to drive the rudder controlling the yaw of the plane. The pedals are also used for taxiing (steering the plane on ground). The pedals are also used for differential braking (separate brakes on both pedals) by pushing on the top of one pedal which would apply differential brake on one wheel causing a differential turn.

Figure 2.29: Rudder Pedals

Flaps Switch -35

Used to move the flaps of the plane. The Flaps Switch has 4 positions: Flaps Up, Flaps 10 degrees, Flaps 20 degrees and Flaps 30 degrees. The indicator is servo driven and indicates when the Flaps have reached their position.

Figure 2.30: Flaps Switch

Throttle Lever -31

The throttle lever is used to control the thrust output of the aircraft's engine and thus controlling airspeed.

Figure 2.31: Throttle Lever

Trim Wheels -30

Used to move the nose of the plane up and down. Moving it upwards changes the 'hands-off' elevator position to a more nose-down position; moving it downwards does the reverse.

Figure 2.32: Trim Wheel

Chapter 3

System Requirements

In this chapter, the requirements of the system will be discussed and broadened on in multiple sections ranging in complexity and details of the requirements.

3.1 Informal Requirements

The system requirements at hand are to develop a cockpit resembling the Cessna 172 cockpit to be driven by the professionally used and industry standard Presagis FlightSIM software simulation through a distributed system, meaning to have all the components of the cockpit run simultaneously on one computer and to interact through LAN with the FlightSIM software application running on another computer. These components include devices that work as inputs by user, devices that work as output for user and devices that do both inputs and outputs.

The system being developed is a system that will be used by a person who expects to be using a flight simulator that resembles the real cockpit in both controls and instruments gives the requirement for the system to resemble the shape and the feel of

the real cockpit which the TRC472 cockpit used here does perfectly. The environment and physics should be encompassing, realistic and manipulative for different scenarios which Presagis FlightSIM is more than capable of. It is also required that the controls and the instruments in the simulator will parallel the actual functionality of the original devices to be found in the real plane, meaning they should be as responsive and accurate as their respective counterparts are as possible and to control and display information in the same manner done in the actual cockpit. The system here also being user driven requires the visual aspect of the simulator to react to the controls in the cockpit without lagging or delays or else it would not parallel the real experience and also requires the visual environment simulation to be running smoothly with no frame drops or skips more like a high quality gaming experience.

The system at hand should be stable without crashes, or major problems occurring through the simulation process. The system should also be flexible and forward looking so that it can change in accordance to any future needs or additions.

3.2 Functional Requirements

- Map the virtual instrument output from FlightSIM to its equivalent real USB instrument.
- Map the hardware control input to its virtual control input in FlightSIM.
- The system should run smoothly at 30 Hz or higher.
- The system should work on a distributed computers system or a single computer.

3.3 Non-functional Requirements

The non-functional requirements for this system are:

Integrability The system should be easily implementable with other additions

Maintainability The system should be easily maintainable. So, if any thing breaks, the whole system shouldn't be broken and it should be easy to fix.

Robustness The system should be able to cope with errors during execution such as missing hardware instruments.

Stability Meaning the various system parts will be stable over time and will not need changes.

3.4 Instruments Types

The instruments can be categorized in 4 types:

Simple Gauges These are gauges that will display information needed by the pilot that are simple to use, read and simple to interface and do not require manual user input such as Airspeed, Vertical speed, etc.

Complex Gauges These include the gauges that will tell the pilot about the plane position and navigtion details that are a bit complicated to use and interface often require user input such as VOR, ADF, etc.

Simple Controls These are the devices in which the user is going to drive the plane motion such as Yoke, Rudder Pedals, etc. They only read values from user input

and feed it to the computer without displaying any hardware info or output to the user.

Complex Controls Theese are the controls that read user input and display data to the use as well. Such devices are the radio-stack and the flap switch as they take inputs and display hardware outputs as well.

This simple module diagram shows how the different kind of instruments should interact with the computer and the user.

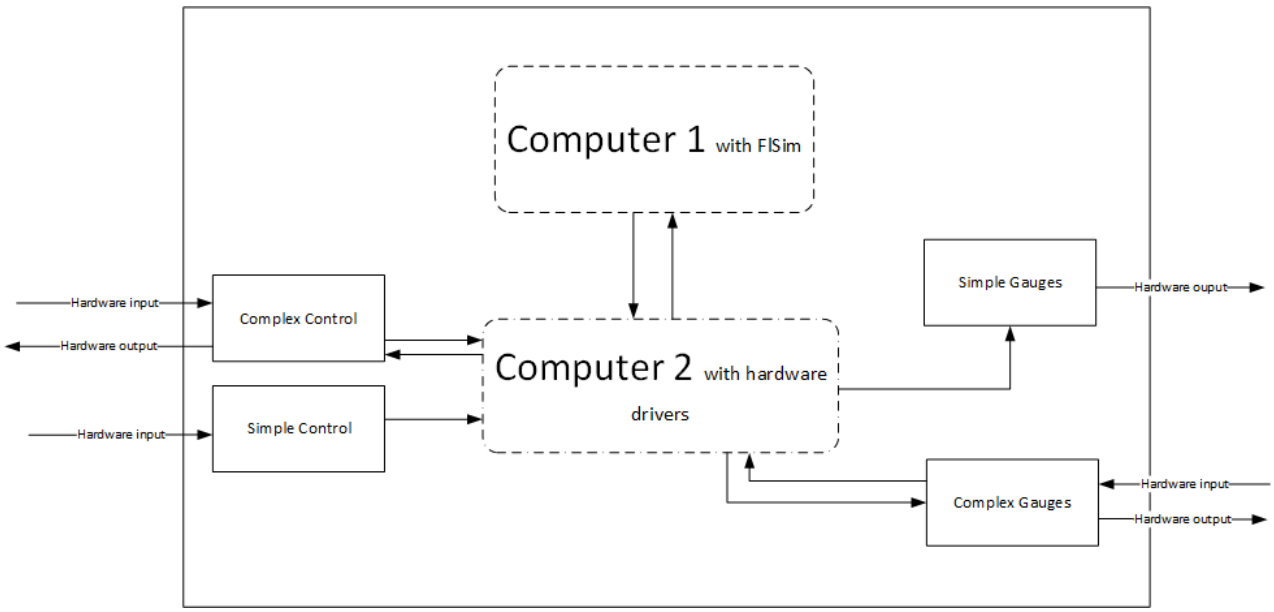


Figure 3.1: simple module decompsition

Chapter 4

System Design

In this chapter, the design of the system, guided by the requirements mentioned in the previous chapter, will be broadened on and the design objectives, choices, and process will be explained.

4.1 Design Objective

The objective of the design was to have all the components of the cockpit to run simultaneously on a computer and to interact through LAN with the FlightSIM software application running on the other computer.

4.2 Design Choices

In this section, some of the choices made will be explained as the reason behind them and their advantages will be explored.

4.2.1 Distributed Computing

One of the first choices made as mentioned before was that the simulator will be running on 2 different computers; one is responsible for the cockpit and the hardware and the other is responsible for the visual environment. This is called distributed computing. Having a distributed system on 2 computers has benefits for the project at hand. First, a design over a distributed system will make it easier to integrate the system with the already existing systems in the motion simulator. Second, it will also make it easier for any future expansions or additions of software or hardware that can be connected in the same manner through the distributed system. Third, a distributed system is also more reliable as there is no single point of failure that will stop the whole system from working.

The distributed computing model that will be used here follows the client-server model which is a software engineering technique often used within distributed computing that allows two independent processes to exchange information, through a dedicated connection. When a client/server connection is made, information may flow both ways through the virtual channel that connects the two software processes. The terms client and server applies to roles that the software components assume during a single connection session. One of the software processes adopts the role of server and waits for connections to be made. The other process adopts the role of client and makes the connection with the server (Rider, 2004).

4.2.2 UDP Connetion

The communication between the 2 computers through LAN was decided to be through UDP rather than TCP. UDP is short for User Datagram Protocol; TCP is short for

Transmission Control Protocol. They are both messaging protocols that differ quite a bit. UDP is suitable for applications that require fast, and efficient transmission; hence, it is a better choice for data communication in this project. First, UDP is connectionless meaning that it doesn't restrict you to connection based communication model as no handshaking is needed, information is sent in one direction from source to destination without verifying the readiness or state of the receiver and it is possible to have multiple sources for messages. Second, it is also much faster that TCP as it is lightweight and there is no ordering of messages, no tracking connections, etc. UDP is a small transport layer designed on top of IP. Packets are sent individually and are checked for integrity only if they arrive. Packets have definite boundaries which are honored upon receipt, meaning a read operation at the receiver socket will yield an entire message as it was originally sent. If one packet is damaged or lost, you don't want to wait on the stream protocol (usually TCP) to issue a re-send request – you need to recover quickly. TCP can take up to some number of minutes to recover, and for real-time protocols this is unacceptable. Using a datagram protocol like UDP allows the software to recover from such an event extremely quickly, by simply ignoring the lost data or re-requesting it sooner than TCP would. These lower delays makes UDP an appealing choice for delay-sensitive applications (Thomas, 2014).

4.2.3 Object Oriented Design

Another design choice was for most of the code to be designed in an object oriented manner in C++. The behavior of an object is defined by the class, which works as the blueprint of the object instantiated. An object must be explicitly created based on a class thus is considered to be an instance of that class. OOP or Object Oriented

Programming applications have several advantages such they are easier to maintain, have more reusable components, and are more scalable, to name a few.
Since modularization is a big issue when designing such a big system that has several components and can be expanded on in the future, the system needed to be built in a way that having several devices and adding devices in wouldn't be such a hassle. So, the Abstract Factory design pattern is used. The Factory Design Pattern is useful in a situation that requires the creation of many different types of objects, all derived from a common base type.
Using the Abstract Factory pattern is useful when:

- a system should be independent of how its products are created, composed, and represented.
- a family of related product objects is designed to be used together, and you need to enforce this constraint.
- a class library of products is provided, but no need to reveal their implementations, just their interfaces (Gamma *et al.*, 2009).

4.2.4 Object Serialization

As said in the previous sections, the design is object oriented and is distributed over 2 computers, thus it lead to another design choice where object serialization was to be used. Object serialization is defined as the process of translating an object state into a stream of data so that it can be stored in a file or memory buffer, or transmitted across a network and reconstructed later in the same or another computer environment to it's state.

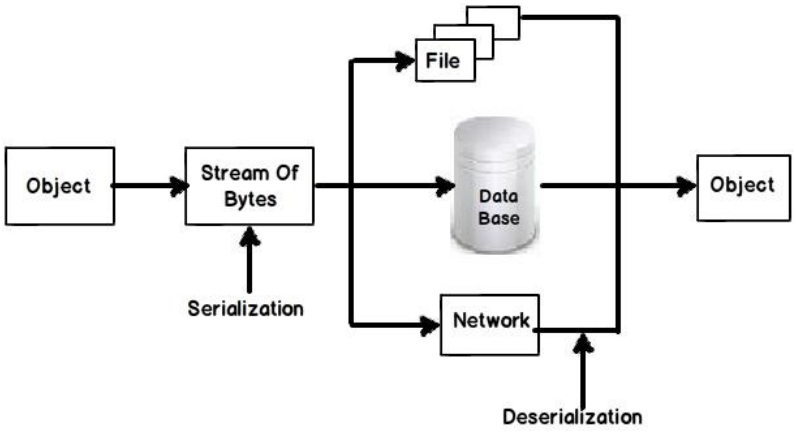


Figure 4.1: Object Serialization

This stream of data needs to be in a format that can be understood by both ends of a communication channel so that the object can be serialized and reconstructed easily. Serialization of an object can also be called marshalling an object. The following are the basic advantages of serialization:

- facilitate the transportation of an object through a network.
- create a clone of an object (Kanjilal, 2006).

4.2.5 Multithreading

To make the system faster and to be able to use various instruments at the same time multi-threading was to be used in some instances. This was done as some instruments require longer time to write read from than others do, so running them on different threads was a better option for speed and stability. A multithreaded program has the ability to manage multiple threads existing within the context of a single process that can share the process's resources, but are able to execute independently.

4.2.6 Concurrency

When dealing with multi-threads that are sharing the same resource, concurrency can become an issue. Race conditions can occur between a writing thread and a reading thread. These problems were avoided using mutexes and atomic variables. A mutex is a lockable object that is designed to signal when a thread needs exclusive access, preventing other threads from access to the same memory locations. The mutexes were used in this project for example so the processing threads would not start unless there is data that can be processed. So, the shared resource, in this case the message, is locked by a mutex by the writing thread until it starts receiving data. Once data is being received, the mutex is unlocked that unblocks all threads currently blocked so they can start processing the received messages.

4.2.7 Atomic Variables

Atomic Variables were also used to also eliminate any read-write races. C++11 concurrency library introduces atomic types as a template class: std::atomic. The library can be used to instantiate variables and the operation on that variable will be atomic and hence thread-safe(Xia, 2012).

If an operation is atomic (i.e indivisible such as an atom), no other thread can modify or read any partial results during the operation on the object eliminating race conditions. Each instantiation and full specialization of the std::atomic template defines an atomic type. Objects of atomic types are the only C++ objects that are free from data races (cppreference.com, 2013). This can allow for lock-free programming so the threads don't need any locks or mutex for a simultaneous read and write.

4.3 Context Diagram

The following page includes a draft top level context diagram showing the various hardware inputs (user input variables) and hardware outputs (controlled variables) of the flight simulator System:

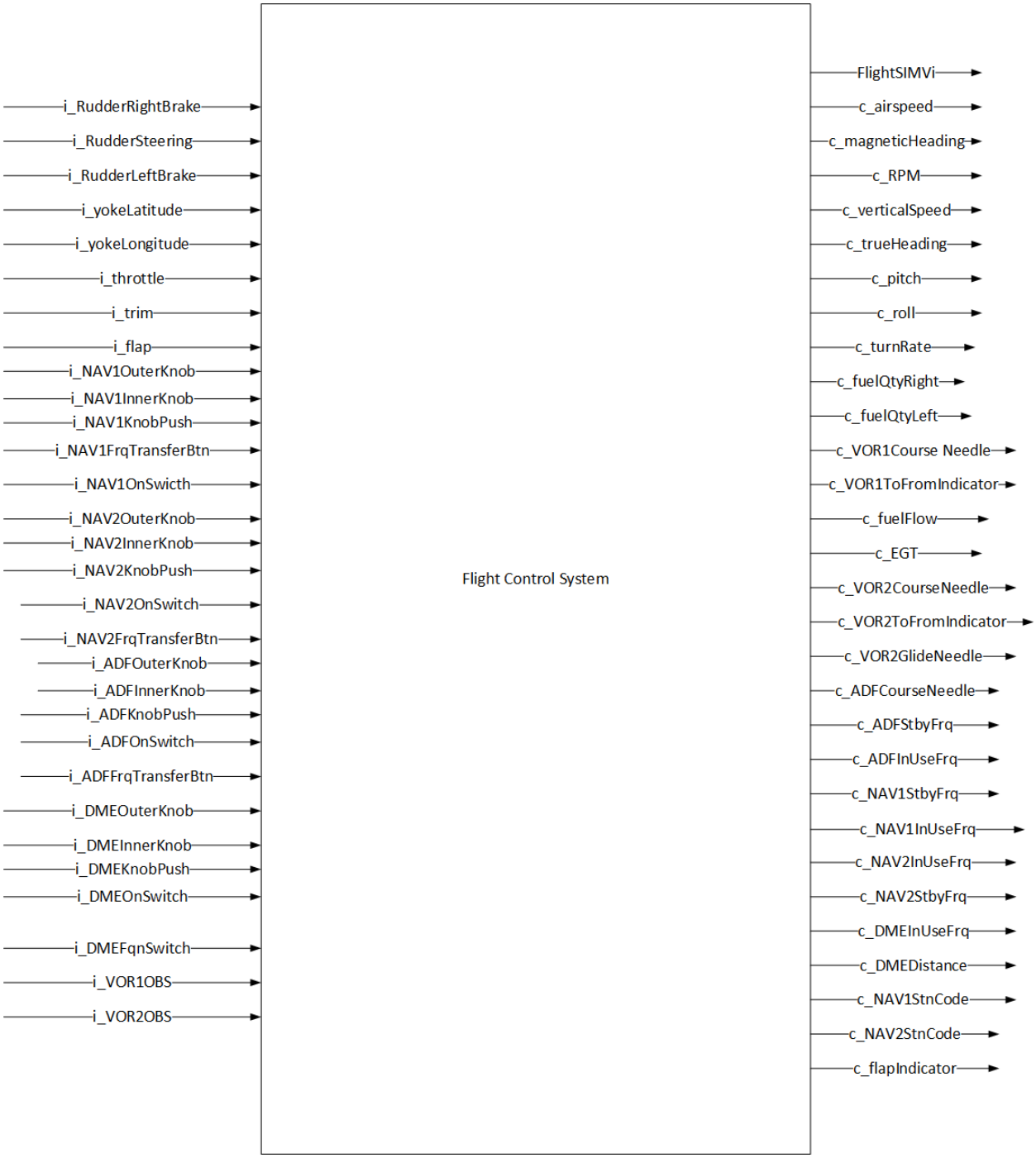


Figure 4.2: Draft Context Diagram

4.3.1 Input variables

The following table explains the input variables; these are all of the variables the pilot inputs in the system through interacting with the controls. Once these values are read, they're calibrated

Table 4.1 Input Variables Table

Input Variable	Description
i_rudderRightBrake	The value read for pressing the rudder right brake
i_rudderSteering	The value read for moving the rudders for steering
i_rudderLeftBrake	The value read for pressing the rudder left brake
i_yokeUpDown	The value read from moving the yoke in the up/down direction
i_yokeRightLeft	The value read from moving the yoke in the right/left direction
i_throttle	The value read from moving the throttle lever
i_trim	The value read from moving the trim wheel
i_flap	The value read from moving the flap switch
i_NAV1OuterKnob	The value read from rotating the NAV1 outer knob
i_NAV1InnerKnob	The value read from rotating the NAV1 inner knob
i_NAV1KnobPush	The value read from pushing in or pulling out the NAV1 inner knob
i_NAV1FrqTransferBtn	The value read from pressing the NAV1 frequency transfer button
i_NAV1OnSwitch	The value read from rotating the NAV1 On knob
i_NAV2OuterKnob	The value read from rotating the NAV2 outer knob
i_NAV2InnerKnob	The value read from rotating the NAV2 inner knob
i_NAV2KnobPush	The value read from pushing in or pulling out the NAV2 inner knob
i_NAV2FrqTransferBtn	The value read from pressing the NAV2 frequency transfer button

Table 4.1 Continued from previous page

Input Variable	Description
i_NAV2OnSwitch	The value read from rotating the NAV1 On knob
i_ADFOuterKnob	The value read from rotating the ADF outer knob
i_ADFInnerKnob	The value read from rotating the ADF inner knob
i_ADFKnobPush	The value read from pushing in or pulling out the ADF inner knob
i_ADFOnSwitch	The value read from rotating the ADF On knob
i_DMEOuterKnob	The value read from rotating the DME outer knob
i_DMEInnerKnob	The value read from rotating the DME inner knob
i_DMEKnobPush	The value read from pushing in or pulling out the DME inner knob
i_DMEOnSwitch	The value read from rotating the DME On knob
i_DMEFqnSwitch	The value read from pressing the DME frequency transfer button
i_VOR1OBS	The value read from rotating the VOR1 OBS knob
i_VOR2OBS	The value read from rotating the VOR2 OBS knob

Table 4.1: Input Variables Table

4.3.2 Controlled variables

The following table explains the output(controlled) variables. These are the variables that are to be displayed on the cockpit various pieces of hardware

Table 4.2 Output Variables Table

Output Variable	Description
c_airspeed	The airspeed indicated to the user on the airspeed gauge

Table 4.2 Continued from previous page

Output Variable	Description
c_magneticHeading	The magnetic heading displayed on the wet compass
c_RPM	The rotations per minute value displayed on the RPM gauge
c_verticalSpeed	The vertical speed value displayed on the vertical speed gauge
c_trueHeading	The heading value indicated on the true heading gauge
c_pitch	The pitch value that's indicated on the attitude indicator
c_roll	The roll value that's indicated on the attitude indicator
c_yaw	The yaw value that's indicated on the Inclinometer
c_turnRate	The turn rate displayed on the turn rate indicator
c_fuelQtyRight	The fule Quantity in the right tank indicated in the fuel indicator
c_fuelQtyLeft	The fule Quantity in the left tank indicated in the fuel indicator
c_VOR1CourseNeedle	The angle deviation indicated by the VOR 1 course needle
c_VOR1ToFromIndicator	The to or from direction displayed by the VOR 1
c_VOR2CourseNeedle	The angle deviation indicated by the VOR 2 course needle
c_VOR2ToFromIndicator	The to or from direction displayed by the VOR 2
c_VOR2GlideNeedle	The glide deviation indicated by the VOR2 glide needle
c_EGT	the exhaust gas temperature displayed on the fuel flow Indicator
c_fuelFlow	the fuel flow displayed on the fuel flow Indicator
c_ADFCourseNeedle	The angle deviation indicated by the ADF course needle
c_ADFInUseFrq	The ADF in-use frequency displayed on the ADF radio
c_ADFStbyFrq	The ADF standby frequency displayed on the ADF radio
c_NAV1StbyFrq	The NAV 1 standby frequency displayed on the NAV 1 radio

Table 4.2 Continued from previous page

Output Variable	Description
c_NAV1InUseFrq	The NAV 1 in-use frequency displayed on the NAV 1 radio
c_NAV2StbyFrq	The NAV 2 standby frequency displayed on the NAV 2 radio
c_NAV2InUseFrq	The NAV 2 in-use frequency displayed on the NAV 2 radio
c_DMEStbyFrq	The DME standby frequency displayed on the DME radio
c_DMEInUseFrq	The DME in-use frequency displayed on the DME radio
c_DMEDistance	The distance in nautical miles displayed on the DME radio
c_NAV1StnCode	The IATA airport code displayed on the NAV1 radio
c_NAV2StnCode	The IATA airport code displayed on the NAV2 radio
c_flapIndicatior	The flaps angle indicated by the flaps indicator

Table 4.2: Output Variables Table

4.4 Class Diagrams

The class diagram is the main building block of any object oriented design. It is used both for general conceptual modeling of the systematics of the application, and for detailed modeling translating the models into programming code. The following figure is a simple class diagram that shows the different classes and their hierarchy.

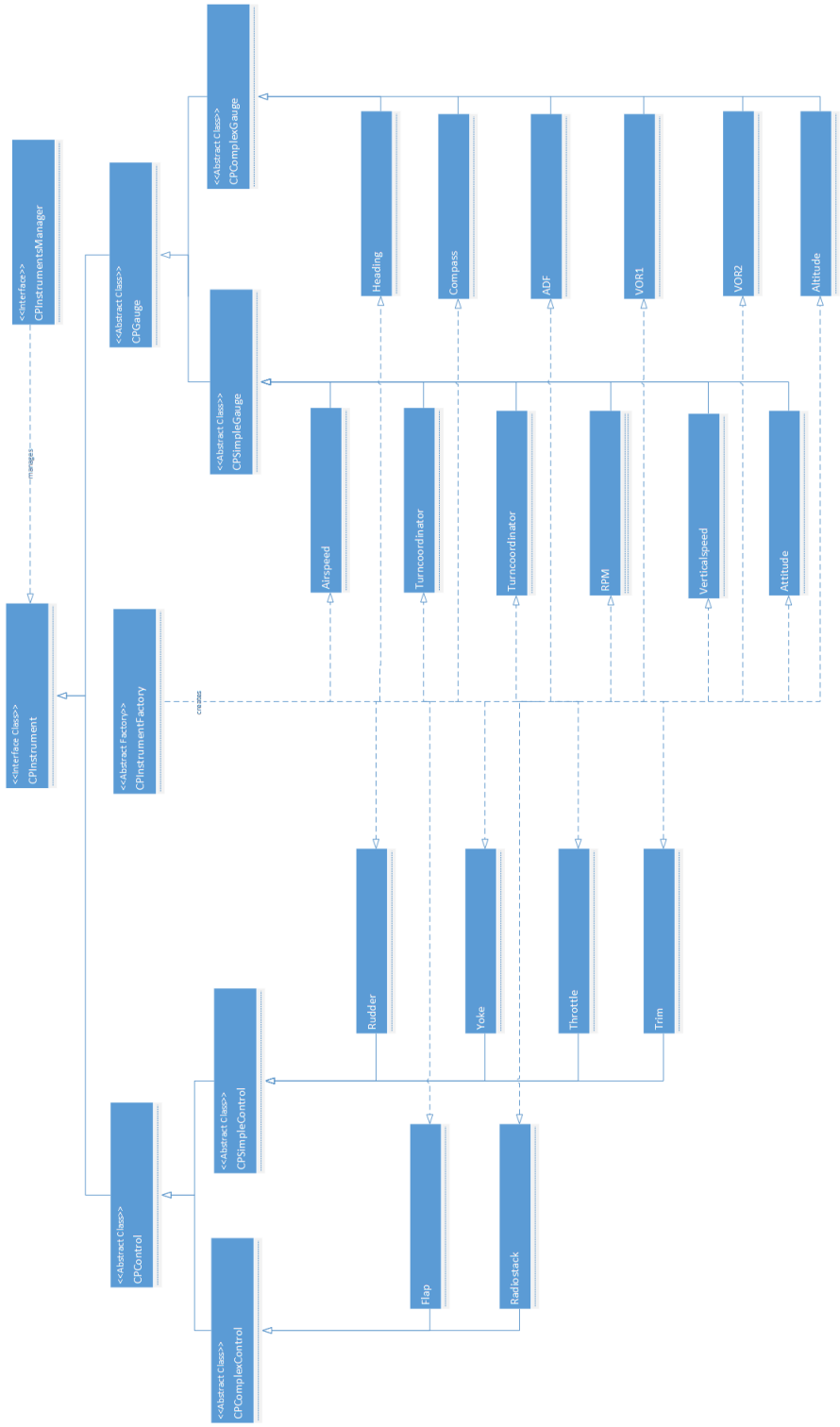


Figure 4.3: Simple Class Diagram

4.5 System Architecture

The following page includes a draft system architecture. It shows the decomposition of hardware and software modules. It also shows the different sets of software modules that run on different systems.

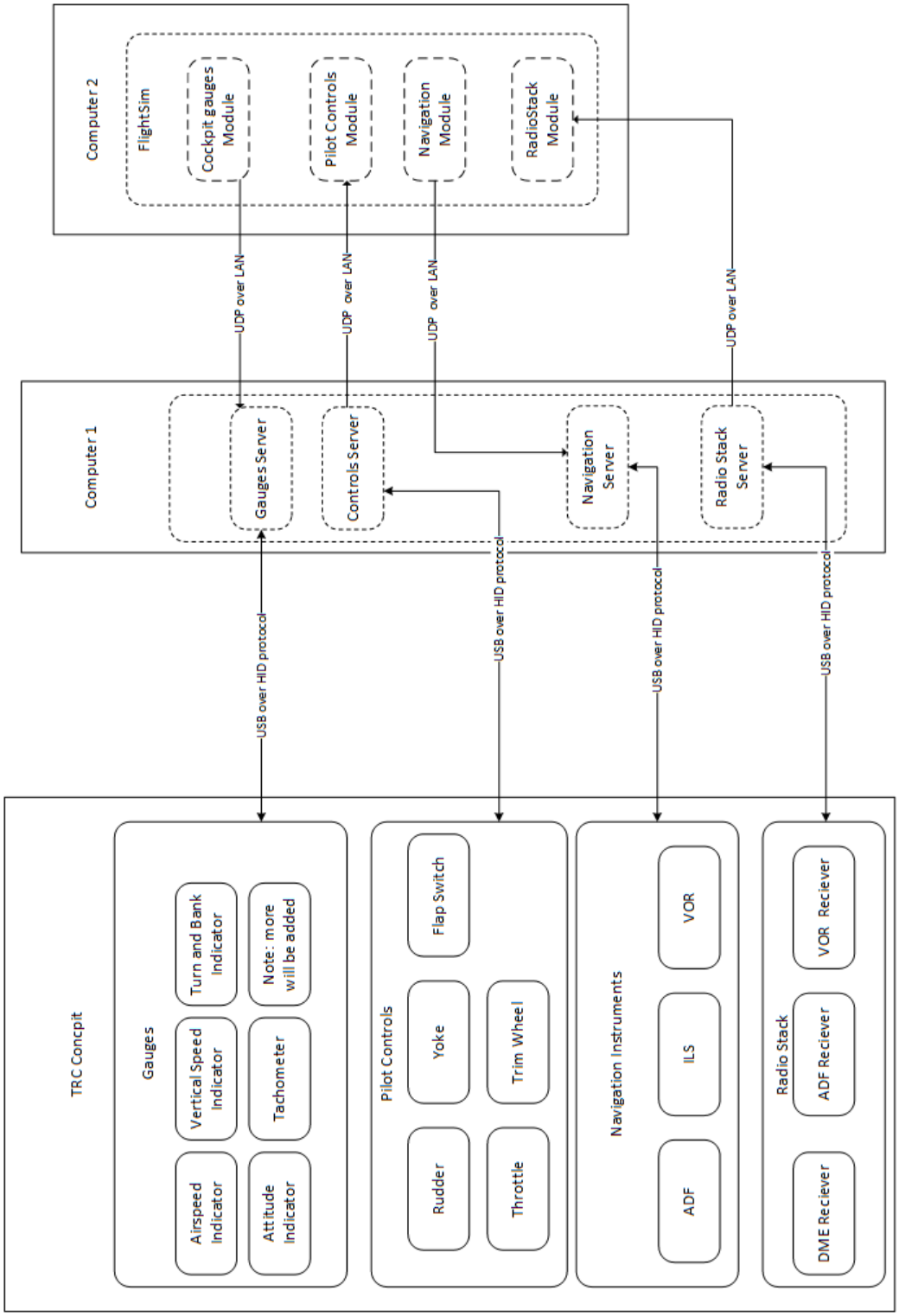


Figure 4.4: Draft Hardware System Architecture

4.6 Server Side

On the server side, there are 4 servers running independently (either as different processes or as threads) and each server being connected over a different UDP socket connection is responsible for a set of inputs or controls. The servers are what will be interacting with the hardware of the cockpit by either reading from controls or writing to gauges and communicating in turn with the client modules implemented in FlightSIM.

Simple Gauges Server is the server responsible for writing the data it gets from its respective client to the general and simpler gauges

Complex Gauges Server is the server responsible for writing the data it gets from its respective client to the navigation and more complex gauges; it also reads inputs from the complex gauges that do have input such as the OBS knob in VOR.

Control Server is the server that will read data from controls and send it to the respective controls client on the FlightSIM side. It will also write data to more complex controls that require it such as the flap switch indicator.

RadioStack Server is the server that will read and write from the radiostack and communicate it respectively with the radiostack client on the FlightSIM side.

4.7 Presagis FlightSIM Integration

4.7.1 User-Developed Modules

Presagis FlightSIM is capable of integration with user written simulation modules. These modules can range from making a in-flight scenario to overwriting a whole system with its inputs and outputs. Using the FlightSIM development libraries, the functionality can be extended to have new features not provided by the FlightSIM simulation models. To do so, additional C/C++ code files must be compiled with the standard simulation.

These user-written files are referred to as user modules. User modules can be added to FlightSIM by including them in a stand-alone shared library; in Linux, this is done using a '.so' file; '.so' files are dynamic libraries that can get loaded to FlightSIM during the runtime.

A user module is used only if it is added to the Aircraft Systems through the systems panel in the Flight Models of the FlightSIM DE as seen in Figure 4.5. The API structures(provided by the development libraries of FlightSIM) can be used to replace a system model or a subsystem model, or to add a new model completely defined by the user.

The user modules to be included here are the clients that will interact with the servers by sending or receiving data from the servers; they will be 4 in total. These 4 clients are to be integrated in FlightSIM as modules each to interact with a server. A client for the pilot controls, a client for the radio stack, a client for the simple gauges, and a client for the complex gauges.

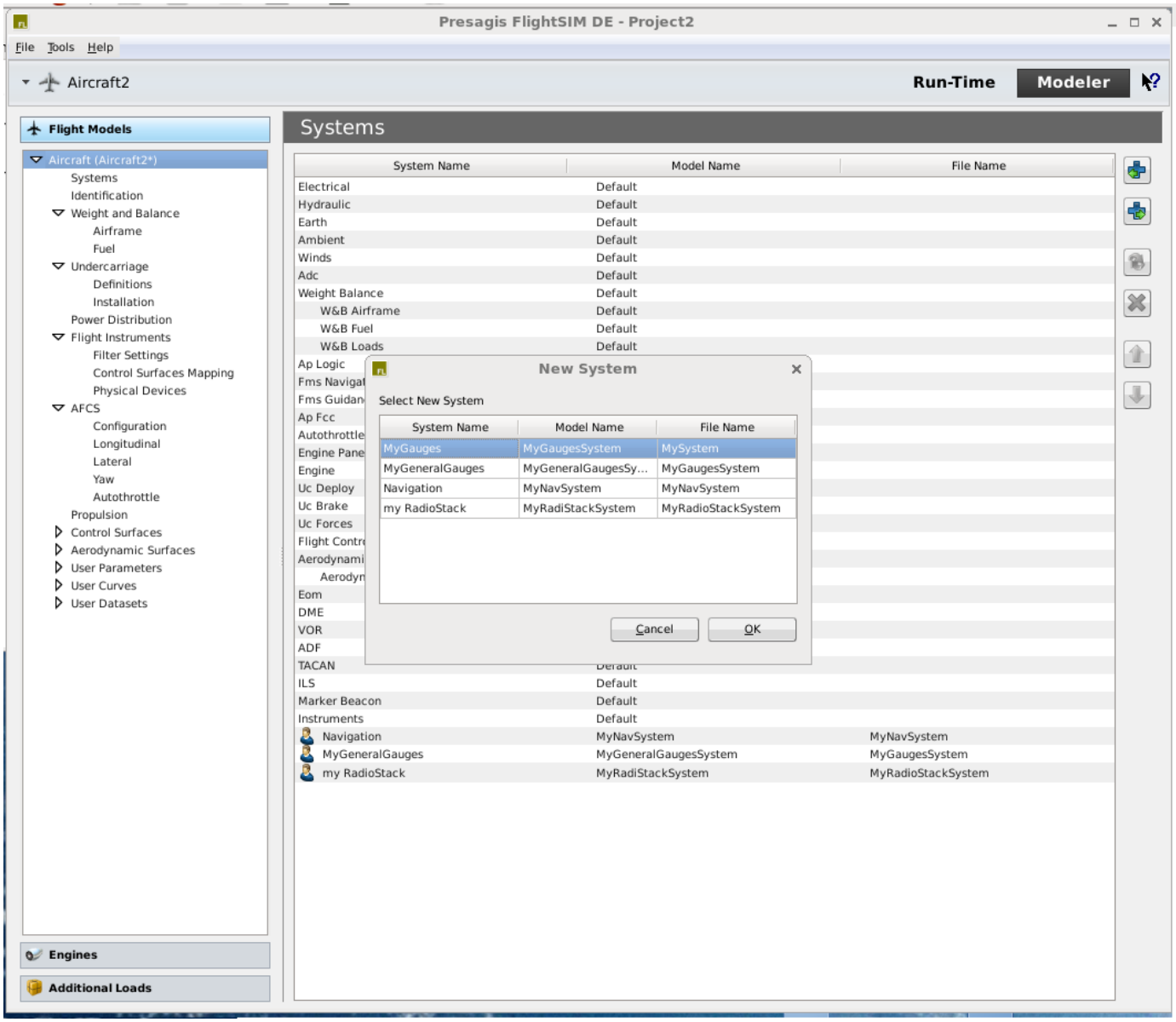


Figure 4.5: Screenshot of adding a user model

4.7.2 Pilot Input Interface

The Pilot Inputs panel enables you to specify the source for each pilot input required to fly the aircraft. It allows to use a custom pilot interface using the API to send

command messages. The development libraries provide you with the capability of adding your own source code to interface with your devices. Once a user module is created using the pilot input API and the shared library is added to FlightSIM it can be chosen through the interface seen in Figure 4.6.

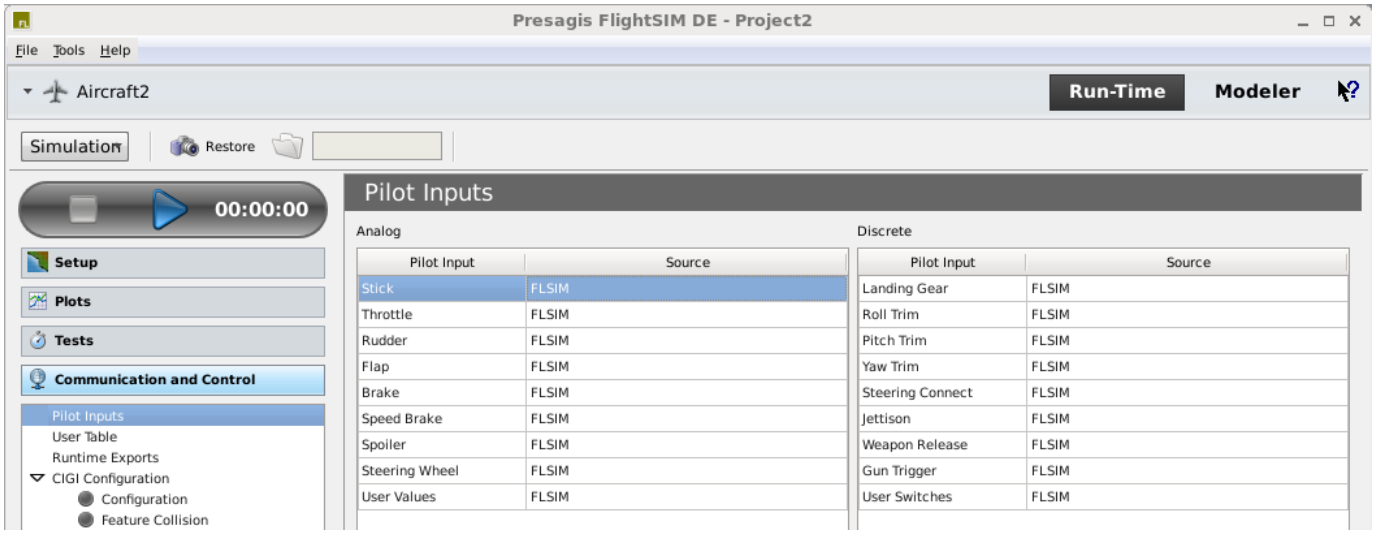


Figure 4.6: Screenshot of pilot input interface

4.7.3 Weight and Balance system

Using the weight and balance system, one can set the plane physics to match the physics of the plane needed to be simulated, in this case the Cessna 172. The weight and balance system inputs can be seen in Figure 4.7 .

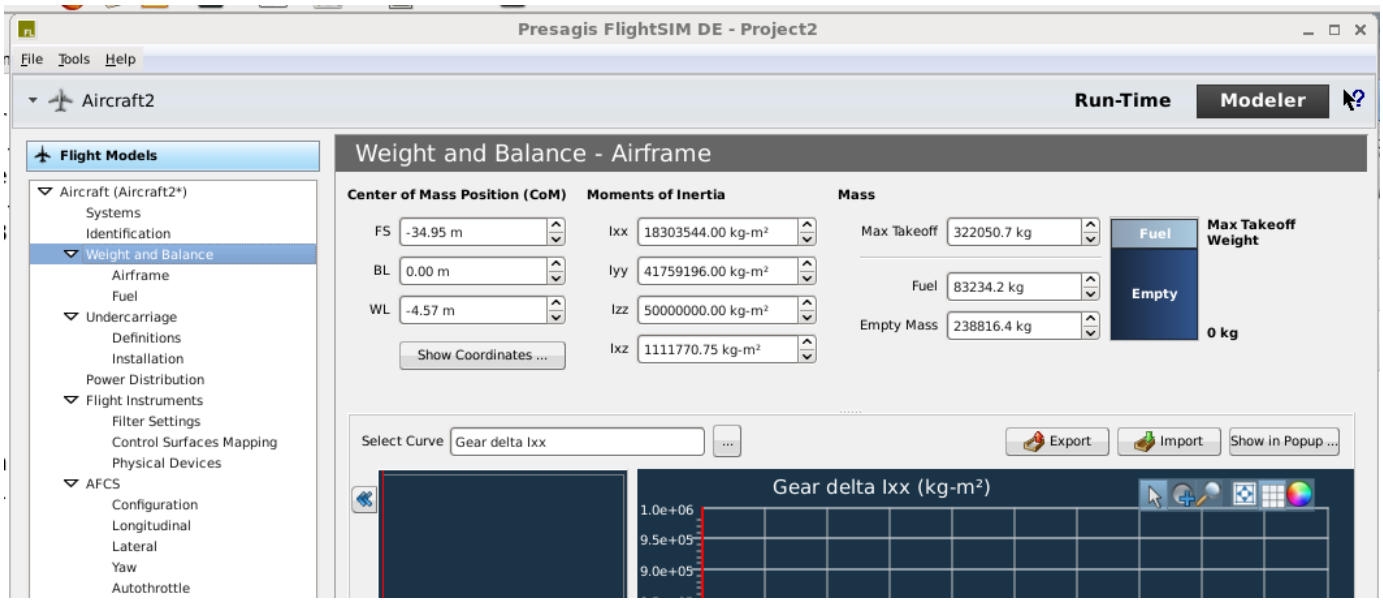


Figure 4.7: Screenshot of weight and balance system

The model assumes that the aircraft can be represented by a combination of three sets of bodies, corresponding to the three sub-systems, with each body having its own mass, center of gravity position, and moments of inertia. These three systems are:

Airframe The default simulation model for the Airframe sub-system, of the Weight and Balance system, assumes that the mass of the airframe is constant and that its center of gravity position and moments of inertia change only with the gear deployment position.

Fuel Load The default simulation model for the Fuel Load sub-system, of the Weight and Balance system, assumes that it can be represented by a combination of two tanks, with each tank having its own mass, center of gravity position, and moments of inertia. The model assumes that the center of gravity position and moments of inertia of each tank are dependent only on the tank fuel mass.

Additional Loads The default simulation model for the Additional Loads subsystem, of the Weight and Balance system, assumes that it can be represented by a combination of bodies where a body is an additional load, with each load having its own mass, center of gravity position, and moments of inertia. The model assumes that these characteristics (mass, center of gravity, moments of inertia) are constant and that they can only be modified through direct commands.

Chapter 5

Implementation and Development

In this chapter, the implementation and development based on the design of the system (discussed in the previous chapter) will be discussed in details showing the approach taken to ensure a good implementation phase.

5.1 Class-based Model

This section will show the class-based model of the instruments. A simple class diagram showing the hierarchy is included in the design chapter. The following diagrams show in more details the class diagram with the methods and variables included.
The diagrams given here samples each including limited number of instruments to be clear ans show the functionality. The full diagram can be found in the appendix.

5.1.1 Gauges sub-Class Diagram

This diagram show in more details the functions and the hierarchy of the Gauges.

Figure 5.1: CPGauges Class Diagram

5.1.2 Controls sub-Class Diagram

This diagram show in more details the functions and the hierarchy of the Controls.

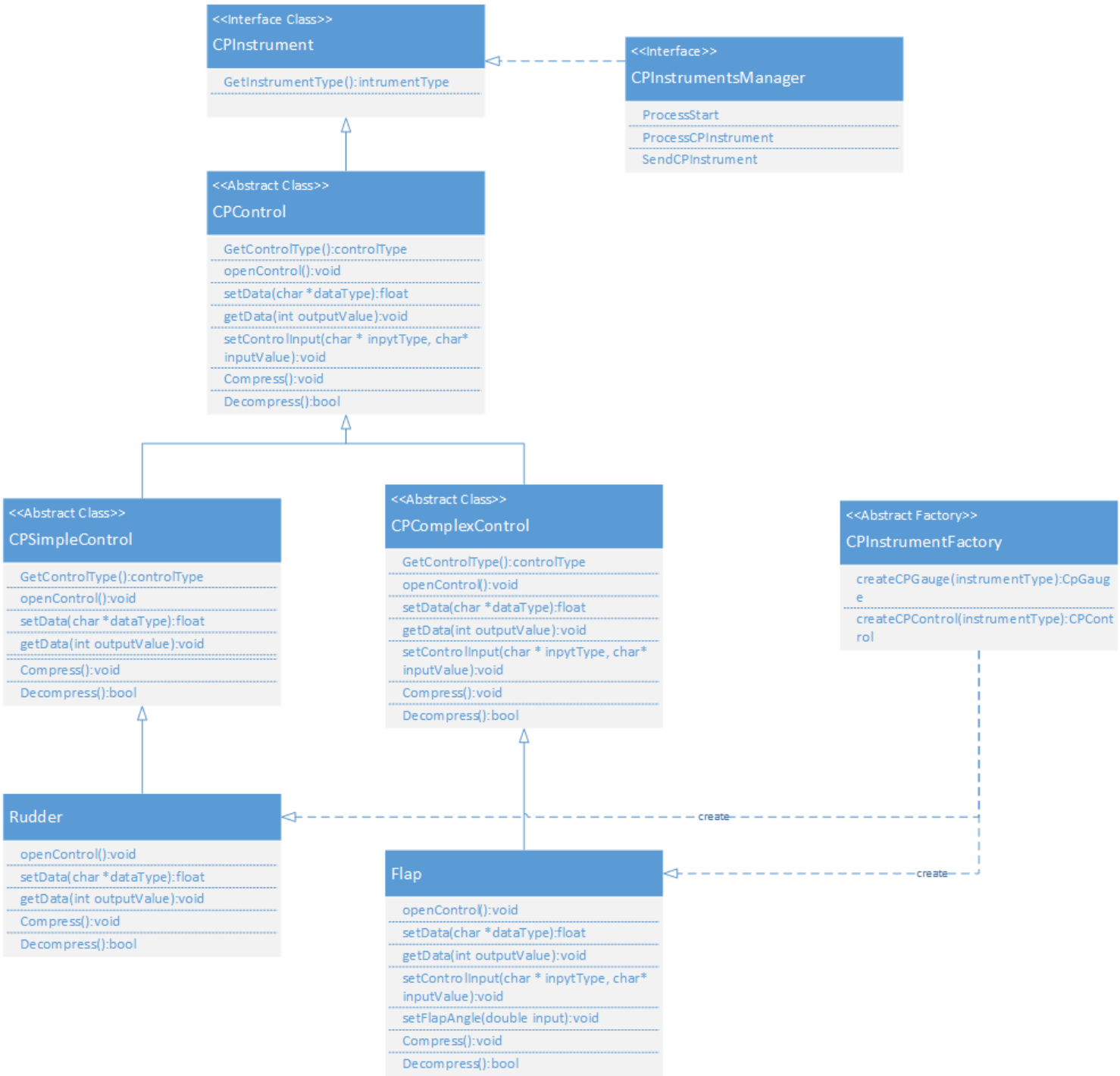


Figure 5.2: CPControl Class Diagram

5.1.3 Cockpit Instruments Factory

The Cockpit Instruments Factory is the one responsible for object creation. It works as an abstract factory(explained in the prevoius chapter) in which the actual concrete type of object to be created is determined by the factory itself. Using the Cockpit Instruments Factory, the server/client code can have no knowledge of the concrete classes available and there's no need to include any of their class declarations headers in the main program, adding a layer of insulation and information hiding.

It has these 2 methods to make the different instruments:

```
CPControl* createCPControl(CPInstrumentType type);
// returns concrete control object and the only method that can
   instantiate a control object
```

```
CPGauge* createCPGauge(CPInstrumentType type);
//returns concrete gauge object and the only method that can
   instantiate a gauge object
```

5.1.4 Cockpit Instruments Manager

The Cockpit Instruments Manager works like a manger of all the instruments. It is responsible for packing an object to sent through network, and on the other hand for processing a message and knowing what data of an object is it and saving it to the right object.

These are the main methods:

```
int SendCPInstrument(CPInstrument* cpcObj,char* buffer);
// will take the object, and serialize it to a stream of bytes
   saving it to the buffer that can be sent as a message. It returns
    the size of the data saved in the buffer.
```

```
void ProcessStart(CPInstrument* cpiObj);
//used to start the process of an object by opening the hardware
    connection to it and telling the system that this is an object
    that the deserialized data can be saved to accordingly.
```

```
int  ProcessCPInstrument(char* data);
//will take the buffer that was received as an input and deserialize
     it and saving the data to the appropriate object
```

5.2 HIDAPI

The API that was used to interact with the HID (Human Interface Devices) usb devices is HIDAPI. HIDAPI is a multi-platform library which allows an application to interface with USB HID-Class devices on Windows, Linux, and Mac OS X. This API is most useful when used with custom or Vendor-Defined HID devices like the ones used here in the cockpit. Each device has its unique Vendor ID (VID) and Product ID (PID); through this information, the API provides functions to interact with HID devices and various methods can be used as seen in Figure 5.3.

Functions

int HID_API_EXPORT HID_API_CALL	**hid_init** (void) Initialize the HIDAPI library.
int HID_API_EXPORT HID_API_CALL	**hid_exit** (void) Finalize the HIDAPI library.
struct **hid_device_info** HID_API_EXPORT *HID_API_CALL	**hid_enumerate** (unsigned short vendor_id, unsigned short product_id) Enumerate the HID Devices.
void HID_API_EXPORT HID_API_CALL	**hid_free_enumeration** (struct **hid_device_info** *devs) Free an enumeration Linked List.
HID_API_EXPORT **hid_device** *HID_API_CALL	**hid_open** (unsigned short vendor_id, unsigned short product_id, const wchar_t *serial_number) Open a HID device using a Vendor ID (VID), Product ID (PID) and optionally a serial number.
HID_API_EXPORT **hid_device** *HID_API_CALL	**hid_open_path** (const char *path) Open a HID device by its path name.
int HID_API_EXPORT HID_API_CALL	**hid_write** (**hid_device** *device, const unsigned char *data, size_t length) Write an Output report to a HID device.
int HID_API_EXPORT HID_API_CALL	**hid_read_timeout** (**hid_device** *dev, unsigned char *data, size_t length, int milliseconds) Read an Input report from a HID device with timeout.
int HID_API_EXPORT HID_API_CALL	**hid_read** (**hid_device** *device, unsigned char *data, size_t length) Read an Input report from a HID device.
int HID_API_EXPORT HID_API_CALL	**hid_set_nonblocking** (**hid_device** *device, int nonblock) Set the device handle to be non-blocking.
int HID_API_EXPORT HID_API_CALL	**hid_send_feature_report** (**hid_device** *device, const unsigned char *data, size_t length) Send a Feature report to the device.
int HID_API_EXPORT HID_API_CALL	**hid_get_feature_report** (**hid_device** *device, unsigned char *data, size_t length) Get a feature report from a HID device.
void HID_API_EXPORT HID_API_CALL	**hid_close** (**hid_device** *device) Close a HID device.
int HID_API_EXPORT_CALL	**hid_get_manufacturer_string** (**hid_device** *device, wchar_t *string, size_t maxlen) Get The Manufacturer String from a HID device.
int HID_API_EXPORT_CALL	**hid_get_product_string** (**hid_device** *device, wchar_t *string, size_t maxlen) Get The Product String from a HID device.
int HID_API_EXPORT_CALL	**hid_get_serial_number_string** (**hid_device** *device, wchar_t *string, size_t maxlen) Get The Serial Number String from a HID device.
int HID_API_EXPORT_CALL	**hid_get_indexed_string** (**hid_device** *device, int string_index, wchar_t *string, size_t maxlen) Get a string from a HID device, based on its string index.
HID_API_EXPORT const wchar_t *HID_API_CALL	**hid_error** (**hid_device** *device) Get a string describing the last error which occurred.

Figure 5.3: HID API functions

HIDAPI provides a clean and consistent interface for each platform, making it easy to develop applications which communicate with USB HID devices without having to know the details of the HID libraries and interfaces on each platform. Programs using HIDAPI are driverless, meaning they do not require the use of a custom driver

for each device on each platform (HID API, 2014).

5.3 Simple Gauge Interface

As said before a simple gauge is simple to write to as no info is needed to be read from the gauge. To write to a simple gauge, these are the steps that are needed to be taken:

First, declare an HID device pointer and set it to the output of HID Open with inputs of the product ID and Vendor ID that matches the gauge.

Second, set the data you need to write to the gauge and calibrate it to the appropriate unit if needed.

Third, call the HID Write function with inputs of the data you need written. hid_write takes inputs of the device pointer, instructions array(with data to be written), and the length of array.

5.4 Complex Gauge Interface

Complex gauge is a bit more complicated as there is data to be read from the gauge as well that also influnces what should be written to the gauge. These are the steps that are needed to be taken:

First, declare an HID device pointer and set it to the output of HID Open with inputs of the product ID and Vendor ID that matches the gauge

Second, call the HID Write function with the request array to be able to read the needed data from the gauge

Third, call HID Read to read the data array that was requested from the gauge

Fourth, set the data you need to write to the gauge and calibrate it to the appropriate unit if needed.

Fifth, now knowing the input of the gauge and the output to be written to the gauge that data can be mapped to the value needed to written and sent to the gauge through HID Write with inputs of the mapped data you need written.

5.5 Linux Input drivers

This is a collection of drivers that is designed to support all input devices under Linux. While it is currently used only on for USB input devices, future use (say 2.5/2.6) is expected to expand to replace most of the existing input system, which is why it lives in drivers/input/ instead of drivers/usb/ (Kernel.org, 2014).
It has events handles that get events from input and pass them where needed via various interfaces - keystrokes to the kernel, mouse movements via a simulated PS/2 interface to GPM and X and so on.

5.6 Gauges Client Process

The gauges client module is simple as in it reads the various value of the different data that needs to be showed on the guages from the simulator and just sends them through UDP connection and keeps sending messages as long as the server is receiving

messages. It knows that the server is getting the messages since the server replies back every time a message is sent.

5.7 Gauges Server Process

The Gauges servers for both the the complex and the simple gauges follow the same implementation. The implementation was done in a way to give the system the best performance possible. As some of the gauges take time to process, read and/or write to, it was better to run them simultaneously rather than in sequence. So the server is running multiple threads so that it can process multiple messages for different gauges at the same time.

There's a thread for reading the messages and saving them in an array. Since this array is of finite length, it only keeps the most updated values and then the other threads each go through an element (message) in the array and process it to control the needed gauge. This way the gauges always get the most updated values and keep displaying them until updated. This can be shown in the Figure 5.5

The Implementation can be simplified to the following:

First, the network connection is initialized by defining the UDP socket, the connection type and the port that will be used.

Second, objects for the gauges that will be used are instantiated using the instruments factory.

Third, using the instruments manager we start the processes for each of the objects. this will call the HID open function so communication can be done with the hardware.

Fourth, multiple threads are created to process the incoming messages. These threads include a read thread that will be reading the messages and saving them in an array of length (n), and (n) process threads each will take an element from the array(the messages that were saved) from 0 to n-1 from the array and process it accordingly. The process threads are locked and not running until the first message is received by the read thread

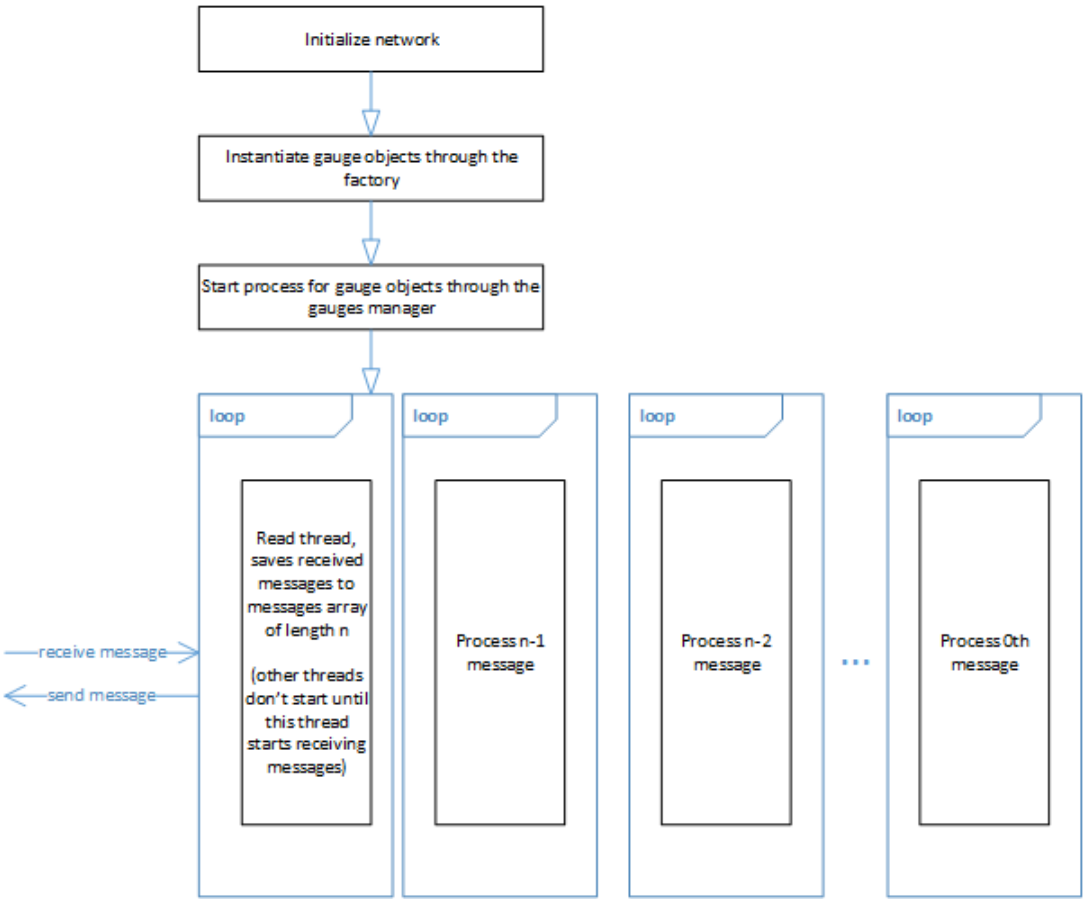


Figure 5.4: Gauges Server Process

5.8 Control Client Process

The control client module is located on the FlightSIM side as an integrated module in FlightSIM. This client is the one that will receive data from the control server to write it to the FlightSIM. So once you start the client it tries to connect to the server; once the connection is set between the client and the serer, and starts receiving input from the server, and then these data can be written to FlightSIM. And if anything needs to be written back to the controls it sent back to the server to be written to the hardware.

5.9 Control Server Process

The server is the module connected to the hardware. So, it waits for the client trying to communicate with it. Once the client asks for data and the connection is set, the server opens HID connection to the hardware controls, reads the data, packs the data, and sends it to the client and then it reads any messages back, process them and write them to hardware if needed.
The Implementation can be simplified to the following:

First, the network connection is initialized by defining the UDP socket, the connection type and the port that will be used.

Second, objects for the controls that will be used are instantiated using the instruments factory.

Third, using the instruments manager we start the processes for each of the objects. and start the process so communication can be done with the hardware.

Fourth, A thread is created for each control to read the input and another communicating thread is also created to send this data and to process any data received back from FlightSIM

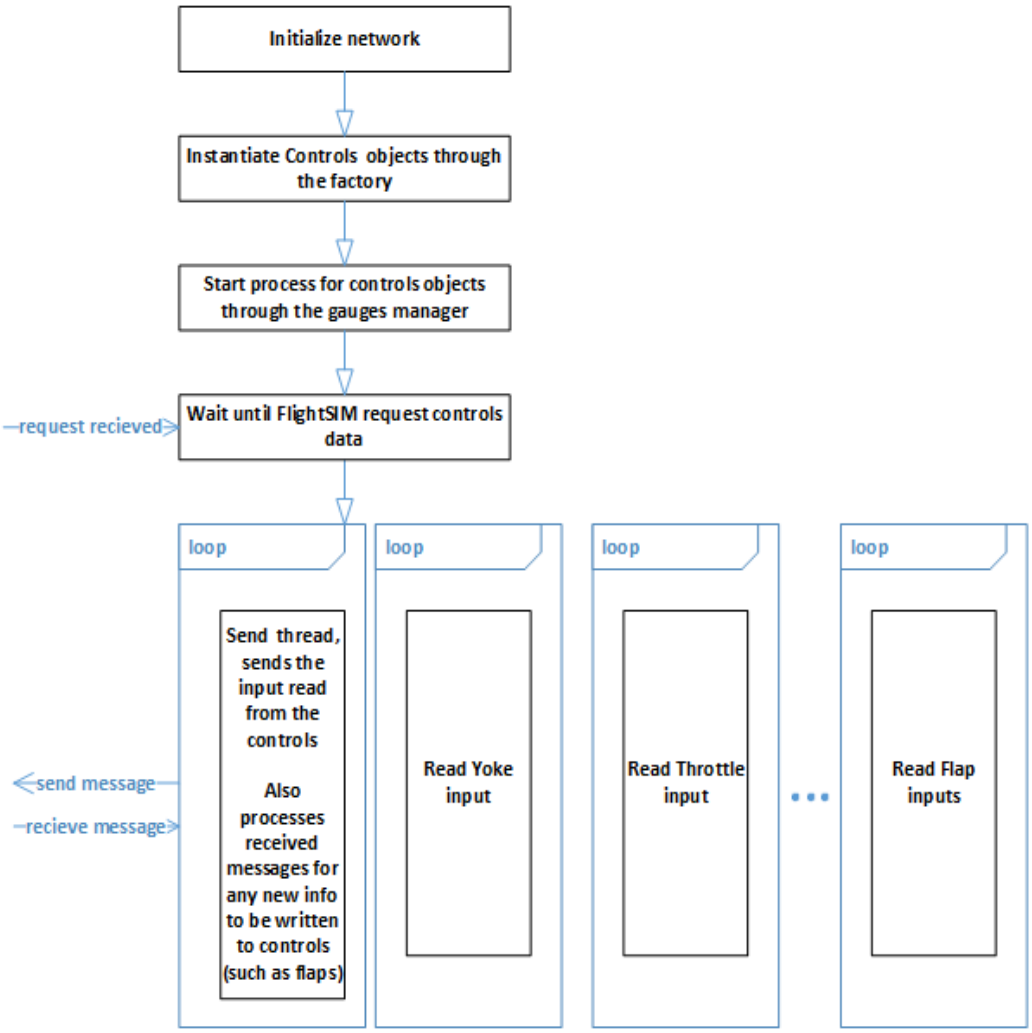


Figure 5.5: Gauges Server Process

5.10 Simple Control Interface

As said before, simple controls are simple in the sense that there is no data to write to it, so we're only reading the hardware input to the system. HID API was also used to interact with most of the controls; however there were also some controls that didn't work HIDAPI and instead it was possible to either use them as joystick or USB events. USB events handler was used in that case as it doesn't require joystick calibration.

In case of HIDAPI simple control:

First, declare an HID device pointer and set it to the output of HID Open with inputs of the product ID and Vendor ID that matches the control instrument.

Second, call the HID Write function with the request array to be able to read the needed data from the control.

Third, call HID Read to read the data input array from the control and this will return the data that requested that can be used with the simulation.

In case of USB device events handler:

First, the device id is needed to be known instead of the product and vendor ID

Second, use int open(const char *path, int oflag, ...) function that takes the path to the USB device and the oflag constructed by a bitwise-inclusive OR of flags, the flags chosen in this case are O_RDONLY— O_NONBLOCK. O_RDONLY opens the device for reading only, O_NONBLOCK for non-block so open() for reading-only shall return without delay. The open() function shall establish the

connection between a device and a descriptor. this descriptor is to be used by other I/O functions to refer to this device.

Third, use ssize_t read(int fd, void *ev, size_t count). read() attempts to read up to count bytes from file descriptor fd into the buffer starting at ev and always returns whole number of input events on a read. Their layout is:

```
struct input_event {
        struct timeval time;
        unsigned short type;
        unsigned short code;
        unsigned int value;
};
```

Each event has it own 'code' so for example in the Rudder, the left brake has event 'code' 3, the right brake has event 'code' 4, and the 'value' is the value the event carries. The 'type' expresses what events can be generated or accepted by this input device; example the event type can be (EV_KEY) key,(EV_REL) relative or (EV_ABS) absolute.The 'value' is the value returned by the event.

5.11 Complex Control Interface

The complex controls require reading the hardware input, and then writing to the hardware as well. There were only 2 controls that were complex: the flaps switch that had a flap indicator that indicates what is the actual angle of the flaps. The flaps switch process follows a procedure that can be simplified as follow:

First, declare an HID device pointer for the flaps and set it to the output of HID

Open with inputs of the product ID and Vendor ID that matches the flaps instrument.

Second, call the HID Write function with the request array to be able to read the needed data from the control, in this case the flap angle that the plane should follow

Third, call HID Read to read the data array from the control that will give the flap angle

Fourth, use HID Write to write the data received from the simulator to the control, in this case the angle the flaps has reached since getting the input from the switch.

The other complex control is the radio stack which is more complicated as it has multiple different inputs and outputs.

5.11.1 Radio Stack

The radio stack is the most complex control or device in the system.The radio stack has 9 inputs to FlightSIM and 5 outputs it receives from FlightSIM to display. In total these nine inputs are read from 20 different hardware inputs(buttons, knobs , switches) that are then calibrated to the nine inputs that will be sent to the simulator and 4 digital displays that have various 7-segments digits and 16-segments digits that will display info received back from the simulator all needed to be programmed bit by bit(no pun intended) to produce quantile values and words that will display frequencies, distances, airports codes, etc.

5.12 Software Scenario

Finally, all the above sections add up to make up the operational software scenario. A software scenario is how the software is expected to run and interact with various parts showing the sequence of the events happening and the synchronization between different processes.

The next three figures show different simplified scenarios of operation for the gauges, simple controls, and complex controls, in this order. They display the events that happen sequentially and the synchronization between different processes that takes place for the system to work when it's fully integrated.

5.12.1 Gauges Scenario

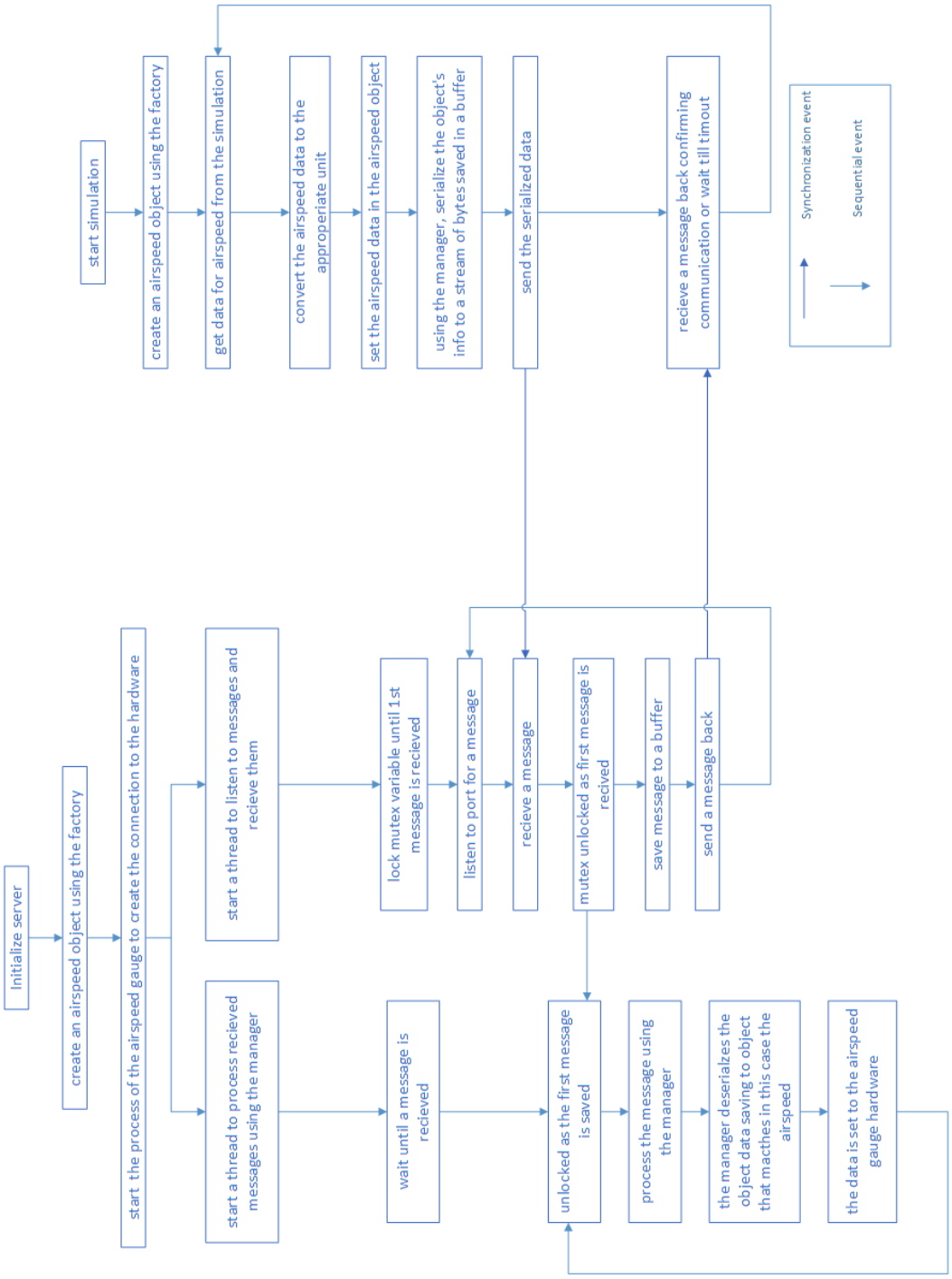


Figure 5.6: Gauges Scenario

5.12.2 Controls Scenarios

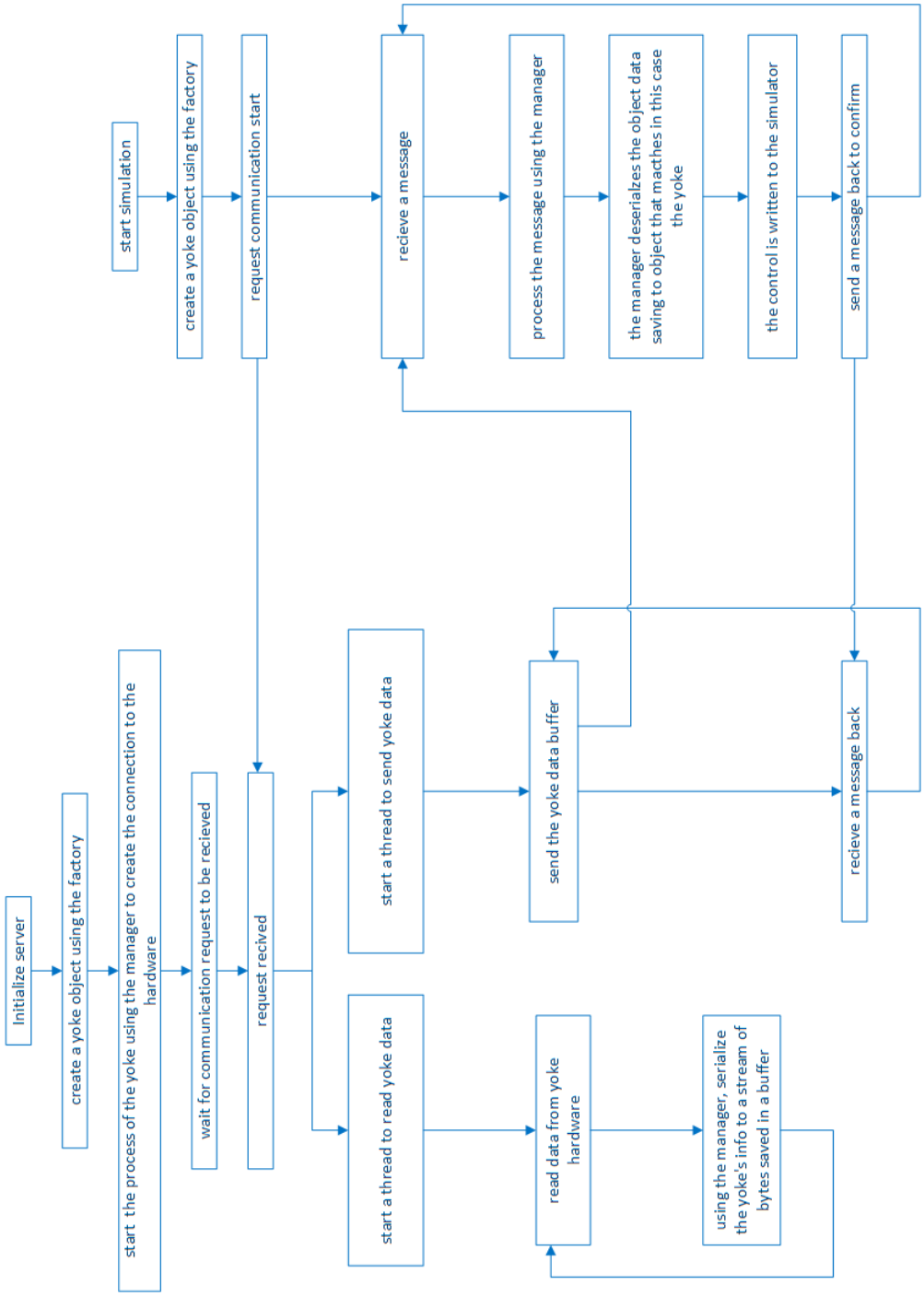


Figure 5.7: Simple Control Scenario

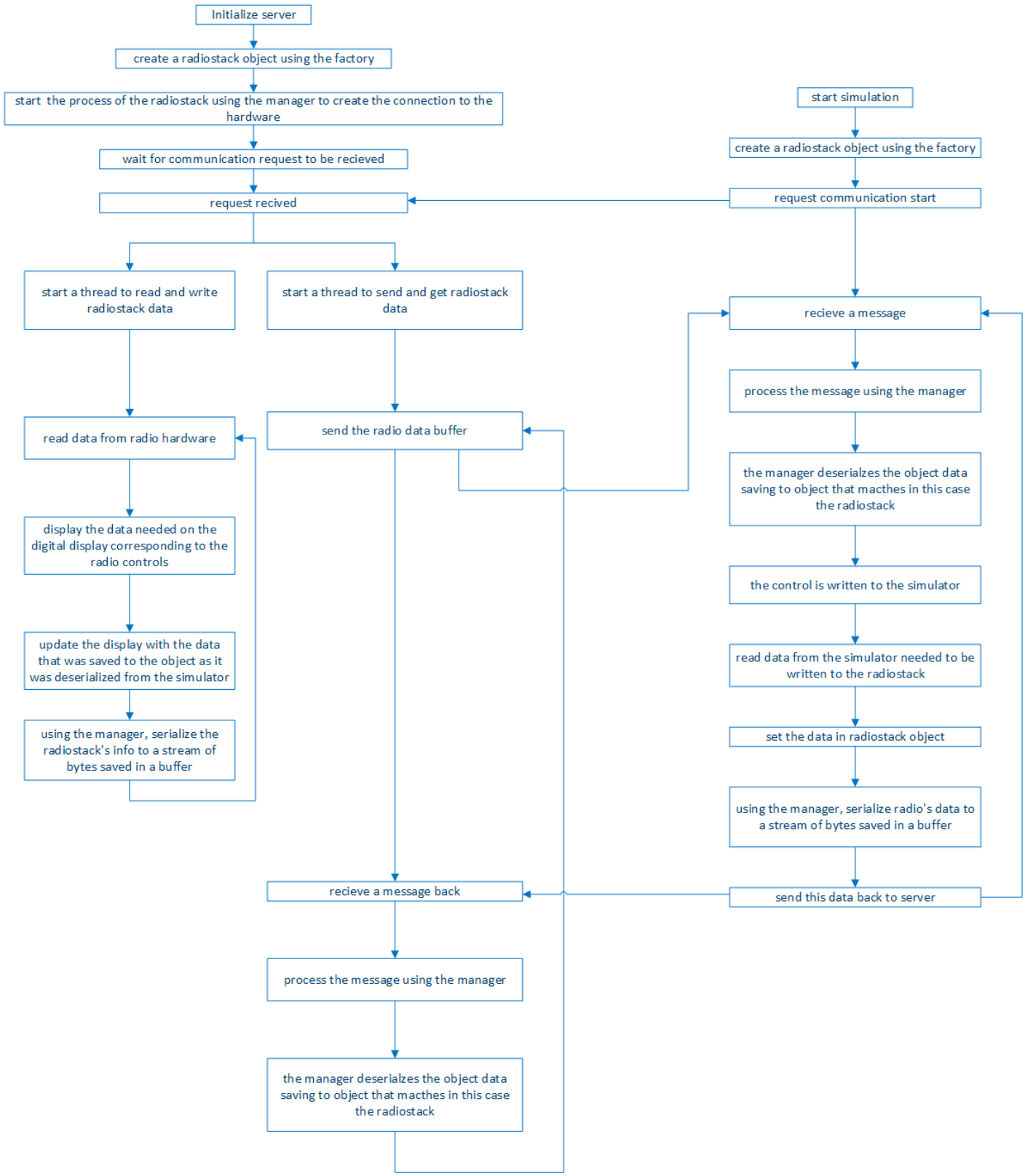


Figure 5.8: Complex Control Scenario

5.13 Error Avoidance

As the system design required stability, some error control needed to be integrated to avoid problems while using the system. The kind of errors that were anticipated are: network errors, and instrument errors.

5.13.1 Network Errors

Network errors are the errors that can occur in the process of communication. UDP as a protocol has error checking using checksums so a packet will always be delivered as a full packet. If a packet arrives with incorrect checksum, it is discarded from the stack. It, however, has no recovery options and no acknowledgment response. So, in the case of a message getting lost on the way to FlightSIM, we do not want FlightSIM to hang waiting for it, and so a very small timeout for receiving data was added so a read statement would not block FlightSIM from processing for too long. If a receive function while blocking for this period of time receives data, the function will return normally with the data. If no data has been received and the timeout was reached then -1 is returned as if the socket was specified to be nonblocking.

5.13.2 Instruments Errors

The other kind of error that can be anticipated is if a server try to read or write from an instrument that is not plugged in. With no error checking, if you try to write to a device that is not connected to USB, the server responsible will crash. So, in order to eliminate this problem, the connection status of each device is checked when a server requests connection to it.

Chapter 6

Testing

In this chapter, some of the testing the simulator went through will be discussed. Through various test cycles, the cockpit interfacing was tested to make sure it has a good working design that maintained usability and functionality. The testing was throughout the implementation process and included different aspects of testing for different parts of the project that will be mentioned be in this chapter.

This system being user driven system that integrates hardware and software, a response can always be expected from hardware and that is what needs to be tested mostly. Automated testing is a challenge when it comes to hardware and so manual testing for this system was more approachable.

6.1 Unit testing

First, unit testing was utilized to test various pieces of code. The main goal of unit testing is to test a small unit of software on its own isolated from the remainder of the code, and determine if it is behaving as required. Then this unit can be integrated

with the remainder of the code.

Since each instrument was created as its own object, each instrument was first tested as a unit individually to see if it's working. Each instrument had it's class methods tested to see if they give the appropriate functionality and if the hardware works with the software written in the intended way.

Example here was testing the yoke. So first the yoke class methods are tested to see if a connection can be connected to the hardware, to see if the input from the yoke can be read, and if it is calibrated to the right values.

6.2 Integration testing

Once a unit was tested to be working and functional, it can be integrated with other units in a larger structure that can be tested to ensure functionality over a bigger part of the project

Example here was the testing done of some the controls put together such as yoke, throttle, rudder in a server client setup that reads and prints the data read from hardware ensuring no software errors are occurring and comparing them to the physical input ensuring no lag, or discrepancies between both values.

6.3 System testing

System testing would mean to test a full system responsible for a full functionality in the way that it should be used to see whether all the components of the system when put together are still working as expected.

Example here is testing the controls system responsible for feeding the yoke, rudder,

throttle, flaps, trim to the simulator, and checking if any errors are happening. Things that were tested in this case would be: if the data are arriving from the hardware to the simulator as expected, if any lag in transferring the data is existent, are the data being used appropriately by the simulator as in if the airplane is corresponding to the data sent, if the simulator is lagging or running at full speed.

6.4 Pilot testing

Once the systems are confirmed to work, the next step is to do acceptance testing which means to test the whole project by someone who can considered a customer of the product. Since the product here is a flight simulator, the best testing user for this would be a flying pilot with prior experience in flight and plane simulators. Abdallah AbdelRahman, a pilot and a friend, volunteered to help test the simulator and informing us of any problems he notices during his test flights on the simulator. First he checked the devices to see whether they function as they are supposed to and informed me of any differences. Second, he checked whether it is as responsive as it should be in air and on ground.

6.5 Timing Analysis

6.5.1 FAA Flight Simulators Sesponse

According to the FAA standards, for a Flying Training Device level 5 and 6, the relative response of flight instruments may not exceed 300 milliseconds. Similarly, 300 milliseconds of airplane response is the maximum acceptable for level A & B

Full Flight Simulator, while 150 milliseconds of airplane response is the maximum acceptable for level C & D. (Part 60 Final Rule, Consolidated Version, 2008)

6.5.2 Response Time Analysis

The FlightSIM software can run at different rates of FPS (frames per second). Comparing FlightSIM to games, 30FPS game has a minimum potential lag of 100ms(3 frames) but a lot of games exceed this number with the average around 133ms(4 frames). Studies show that gamers usually start to notice lag at around 200ms delay in response (Leadbetter, 2009). Studies also show that above 225 ms delay, movement times and error rates increase by 64% and 214% respectively compared to zero lag condition. (MacKenzie and Colin, 1993).

This means that a control input should be updated every 4 out 30 frames at maximum on a 30 FPS system so the response is never longer than 150 milliseconds, and on a 60 FPS every 9 frames out of 60 at maximum. When simulation is running at 60 FPS, and for the main 5 controls being sent sequentially that means each control can have a max lag of 4 frames so lag would not be more than 0.0667 sec or 66 millisecond which is lower than the established threshold and so the software of the system passes the response time requirements to be a responsive flight simulator.

To actually test the response time from the hardware input to the software is out of the scope of this thesis as it is a very complicated thing to do as it requires special sensors and hardware modding and would very time consuming.

6.5.3 Network Delay

The LAN network used in the lab was tested to check if the network would add any noticeable latency to the user input. The tool used for testing the network was Iperf. Iperf is a commonly used network testing tool that can create Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) data streams and measure the throughput of a network that is carrying them. As can be seen in Figure 6.1, after running the test multiple times over the UDP protocol, the results shows a latency on average of 0.017 milliseconds. It was also tested how long does it take to serialize an object data, sent it, read it, and de-serialize it again and that was about 0.22ms on average. Of course, these numbers are very small and thus it was concluded that the network won't have any major delay effect on the inputs and is negligible.

```
Administrator: Command Prompt
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\martin>E:\iperf-2.0.5-2-win32\iperf.exe -c 192.168.0.107 -u
------------------------------------------------------------
Client connecting to 192.168.0.107, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size: 64.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.0.104 port 49951 connected with 192.168.0.107 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[  3] Sent 893 datagrams
[  3] Server Report:
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.007 ms    0/  893 (0%)

C:\Users\martin>E:\iperf-2.0.5-2-win32\iperf.exe -c 192.168.0.107 -u
------------------------------------------------------------
Client connecting to 192.168.0.107, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size: 64.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.0.104 port 49952 connected with 192.168.0.107 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[  3] Sent 893 datagrams
[  3] Server Report:
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.023 ms    0/  893 (0%)

C:\Users\martin>E:\iperf-2.0.5-2-win32\iperf.exe -c 192.168.0.107 -u
------------------------------------------------------------
Client connecting to 192.168.0.107, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size: 64.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.0.104 port 49953 connected with 192.168.0.107 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[  3] Sent 893 datagrams
[  3] Server Report:
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.019 ms    0/  893 (0%)

C:\Users\martin>E:\iperf-2.0.5-2-win32\iperf.exe -c 192.168.0.107 -u
------------------------------------------------------------
Client connecting to 192.168.0.107, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size: 64.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.0.104 port 49954 connected with 192.168.0.107 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[  3] Sent 893 datagrams
[  3] Server Report:
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.019 ms    0/  893 (0%)

C:\Users\martin>E:\iperf-2.0.5-2-win32\iperf.exe -c 192.168.0.107 -u
------------------------------------------------------------
Client connecting to 192.168.0.107, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size: 64.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.0.104 port 49955 connected with 192.168.0.107 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[  3] Sent 893 datagrams
[  3] Server Report:
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.017 ms    0/  893 (0%)

C:\Users\martin>
```

Figure 6.1: Iperf Network Test Results

Chapter 7

Discussion

7.1 Conclusion

Flight Simulators remain today as an important training tool as they have ever been. They have proved how much of important tools they are for learning and practicing flight. They have been advancing over the years to the state where they are at now. Flight simulators range in complexity of the hardware and the software included. Flight Simulators are very expensive and usually also hard to maintain and expand on as they don't usually have a good modular design that is easy to expand on.

In this thesis, the design and implementation of a software system used to interface a hardware controls cockpit that include many USB devices ranging in complexity to a software flight simulator on a distributed system were presented. The system created made for a unique solution that satisfied the needs of real-time plane simulation that was low cost and expendable making it more future proof for various new features and additions. The design was based on a modern object oriented design making the system easy to expand and to add all different sort of instruments. The simulator can

be used to run various experiments with different scenarios and all sorts of different plane physics.

The system was build using C++ following an object oriented design and utilizing various design patterns to make a robust implementation that was capable of handling various devices all at the same time and to have the extendability for future additions without changing much in the design. This lead to a modular design that is both robust and stable.

7.2 Future Work

As this simulator is still work in progress, there is still room for new additions and it can still be enhanced and extended with various features to make for a better experience. Some of these features are:

- The cockpit should be installed in the motion simulator to add motion feedback to the flight experience in accordance with the flight scenario.
- The simulation experience can become more immersive with controls that have haptic feedback. So a user while steering with the yoke and the rudder can feel back the environment forces on the plane based on airspeed and wind.
- VAPS XT can be used for rapid development of interactive graphical interfaces to create, real-time Human-Machine Interface. It can be used to expand and add virtual hardware instrument such as an Aviation GPS unit to be used in the simulator.

Appendix A

Cockpit Instruments Documentation

Contents

Chapter 1

The Cockpit Instruments Documentation

This is the documentation for the Cockpit instruments.

It documents:

The class hierarchy design for the Cockpit instruments.

Chapter 2

Class Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

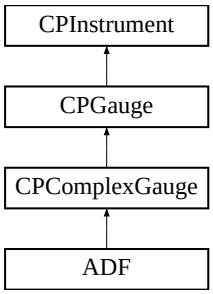
Chapter 4

Class Documentation

4.1 ADF Class Reference

The ADF (Automatic Direction Finder) Gauge Class.

`#include <adf.h>`Inheritance diagram for ADF::



Public Member Functions

- void setGauge ()

  Set gauge is used to set the data saved in the object to the hardware.

- void setHID ()

  Initializes the link to the hardware.

- void setData (int x, float i)

  saves the data that will be written to hardware in the object. It takes 2 inputs, data type is the data type to be set in case the device takes more than one input; dataVal is the value to be set.

Friends

- class CPInstrumentFactory

4.1.1 Detailed Description

The ADF (Automatic Direction Finder) Gauge Class.

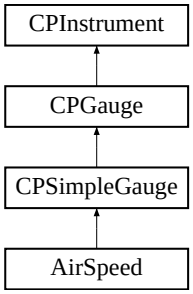
The documentation for this class was generated from the following files:

- CPInstruments/adf.h
- CPInstruments/adf.cpp

4.2 AirSpeed Class Reference

The AirSpeed Indicator Class.

`#include <airspeed.h>`Inheritance diagram for AirSpeed::



Public Member Functions

- void setGauge ()

  Set gauge is used to set the data saved in the object to the hardware.

- void setHID ()

  Initializes the link to the hardware.

- void setData (int x, float i)

  saves the data that will be written to hardware in the object. It takes 2 inputs, data type is the data type to be set in case the device takes more than one input; dataVal is the value to be set.

Friends

- class CPInstrumentFactory

4.2.1 Detailed Description

The AirSpeed Indicator Class.

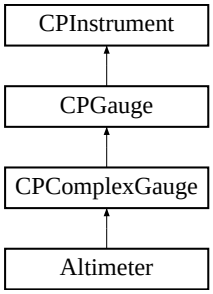
The documentation for this class was generated from the following files:

- CPInstruments/airspeed.h
- CPInstruments/airspeed.cpp

4.3 Altimeter Class Reference

The Altimeter Gauge Class.

`#include <altimeter.h>`Inheritance diagram for Altimeter::



Public Member Functions

- void setGauge ()

  Set gauge is used to set the data saved in the object to the hardware.

- void setHID ()

  Initializes the link to the hardware.

- void setData (int x, float i)

  saves the data that will be written to hardware in the object. It takes 2 inputs, data type is the data type to be set in case the device takes more than one input; dataVal is the value to be set.

Friends

- class CPInstrumentFactory

4.3.1 Detailed Description

The Altimeter Gauge Class.

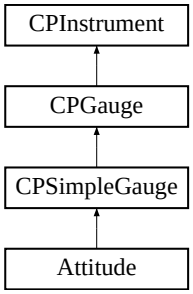
The documentation for this class was generated from the following files:

- CPInstruments/altimeter.h
- CPInstruments/altimeter.cpp

4.4 Attitude Class Reference

The Attitude Indicator Class.

`#include <attitude.h>`Inheritance diagram for Attitude::



Public Member Functions

- void setGauge ()

  Set gauge is used to set the data saved in the object to the hardware.

- void setHID ()

  Initializes the link to the hardware.

- void setData (int x, float i)

  if x is 1 set i to pitch angle, if x is 2 set i to roll angle

Friends

- class CPInstrumentFactory

4.4.1 Detailed Description

The Attitude Indicator Class.

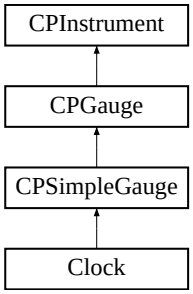
The documentation for this class was generated from the following files:

- CPInstruments/attitude.h
- CPInstruments/attitude.cpp

4.5 Clock Class Reference

The Digital Clock Class.

`#include <clock.h>`Inheritance diagram for Clock::



Public Member Functions

- void setGauge ()

  Set gauge is used to set the data saved in the object to the hardware.

- void setHID ()

  Initializes the link to the hardware.

- void setData (int x, float i)

  saves the data that will be written to hardware in the object. It takes 2 inputs, data type is the data type to be set in case the device takes more than one input; dataVal is the value to be set.

Friends

- class CPInstrumentFactory

4.5.1 Detailed Description

The Digital Clock Class.

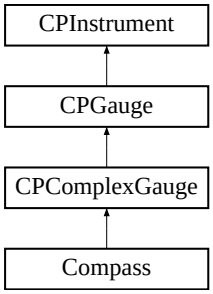
The documentation for this class was generated from the following files:

- CPInstruments/clock.h
- CPInstruments/clock.cpp

4.6 Compass Class Reference

The Magnetic Compass Class.

`#include <compass.h>`Inheritance diagram for Compass::



Public Member Functions

- void setGauge ()

  Set gauge is used to set the data saved in the object to the hardware.

- void setHID ()

  Initializes the link to the hardware.

- void setData (int x, float i)

  saves the data that will be written to hardware in the object. It takes 2 inputs, data type is the data type to be set in case the device takes more than one input; dataVal is the value to be set.

Friends

- class CPInstrumentFactory

4.6.1 Detailed Description

The Magnetic Compass Class.

The documentation for this class was generated from the following files:

- CPInstruments/compass.h
- CPInstruments/compass.cpp

4.7 Compressor Class Reference

Compressor Class for compressing data.

```
#include <messages.h>
```

Public Member Functions

- **Compressor** (char *ptr)
- unsigned long **GetSize** (void) const
- Compressor & **operator**<< (const bool &x)
- Compressor & **operator**<< (const char &x)
- Compressor & **operator**<< (const unsigned char &x)
- Compressor & **operator**<< (const short &x)
- Compressor & **operator**<< (const unsigned short &x)
- Compressor & **operator**<< (const long &x)
- Compressor & **operator**<< (const unsigned long &x)
- Compressor & **operator**<< (const float &x)
- Compressor & **operator**<< (const char *text)

4.7.1 Detailed Description

Compressor Class for compressing data.

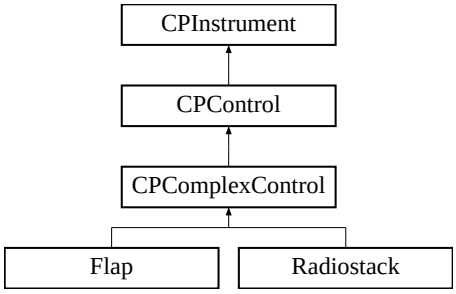
The documentation for this class was generated from the following file:

- CPInstruments/messages.h

4.8 CPComplexControl Class Reference

CPComplexControl Abstract Class.

`#include <CPComplexControl.h>`Inheritance diagram for CPComplexControl::



Public Member Functions

- virtual float setData (char ∗dataType)=0

  Set Data is used to set the data that was read from the hardware control to a specfic flightSIM input. dataType is used in case of an control that has multiple inputs to the system.

- virtual void openControl ()=0

  Open control is used to set the link to read from the hardware.

- virtual void getData (int number)=0

  Get Data is used to read the data from the control. val is used in some cases where a control that has multiple inputs to the system.

- virtual void **Compress** (Compressor &data) const =0
- virtual bool **Decompress** (Decompressor &data)=0
- virtual void setControlInput (char ∗inputType, char ∗inputVal)=0

  Set control input is used for complex controls that have hardware output as well. InputType is the type of data to be written, and inputVAl is the value of the data.

- CPControl::CPInstrumentType **GetControlType** (void) const

Public Attributes

- CPControl::CPInstrumentType **cpcType**

Protected Member Functions

- **CPComplexControl** (CPControl::CPInstrumentType tp)

4.8.1 Detailed Description

CPComplexControl Abstract Class.

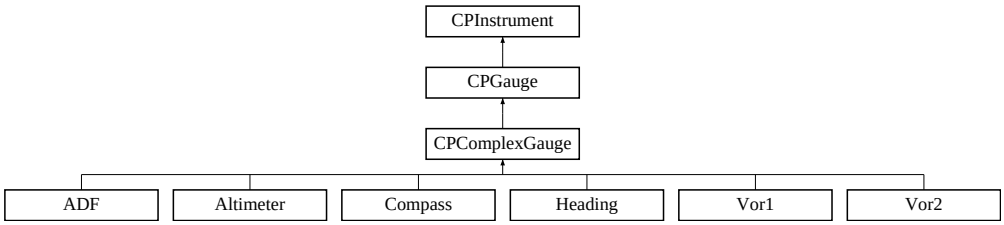
The documentation for this class was generated from the following files:

- CPInstruments/CPComplexControl.h
- CPInstruments/CPComplexControl.cpp

4.9 CPComplexGauge Class Reference

CPComplexGauge Abstract Class.

`#include <CPComplexGauge.h>`Inheritance diagram for CPComplexGauge::



Public Member Functions

- virtual void setGauge ()=0

  Set gauge is used to set the data saved in the object to the hardware.

- virtual void setHID ()=0

  Initializes the link to the hardware.

- virtual void setData (int x, float i)=0

  saves the data that will be written to hardware in the object. It takes 2 inputs, data type is the data type to be set in case the device takes more than one input; dataVal is the value to be set.

- virtual void **Compress** (Compressor &data) const =0
- virtual bool **Decompress** (Decompressor &data)=0
- CPGauge::CPInstrumentType **GetGaugeType** (void) const

Public Attributes

- CPGauge::CPInstrumentType **cpgType**

Protected Member Functions

- **CPComplexGauge** (CPGauge::CPInstrumentType tp)

4.9.1 Detailed Description

CPComplexGauge Abstract Class.

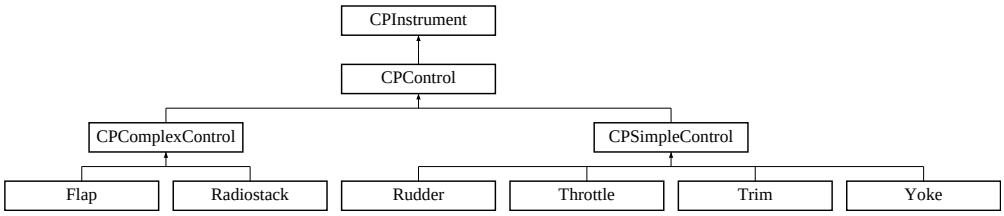
The documentation for this class was generated from the following files:

- CPInstruments/CPComplexGauge.h
- CPInstruments/CPComplexGauge.cpp

4.10 CPControl Class Reference

Abstract base class for CPControl.

`#include <CPControl.h>`Inheritance diagram for CPControl::



Public Types

- enum CPInstrumentType {

  yoke = 0, **trim** = 1, **rudder** = 2, **throt** = 3,

  radiostack = 4, **flap** = 5 }

  The enumeration of the cockpit controls instruments.

- enum controlType { **complexControl**, **simpleControl** }

  The control's type if complex or simple.

- typedef unsigned long **CPControlType**

Public Member Functions

- virtual float setData (char ∗dataType)=0

  Set Data is used to set the data that was read from the hardware control to a specfic flightSIM input. dataType is used in case of an control that has multiple inputs to the system.

- virtual void openControl ()=0

  Open control is used to set the link to read from the hardware.

- virtual void getData (int val)=0

  Get Data is used to read the data from the control. val is used in some cases where a control that has multiple inputs to the system.

- virtual void **Compress** (Compressor &data) const =0
- virtual CPInstrumentType **GetControlType** () const =0
- virtual bool **Decompress** (Decompressor &data)=0
- virtual void setControlInput (char ∗inputType, char ∗inputVal)

  Set control input is used for complex controls that have hardware output as well. InputType is the type of data to be written, and inputVAl is the value of the data.

- controlType **GetCPControlType** (void) const

Public Attributes

- controlType **cpcType**

Protected Member Functions

- **CPControl** (controlType tp)

4.10.1 Detailed Description

Abstract base class for CPControl.

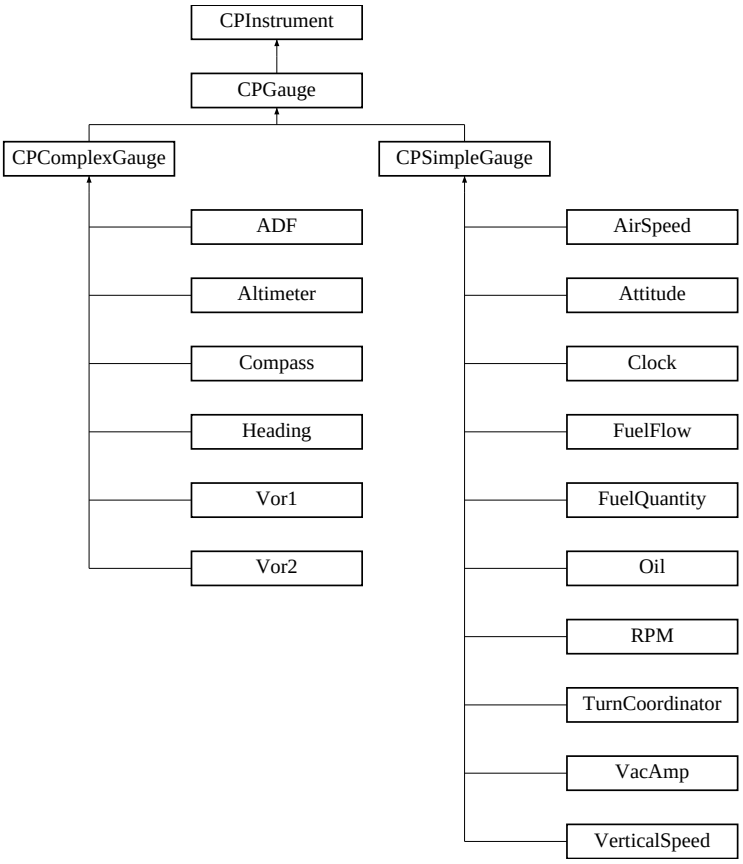
The documentation for this class was generated from the following files:

- CPInstruments/CPControl.h
- CPInstruments/CPControl.cpp

4.11 CPGauge Class Reference

CPGauge Abstract Base Class.

`#include <CPGauge.h>`Inheritance diagram for CPGauge::



Public Types

- enum CPInstrumentType {

  airspeed = 6, **turncoordinator** = 7, **rpm** = 8, **verticalspeed** = 9,

  attitude = 10, **compass** = 11, **adf** = 12, **vor1** = 13,

  vor2 = 14, **heading** = 15, **altimeter** = 16, **fuelquantity** = 17,

  fuelflow = 18, **oil** = 19, **vacamp** = 20, **clock** = 21 }

  The enumeration of the cockpit gauges instruments.

- enum gaugeType { **complexGauge**, **simpleGauge** }

  The gauge's type if complex or simple.

Public Member Functions

- virtual void setGauge ()=0

  Set gauge is used to set the data saved in the object to the hardware.

- virtual void setHID ()=0

  Initializes the link to the hardware.

- virtual void setData (int dataType, float dataVal)=0

  saves the data that will be written to hardware in the object. It takes 2 inputs, data type is the data type to be set in case the device takes more than one input; dataVal is the value to be set.

- virtual void **Compress** (Compressor &data) const =0
- virtual bool **Decompress** (Decompressor &data)=0
- virtual CPInstrumentType **GetGaugeType** () const =0
- gaugeType **GetCPGaugeType** (void) const

Public Attributes

- gaugeType **cpgType**

Protected Member Functions

- **CPGauge** (gaugeType tp)

4.11.1 Detailed Description

CPGauge Abstract Base Class.

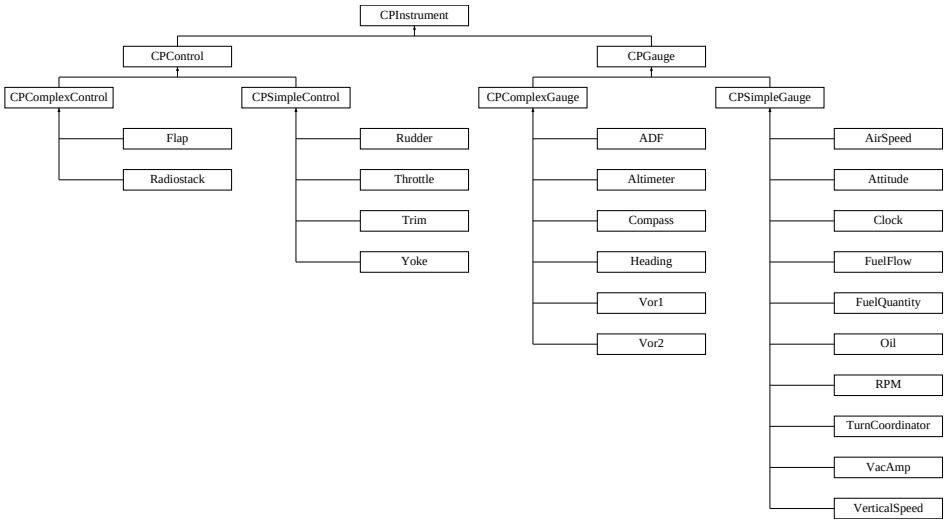
The documentation for this class was generated from the following files:

- CPInstruments/CPGauge.h
- CPInstruments/CPGauge.cpp

4.12 CPInstrument Class Reference

Abstract base class for CPInstrument.

`#include <CPInstrument.h>`Inheritance diagram for CPInstrument::



Public Types

- enum instrumentType { **cpgauge**, **cpcontrol** }

  Types of instruments.

Public Member Functions

- instrumentType GetInstrumentType (void) const

  Return if the instrument is a gauge or a control.

Public Attributes

- instrumentType cpiType

  Instrument type.

Protected Member Functions

- CPInstrument (instrumentType tp)

  Constructor.

- virtual ~CPInstrument ()

  Destructor.

4.12.1 Detailed Description

Abstract base class for CPInstrument.

The documentation for this class was generated from the following files:

- CPInstruments/CPInstrument.h
- CPInstruments/CPInstrument.cpp

4.13 CPInstrumentFactory Class Reference

CPIntrumentFactory for creating concrete objects.

`#include <CPInstrumentFactory.h>`

Public Types

- typedef unsigned long **CPInstrumentType**

Public Member Functions

- CPControl ∗ createCPControl (CPInstrumentType type)

  Creates concrete Controls.

- CPGauge ∗ createCPGauge (CPInstrumentType type)

  creates concrete Gauges

Static Public Member Functions

- static CPInstrumentFactory ∗ **instance** ()

4.13.1 Detailed Description

CPIntrumentFactory for creating concrete objects.

The documentation for this class was generated from the following files:

- CPInstruments/CPInstrumentFactory.h
- CPInstruments/CPInstrumentFactory.cpp

4.14 CPInstrumentManager Class Reference

manages the serialization of instruments objects.

`#include <CPInstrumentManager.h>`

Public Member Functions

- int SendCPInstrument (CPInstrument ∗cpcObj, char ∗buffer)
  compresses an object to be sent as serial data.
- void ProcessStart (CPInstrument ∗cpiObj)
  starts the process for the object.
- int ProcessCPInstrument (char ∗data)
  processes data and de-serializes it to the intended object.

Static Public Member Functions

- static CPInstrumentManager ∗ instance (void)
  Singleton instantiation.

4.14.1 Detailed Description

manages the serialization of instruments objects. CPInstrumentManager manages the serialization and de-serialization of objects.

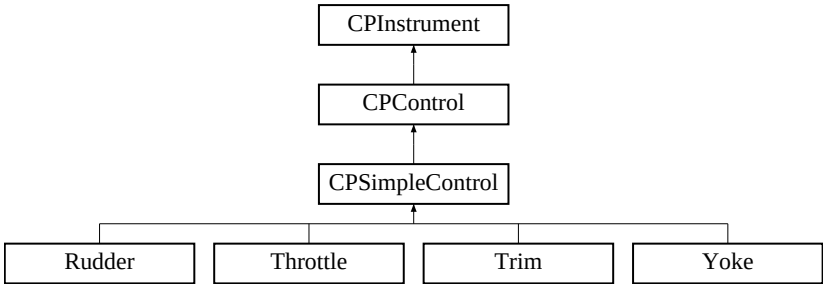
The documentation for this class was generated from the following files:

- CPInstruments/CPInstrumentManager.h
- CPInstruments/CPInstrumentManager.cpp

4.15 CPSimpleControl Class Reference

CPSimpleGauge Abstract Class.

`#include <CPSimpleControl.h>`Inheritance diagram for CPSimpleControl::



Public Member Functions

- virtual float setData (char ∗dataType)=0

  Set Data is used to set the data that was read from the hardware control to a specfic flightSIM input. dataType is used in case of an control that has multiple inputs to the system.

- virtual void openControl ()=0

  Open control is used to set the link to read from the hardware.

- virtual void getData (int number)=0

  Get Data is used to read the data from the control. val is used in some cases where a control that has multiple inputs to the system.

- virtual void **Compress** (Compressor &data) const =0
- virtual bool **Decompress** (Decompressor &data)=0
- CPControl::CPInstrumentType **GetControlType** (void) const

Public Attributes

- CPControl::CPInstrumentType **cpcType**

Protected Member Functions

- **CPSimpleControl** (CPControl::CPInstrumentType tp)

4.15.1 Detailed Description

CPSimpleGauge Abstract Class.

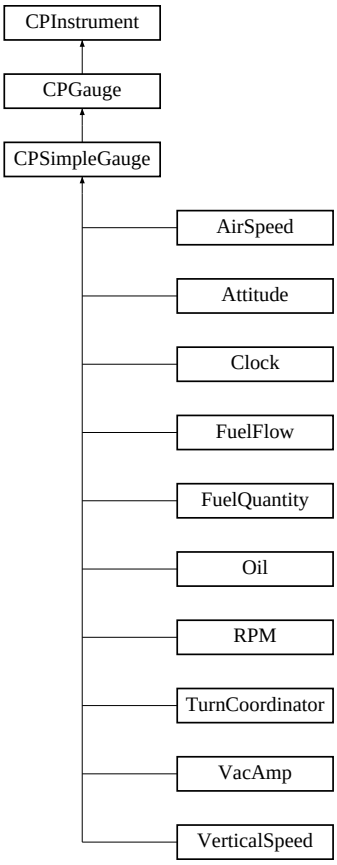
The documentation for this class was generated from the following files:

- CPInstruments/CPSimpleControl.h
- CPInstruments/CPSimpleControl.cpp

4.16 CPSimpleGauge Class Reference

CPSimpleGauge Abstract Class.

`#include <CPSimpleGauge.h>`Inheritance diagram for CPSimpleGauge::



Public Member Functions

- virtual void setGauge ()=0

  Set gauge is used to set the data saved in the object to the hardware.

- virtual void setHID ()=0

  Initializes the link to the hardware.

- virtual void setData (int x, float i)=0

  saves the data that will be written to hardware in the object. It takes 2 inputs, data type is the data type to be set in case the device takes more than one input; dataVal is the value to be set.

- virtual void **Compress** (Compressor &data) const =0
- virtual bool **Decompress** (Decompressor &data)=0
- CPGauge::CPInstrumentType **GetGaugeType** (void) const

Public Attributes

- CPGauge::CPInstrumentType **cpgType**

Protected Member Functions

- **CPSimpleGauge** (CPGauge::CPInstrumentType tp)

4.16.1 Detailed Description

CPSimpleGauge Abstract Class.

The documentation for this class was generated from the following files:

- CPInstruments/CPSimpleGauge.h
- CPInstruments/CPSimpleGauge.cpp

4.17 Decompressor Class Reference

Decompressor Class for decompressing data.

```
#include <messages.h>
```

Public Member Functions

- **Decompressor** (const char *ptr)
- unsigned long **GetSize** (void) const
- Decompressor & **operator**>> (bool &x)
- Decompressor & **operator**>> (char &x)
- Decompressor & **operator**>> (unsigned char &x)
- Decompressor & **operator**>> (short &x)
- Decompressor & **operator**>> (unsigned short &x)
- Decompressor & **operator**>> (long &x)
- Decompressor & **operator**>> (unsigned long &x)
- Decompressor & **operator**>> (float &x)
- Decompressor & **operator**>> (char *text)

4.17.1 Detailed Description

Decompressor Class for decompressing data.

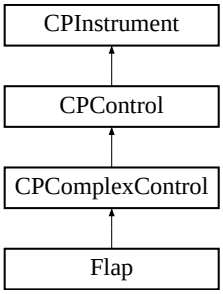
The documentation for this class was generated from the following file:

- CPInstruments/messages.h

4.18 Flap Class Reference

The Flap Switch Class.

`#include <flap.h>`Inheritance diagram for Flap::



Public Member Functions

- float setData (char ∗dataType)

  Set Data is used to set the data that was read from the hardware control to a specfic flightSIM input. dataType is used in case of an control that has multiple inputs to the system.

- void openControl ()

  Open control is used to set the link to read from the hardware.

- void getData (int number)

  Get Data is used to read the data from the control. val is used in some cases where a control that has multiple inputs to the system.

- void setControlInput (char ∗inputType, char ∗inputVal)

  Set control input is used for complex controls that have hardware output as well. InputType is the type of data to be written, and inputVAl is the value of the data.

Friends

- class CPInstrumentFactory

4.18.1 Detailed Description

The Flap Switch Class.

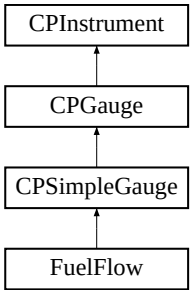
The documentation for this class was generated from the following files:

- CPInstruments/flap.h
- CPInstruments/flap.cpp

4.19 FuelFlow Class Reference

FuelFlow gauge Class.

`#include <fuelflow.h>`Inheritance diagram for FuelFlow::



Public Member Functions

- void setGauge ()

  Set gauge is used to set the data saved in the object to the hardware.

- void setHID ()

  Initializes the link to the hardware.

- void setData (int x, float i)

  saves the data that will be written to hardware in the object. It takes 2 inputs, data type is the data type to be set in case the device takes more than one input; dataVal is the value to be set.

Friends

- class CPInstrumentFactory

4.19.1 Detailed Description

FuelFlow gauge Class.

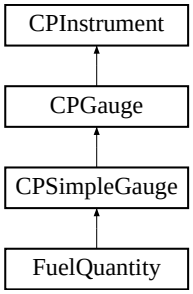
The documentation for this class was generated from the following files:

- CPInstruments/fuelflow.h
- CPInstruments/fuelflow.cpp

4.20 FuelQuantity Class Reference

FuelQuantity gauge Class.

`#include <fuelquantity.h>`Inheritance diagram for FuelQuantity::



Public Member Functions

- void setGauge ()

  Set gauge is used to set the data saved in the object to the hardware.

- void setHID ()

  Initializes the link to the hardware.

- void setData (int x, float i)

  saves the data that will be written to hardware in the object. It takes 2 inputs, data type is the data type to be set in case the device takes more than one input; dataVal is the value to be set.

Friends

- class CPInstrumentFactory

4.20.1 Detailed Description

FuelQuantity gauge Class.

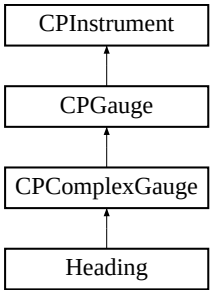
The documentation for this class was generated from the following files:

- CPInstruments/fuelquantity.h
- CPInstruments/fuelquantity.cpp

4.21 Heading Class Reference

True Heading gauge Class.

`#include <heading.h>`Inheritance diagram for Heading::



Public Member Functions

- void setGauge ()

  Set gauge is used to set the data saved in the object to the hardware.

- void setHID ()

  Initializes the link to the hardware.

- void setData (int x, float i)

  saves the data that will be written to hardware in the object. It takes 2 inputs, data type is the data type to be set in case the device takes more than one input; dataVal is the value to be set.

Friends

- class CPInstrumentFactory

4.21.1 Detailed Description

True Heading gauge Class.

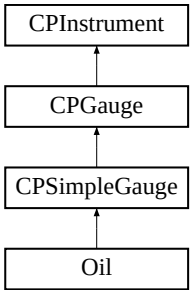
The documentation for this class was generated from the following files:

- CPInstruments/heading.h
- CPInstruments/heading.cpp

4.22 Oil Class Reference

Oil gauge Class.

`#include <oil.h>`Inheritance diagram for Oil::



Public Member Functions

- void setGauge ()

  Set gauge is used to set the data saved in the object to the hardware.

- void setHID ()

  Initializes the link to the hardware.

- void setData (int x, float i)

  saves the data that will be written to hardware in the object. It takes 2 inputs, data type is the data type to be set in case the device takes more than one input; dataVal is the value to be set.

Friends

- class CPInstrumentFactory

4.22.1 Detailed Description

Oil gauge Class.

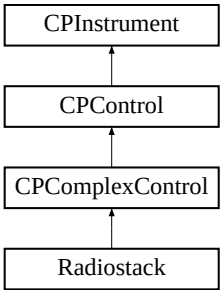
The documentation for this class was generated from the following files:

- CPInstruments/oil.h
- CPInstruments/oil.cpp

4.23 Radiostack Class Reference

The Radiostack Class.

`#include <radiostack.h>`Inheritance diagram for Radiostack::



Public Member Functions

- float setData (char ∗dataType)

  Outputs float output based on dataType. These dataTypes are:ADFinUseFreq,DMEinUseFreq, NAVinUseFreq,NAV2inUseFreq,ADFActiveSend,DMEActiveSend,NAVActiveSend,NAV2ActiveSend,DMEfunctionSwitch.

- void openControl ()

  Open control is used to set the link to read from the hardware.

- void getData (int number)

  If input is 0 get data from NAV1 radio, if 1 get data from DME radio, if 2 get data from ADF radio, if 3 get data from NAV2 radio, if 4 get data from XPDR radio.

- void setControlInput (char ∗inputType, char ∗inputVal)

  SetsRadiostack the inputVal to hardware input based on inputType. These inputTypes are:NAV1,NAV2,DME,ALT,NAV1OBS,NAV2OBS.

Friends

- class CPInstrumentFactory

4.23.1 Detailed Description

The Radiostack Class.

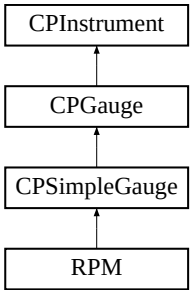
The documentation for this class was generated from the following files:

- CPInstruments/radiostack.h
- CPInstruments/radiostack.cpp

4.24 RPM Class Reference

The RPM Gauge Class.

`#include <rpm.h>`Inheritance diagram for RPM::



Public Member Functions

- void setGauge ()

  Set gauge is used to set the data saved in the object to the hardware.

- void setHID ()

  Initializes the link to the hardware.

- void setData (int x, float i)

  saves the data that will be written to hardware in the object. It takes 2 inputs, data type is the data type to be set in case the device takes more than one input; dataVal is the value to be set.

Friends

- class CPInstrumentFactory

4.24.1 Detailed Description

The RPM Gauge Class.

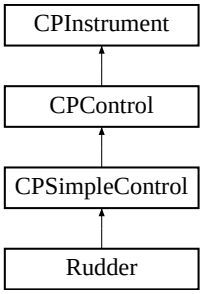
The documentation for this class was generated from the following files:

- CPInstruments/rpm.h
- CPInstruments/rpm.cpp

4.25 Rudder Class Reference

The Rudder Control Class.

`#include <rudder.h>`Inheritance diagram for Rudder::



Public Member Functions

- float setData (char *dataType)

  setData has 3 different possible inputs here: "steering" to set the data read from steering motion; "leftBrake" to set the data read from the differential left brake; and "rightBrake" to set the data read from the differential right brake.

- void openControl ()

  Open control is used to set the link to read from the hardware.

- void getData (int number)

  Get Data is used to read the data from the control. val is used in some cases where a control that has multiple inputs to the system.

Friends

- class CPInstrumentFactory

4.25.1 Detailed Description

The Rudder Control Class.

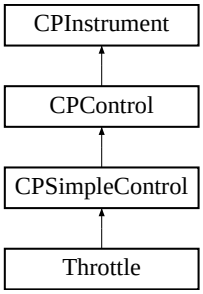
The documentation for this class was generated from the following files:

- CPInstruments/rudder.h
- CPInstruments/rudder.cpp

4.26 Throttle Class Reference

The Throttle Control Class.

`#include <throttle.h>`Inheritance diagram for Throttle::



Public Member Functions

- float setData (char *dataType)

  Set Data is used to set the data that was read from the hardware control to a specfic flightSIM input. dataType is used in case of an control that has multiple inputs to the system.

- void openControl ()

  Open control is used to set the link to read from the hardware.

- void getData (int number)

  Get Data is used to read the data from the control. val is used in some cases where a control that has multiple inputs to the system.

Friends

- class CPInstrumentFactory

4.26.1 Detailed Description

The Throttle Control Class.

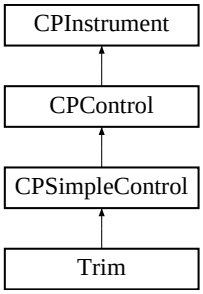
The documentation for this class was generated from the following files:

- CPInstruments/throttle.h
- CPInstruments/throttle.cpp

4.27 Trim Class Reference

Trim Wheel Class.

`#include <trim.h>`Inheritance diagram for Trim::



Public Member Functions

- float setData (char ∗dataType)

  Set Data is used to set the data that was read from the hardware control to a specfic flightSIM input. dataType is used in case of an control that has multiple inputs to the system.

- void openControl ()

  Open control is used to set the link to read from the hardware.

- void getData (int number)

  Get Data is used to read the data from the control. val is used in some cases where a control that has multiple inputs to the system.

Friends

- class CPInstrumentFactory

4.27.1 Detailed Description

Trim Wheel Class.

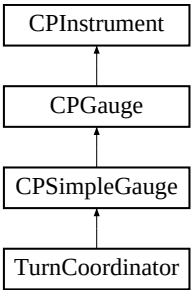
The documentation for this class was generated from the following files:

- CPInstruments/trim.h
- CPInstruments/trim.cpp

4.28 TurnCoordinator Class Reference

TurnCoordinator Gauge Class.

`#include <turncoordinator.h>`Inheritance diagram for TurnCoordinator::



Public Member Functions

- void setGauge ()

  Set gauge is used to set the data saved in the object to the hardware.

- void setHID ()

  Initializes the link to the hardware.

- void setData (int x, float i)

  if x is 1 set i to heading rate, if x is 2 set i to sideslip angle

Friends

- class CPInstrumentFactory

4.28.1 Detailed Description

TurnCoordinator Gauge Class.

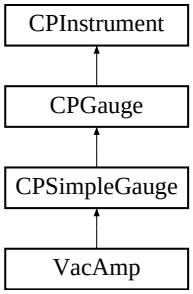
The documentation for this class was generated from the following files:

- CPInstruments/turncoordinator.h
- CPInstruments/turncoordinator.cpp

4.29 VacAmp Class Reference

VacAmp (Suction & Ammeter) Gauge Class.

`#include <vacamp.h>`Inheritance diagram for VacAmp::



Public Member Functions

- void setGauge ()

  Set gauge is used to set the data saved in the object to the hardware.

- void setHID ()

  Initializes the link to the hardware.

- void setData (int x, float i)

  saves the data that will be written to hardware in the object. It takes 2 inputs, data type is the data type to be set in case the device takes more than one input; dataVal is the value to be set.

Friends

- class CPInstrumentFactory

4.29.1 Detailed Description

VacAmp (Suction & Ammeter) Gauge Class.

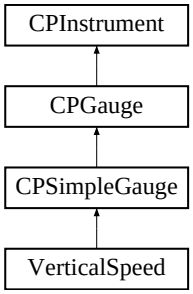
The documentation for this class was generated from the following files:

- CPInstruments/vacamp.h
- CPInstruments/vacamp.cpp

4.30 VerticalSpeed Class Reference

VerticalSpeed Gauge Class.

`#include <verticalspeed.h>`Inheritance diagram for VerticalSpeed::



Public Member Functions

- void setGauge ()

  Set gauge is used to set the data saved in the object to the hardware.

- void setHID ()

  Initializes the link to the hardware.

- void setData (int x, float i)

  saves the data that will be written to hardware in the object. It takes 2 inputs, data type is the data type to be set in case the device takes more than one input; dataVal is the value to be set.

Friends

- class CPInstrumentFactory

4.30.1 Detailed Description

VerticalSpeed Gauge Class.

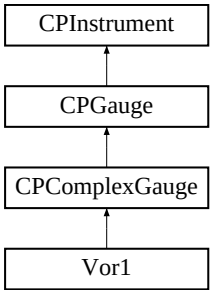
The documentation for this class was generated from the following files:

- CPInstruments/verticalspeed.h
- CPInstruments/verticalspeed.cpp

4.31 Vor1 Class Reference

Vor1 Gauge Class.

`#include <vor1.h>`Inheritance diagram for Vor1::



Public Member Functions

- void setGauge ()

  Set gauge is used to set the data saved in the object to the hardware.

- void setHID ()

  Initializes the link to the hardware.

- void setData (int x, float i)

  if x is 0 set on or off data, if x is 1 set ILS glidescpole deviation if x is 2 set ILS elevation deviaton, and if x is 3 set VOR angle

Friends

- class CPInstrumentFactory

4.31.1 Detailed Description

Vor1 Gauge Class.

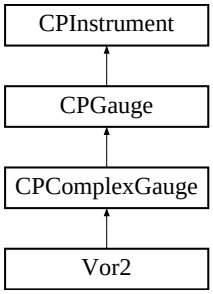
The documentation for this class was generated from the following files:

- CPInstruments/vor1.h
- CPInstruments/vor1.cpp

4.32 Vor2 Class Reference

Vor2 Gauge Class.

`#include <vor2.h>`Inheritance diagram for Vor2::



Public Member Functions

- void setGauge ()

  Set gauge is used to set the data saved in the object to the hardware.

- void setHID ()

  Initializes the link to the hardware.

- void setData (int x, float i)

  saves the data that will be written to hardware in the object. It takes 2 inputs, data type is the data type to be set in case the device takes more than one input; dataVal is the value to be set.

Friends

- class CPInstrumentFactory

4.32.1 Detailed Description

Vor2 Gauge Class.

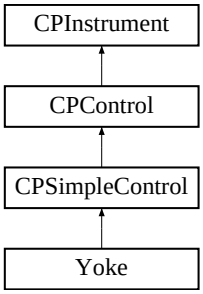
The documentation for this class was generated from the following files:

- CPInstruments/vor2.h
- CPInstruments/vor2.cpp

4.33 Yoke Class Reference

The Yoke Control Class.

`#include <yoke.h>`Inheritance diagram for Yoke::



Public Member Functions

- float setData (char ∗dataType)

  setData has 2 different possible input here: "latitude" to set the data read from the latitude motion; "longitude" to set the data read from the longitude motion.

- void openControl ()

  Open control is used to set the link to read from the hardware.

- void getData (int number)

  Get Data is used to read the data from the control. val is used in some cases where a control that has multiple inputs to the system.

Friends

- class CPInstrumentFactory

4.33.1 Detailed Description

The Yoke Control Class.

The documentation for this class was generated from the following files:

- CPInstruments/yoke.h
- CPInstruments/yoke.cpp

Index

Bibliography

Bonner, J. (2012). Learning to fly: A history of flight simulators. http://ca.ign.com/articles/2012/02/24/learning-to-fly-a-history-of-flight-simulators/. Last accessed: September, 2014.

cppreference.com (2013). std::atomic - cppreference.com. http://en.cppreference.com/w/cpp/atomic/atomic. Last accessed: September, 2014.

Gabbai, J. (2001). The Art of Flight Simulation. http://www.fermentas.com/techinfo/nucleicacids/maplambda.htm. Last accessed: September, 2014.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. (2009). Design patterns: Abstract factory - informit. http://www.informit.com/articles/article.aspx?p=1398599. Last accessed: September, 2014.

HID API (2014). Hid api for linux, mac os x, and windows - signal 11 software. http://www.signal11.us/oss/hidapi/. Last accessed: September, 2014.

Kanjilal, J. (2006). Introducing serialization in .net: Asp alliance. http://aspalliance.com/983_Introducing_Serialization_in_NET.1. Last accessed: September, 2014.

Kernel.org (2014). Programming input drivers. `https://www.kernel.org/doc/Documentation/input/input.txt`. Last accessed: September, 2014.

Leadbetter, R. (2009). Console gaming: The lag factor. `http://www.eurogamer.net/articles/digitalfoundry-lag-factor-article`. Last accessed: September, 2014.

Liu, J. (2014). Cockpit Display System Simulation of general Aviation Aircraft. *Advanced Materials Research Vols. 846-847(2014) pp 1893-1898.*

MacKenzie, I. S. and Colin, W. (1993). Lag as a determinant of human performance in interactive systems. *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems - CHI '93, (New York: ACM, 1993), 488-493.*

Moore, K. (2008). A brief history of aircraft flight simulation (flight training). `http://homepage.ntlworld.com/bleep/SimHist1.html`. Last accessed: September, 2014.

Page, R. (2000). Brief History of Flight Simulation. *The SimTecT 2000 Organising and Technical Committee.*

Part 60 Final Rule, Consolidated Version (2008). Federal register /vol. 73, no. 91 / friday, may 9, 2008 /rules and regulations - national simulator program (nsp). `http://www.faa.gov/about/initiatives/nsp/media/consolidated_version.pdf`. Last accessed: September, 2014.

Presagis (2010). Newsletter- vol. 1 issue 1 spring 2010. `http://www.presagis.com/resources/newsletter/embedded_graphics_newsletter_-_spring_2010/`. Last accessed: September, 2014.

Reisman, R. (1990). A Brief Introduction to the Art of Flight Simulation. *Ars Electronica 1990. Band 2. Virtuelle Welten. Hg. v. Gottfried Hattinger, Morgan Russel, Christine Schpf & Peter Weibel. Linz: Veritas, S. 159-169.*

Rider, J. W. (2004). What is client/server computing? `http://www.jwrider.com/lib/clientserver.htm`. Last accessed: September, 2014.

Thomas, G. (2014). Industrial ethernet book issue 4 / 29. `http://www.iebmedia.com/index.php?id=4070&parentid=74&themeid=255&hft=4&showdetail=true&bb=1&PHPSESSID=pvlvnt7286eoa6fv2dp04nn393`. Last accessed: September, 2014.

Xia, Y. (2012). Concurrency in c++. `http://www.cs.colorado.edu/~kena/classes/5448/f12/presentation-materials/xia.pdf`. Last accessed: September, 2014.

Yan, J. K. (1985). Advances in Computer-Generated Imagery for Flight Simulation. *IEEE 0272-1716 85 0800 0037$01.00.*