School District of Osceola County
University of Central Florida
Department of Electrical Engineering and Computer Science
Senior Design Project

VOLTES FLY

11

Group # 3

Hector Bermudez

Manuel A. Arredondo

Joe Paolini

Table of Contents

1 Executive Summary

Voltes Fly is a low cost flight simulator which allows the user to take part in a virtual world and giving you the ability to fly several airplanes like the Cessna and other smaller aircraft. The interfacing system consists of the joystick, throttles, rudder pedals, a motion platform with comfortable seating, and a computer with 2-3 screens or projector. These components all work together to give the pilot the realistic feeling of being immersed into the flight experience. The system creates the feeling of pitch and roll as felt in real airplanes. As the airplane model is moved within the computer software, the algorithms, accurate movement, and tracking will make the person feel as if they are really flying. This project will be completed by December of 2011.

Our group decided to implement this project for several reasons. Being computer engineers with more of a software background, the team agreed to work on something where one would gain experience with hardware design and at the same time have the chance to exercise our programming skills in both high level and embedded system environments. Voltes Fly is exactly what the group was looking for; it is both challenging and rewarding, with lots of fun flight hours to be done. As a side effect of this project, the team will learn more about the aviation industry and what it takes to fly an airplane. A lot of calibration and testing is needed for the subparts of the simulator. However; after all the hard work, the result will allow the pilot to escape the world for a while and experience flight from the comfort of their home and even better, without having to leave the ground. Voltes Fly is just what the group were looking for, it is both challenging and rewarding, with lots of fun testing to be done. It lets the user escape the world for a while and experience flight from the comfort of their home or even better, without having to leave the ground.

The joystick and throttle will be designed and assembled by our group, and it will send the position information to a main component called the Control Loader [CL]. The control loader is a piece of hardware which allows for communication between the joystick, throttle, motion base, and host computer through one USB port. Not only does the joystick control the plane's position, but it has a built in force feedback similar to the rumble effect on most console systems available in the market today. Whenever the user moves the joystick handle, internal motors will be triggered to create this realistic effect. The motion platform itself moves in manner in which an illusion of reaching steeper angles is created. In reality the actual angle the seat will tilt will be much lower. This behavior enhances the experience of the user while keeping their safety in mind. On the other hand, rudder pedals will be acquired as a COTS item. On the computer side, a specialized program is provided to the user. This program works along with the flight simulator software and it simplifies the set-up of the virtual world. It will allow the trainer to control various parameters in the simulation run, from weather conditions to sound. This feature is a great tool for trainers that want to sit on the side and watch the student fly through the parameters he/she set for them.

There are several companies and hobbyists dedicated to flight simulation in the Orlando area, what sets Voltes fly apart from the rest is that most of the parts are being designed by the members themselves(except for the platform itself). Most people in the field just buy all the input peripherals from large companies. However, building the components ourselves creates a system that is more compact, integrated and gives us enough freedom to add special functionality and shape the interface devices to our liking. Also, the motion platform being used is one of the cheaper options (two degrees of freedom) already giving us a financial advantage over others whose project's budget can go way above $20,000.

2 Project Definition

2.1 Motivation

The idea about building a flight simulator was born when the group noticed a huge gap between the training received by the military or others high budget companies within the commercial aviation businesses and those who because of money, cannot access to a more realistic training. In addition, there is thousands of avid enthusiastic out there desiring to get a more realistic flying experience expending just a reasonable amount of money.

Last semester one of our team members went to Washington DC, and he visited some of the Smithsonian's museums. In one of them, he played with a flight simulator and shared that experience with us. Suddenly, the challenges of designing our own system while achieving a low budget were discussed. It fared challenging at first, and a lot of questions emerged as the system was imagined: How do joysticks work and how is force feedback achieved? How do hardware peripherals interface with flight simulator software? What algorithms are implemented on development boards to achieve synchronized movement with motion platforms? How hard would it be to build it ourselves? These are just some of the questions which caused enough curiosity within the team and inspired us to take on the project. Hoping to learn the answers to these questions and become knowledgeable in the field, the group embarks this journey with excitement.

2.2 Goals and Objectives

There are several goals that were set regarding the system. Overall all the components should work together seamlessly without any noticeable error. The group wants to provide the user with an entertaining experience, and for pilots; a form of training without having to fly an actual plane. Also, the simulator should be good enough to compete with other low cost systems available online. Our system shall be broken up into 6 sub components as seen in **Figure 2.2.1.** For each of these components, all communication with the main CPU must be done through USB communication. Additionally, there are specific goals related to each of these. The Interfacing System is composed of several components mostly designed by the group.

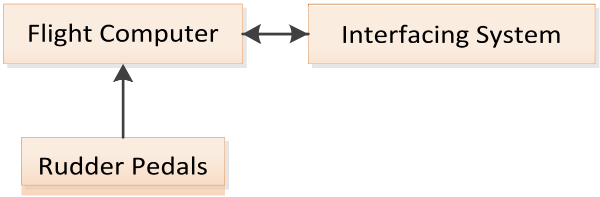


Figure 2.2.1 Flight Simulator

Rudder Pedals: This piece will communicate independently from the other systems as shown on the diagram above. It will be purchased off the shelf and communication will be carried out through its USB connection. The outside frame must be strong enough to protect the circuits inside from the force coming from the pilots' feet. The reaction time must be small and have a plug and play capability. Each left and right pedals must effectively send digital signals that the flight computer can interpret.

Interfacing System: The main objective of this system is to provide realism to the virtual flight experience through input and output devices. There are three components within this system which provide communication between the pilot and the simulator as seen in **Figure 2.2.2;** these devices are monitored by an electronic component called the control loader (CL).

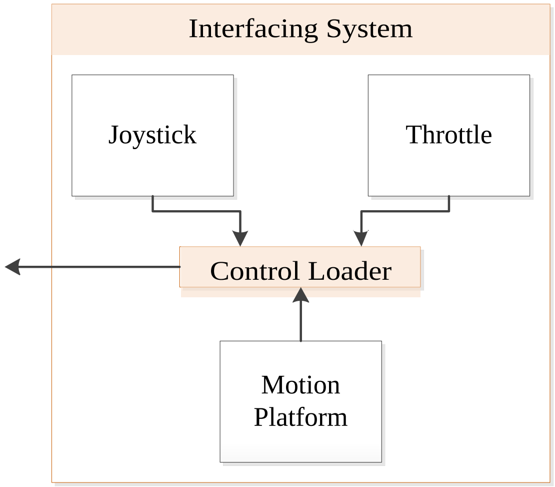


Figure 2.2.2 Interfacing Components

Any information coming or going to the flight computer will be arranged and controlled by this unit. These devices are now explained and their goals and objectives are defined:

Joystick: The objective for this hardware peripheral is to be able to translate the user's hand movements into digital X and Y coordinate values that can be sent to the flight computer. The reaction time has to be low and the moving parts should cause as less friction as possible so that the device can have a significant lifetime. The outside frame design can either consist of an arcade style joystick or a more realistic Yoke design. As far as the force feedback, the goal is to have several different levels of intensity available.

There is flexibility regarding the type of feedback to be experienced: a static feedback which responds to the force applied by the pilot, or a dynamic one which deals with events within the game. In the second case the software and hardware interface must provide enough software events to experience the effect to the maximum. The rumble or force caused by this effect should not affect the joystick mechanical or electrical structure in any negative way. If static feedback is implemented, the circuitry must control the operation of the motors accurately enough so that it stops rumbling once the user has let go of the stick. In conclusion, the flight computer will expect values from 0-100 for both the x and y axis of movement.

Throttle: It is only necessary to implement one throttle to support the kind of planes to be simulated. It will be placed to the right of the joystick module. The user must be able to set it at a value and the throttle must stay in place until the next interaction. This lever component will allow for reversing the engines as well as turning them on from zero to their max value. Any moving parts (gears etc.) must experience the least friction possible and most also have minimum lag when interfacing with the software. No buttons or flip switches will be implemented; however, the possibility is there if time allows. The value expected by the flight computer is an integer from 0-100.

Control Loader: The motion controller unit must effectively output the right voltages to the servomotors according to the pitch and roll values entering from the flight computer. The algorithms implemented must be able to realistically translate these values and have as little lag and as little awkward movement as possible. Just like the other components, the parts used for this subsystem must be cost efficient. It must effectively pack and send both joystick and throttle information, simultaneously receiving the pitch and roll values from the flight computer.

The flight computer will contain a crucial software component, which will control several aspects of the flight simulator. It will help all the different systems work together. This system is the Instructor Operator Station and its goals are described below.

Instructor Operator Station: The goal of the Instructor Operator Station (IOS) is to allow an instructor to control simulation setting such as environment and starting airport runway. The IOS is the only user interface to the simulation software. The IOS will also include modules responsible for sending and receiving packages of data from the Control Loader for the motion base, throttle, and joystick. Included with the IOS will be a Flight Test page which will allow the viewing and setting of flight data parameters during an active simulation for the testing phase of the project.

2.3 Scope of Work

Flight Simulators have been around for some years already, but they usually present an expensive solution for those who often use them. Companies and enthusiastic fans are always trying to build their projects at a lower cost without compromising other features like safety, reliability, accuracy, and scalability. Our project will achieve all these functionalities at a lower cost, making possible the reproduction of this project by whoever is interested.

2.3.1 Objectives

The simulator to be built should give both pilots and non-pilots a satisfactory experience. The group members want the user to be immersed in a virtual reality world and feel like they are off the ground even though they are inside a building. Most importantly, the objective of this project is to impress the board which will be seeing Voltes Fly perform in December 2011; consequently allowing our group members to graduate.

2.3.2 Tasks

Even though each part of the project is split across each group member, all will contribute in the research, design, and prototyping of all the components to some extent.

Task	Member Responsabilities
IOS- Instruction Operator Station	Joe Paolini
Hardware/Software Interface	Joe Paolini/Hector Bermudez
Control Loader	Hector Bermudez
Throttle	Manuel Arredondo
Joystick	Manuel Arredondo

Table 2.3.1 Member Responsibilities

2.3.3 Schedule

The project should be done in about four months. The assembly and construction will begin by August 22nd and it should be finished by December 5th. However; it will be ideal to finish the construction and assembly of all components a month preceding December 5th in order to allow enough time for testing.

2.4 Significance

The fact that the group has limited resources is also the key to demonstrate that a project like this can be built with a small amount of money. In the great scheme of things, the group hopes that this project will leave a mark on the commercial and/or military aviation businesses because even though it is understood that this project will not be implementing several features due to time restrictions, they could be built in later versions. For instance, later versions could implement more degrees of freedom on the motion base platform, vibration effect for the chair, more programmable buttons and sliders, and recreate every type gauge on the market. In addition, future projects could also add G forces and enclosed cockpits.

In a more personal point of view, this project will give us the opportunity to apply most of the knowledge acquired in college. Finally, the group members will have some real world experience as engineers by building a project that the has to be put together in about four months without knowing most of the details and the necessary know-how to successfully accomplish it. By December, 2011 the whole team should have a more clear idea and a better understanding about embedded processors, electronic system design, software design, hardware/software interfacing, real-life projects, and the development process of complex systems.

2.5 Budget Projection and Financing

Even though initially, the group had Mr. Wayne Hilmer as the sponsor, it was decided to borrow the motion base platform from Servos and Simulation Inc., so now the plan is to split the rest of the expenses for the project equally. It is expected that the expenses are going to be around $1000 to $1600.

Parts	Quantity	Expected Cost (in dollars)
A PC with Windows XP Pro OS installed.	1	$0.00 – One of our team members will provide the computer.
Motion Base Platform	1	$0.00 – Borrowed from Servos and Simulation Inc.
Development Board – MSP-EXP430F5529	1	$150.00
Rudder Pedals Set	1	$0.00 -- Loaned by Corsair Engineering.
Displays 17 – 19”	2	$230.00
Joystick Components		Currently up to $81.00
X-Plane	1	$30.00

Table 2.5.1 Budget Projection

2.6 Specifications

2.6.1 Overall Specifications

The final product should be portable, light weight and easy to use. The user will have the ability to choose the aircraft of his preference from the available list on the X-Plane simulator. The front external windows will be recreated by three displays of approximately 19 inches each. By having this configuration, the “pilot” should achieve the desired 120 degrees of sight. All the necessary changes and/or adjustments to the software simulator are going to be made by the Instruction Operating System or IOS. In addition, the system should be able to send readable information from the Joystick trough USB to the Control Loader (CL); this information contains the position of the plane on the X and Y axes at any given time. The CL will be the main unit of our system, every piece of information and/or command will go through it.

Moreover, the system should be able to read and interpret the pitch and roll variables coming out of the software simulator. As before, this information will be sent using USB communication. Once the CL captures these variables, it will send the appropriate commands to each servo-motor, accomplishing the desired motion-platform position. In case of specific events like a plane crash, turbulence or landing, the system should reflect those actions by recreating a rumbling on the joystick end in order to achieve a higher level of reality.

The system also includes a throttle and rudder pedals. Even though the pedals are being purchased off the shelf, the throttle, on the other hand, will be built from the ground up using an analog signal from the throttle to the CL.

2.6.2 Joystick

The joystick will be built from the scratch. It will be made using two servo motors that will create the force feedback effect. Two single-turn rotary potentiometers will be used to track the movement of the X and Y axis. Also, several mechanical parts will be used for the system: universal joint and shaft for the handle along to allow for user input, a system that may center the handle automatically to its origin. Also, a decent range of motion will be allowed so that a rotation of at least 45 degrees in each direction can be achieved.

2.6.3 Throttle

The throttle's communication will be ideally through USB as well, and the mechanical system will be attached to a rotary potentiometer. The rotary potentiometer will send the analog signal to either a Serial to USB connector or to a Microcontroller-firmware for a USB protocol.

2.6.4 IOS

The Instructor Operator Station (IOS) provides communication between the other systems (Control Loader, X-Plane, and Rudder Pedals). It will be broken into four modules: X-Plane Communication, Control Loader Communication, Rudder Pedal Communication, and IOS User Interface.

The X-Plane Communication module will on a dedicated thread and controls the bi-directional communication between X-Plane (contains the flight physicals model, and Image Generator (IG)). The bi-directional communication will be implemented utilizing the User Datagram Protocol (UDP) via a local loop back (IOS and X-Plane run on same machine). The IOS will send the following data to X-Plane: roll, pitch, and weather settings. In return the IOS will also receive those values from X-Plane.

Control Loader Communication module will run on a dedicated thread and controls the bi-directional communication between the Control Loader and the IOS. The bi-directional communication will be implemented using a subset of the USB protocol. Data will be sent and received using data packets as the payload in a USB Data Packet. The IOS will send the following data to the Control Loader: roll and pitch. In return the IOS will receive the following from the Control Loader: joystick position and throttle position.

The Rudder Pedals is a COTS item and contain a driver to handle communication via USB between the Rudder Pedals and the Flight Computer.

The IOS User Interface module handles the input from the instructor and controls the communication between the X-Plane Communication and Control Loader Communication. When the IOS is launched, the User Interface (UI) will be on a dedicated monitor and provides the instructor with the following options: start simulation, end simulation, set environmental parameters, and view flight parameters. It is the primary interface for communication between an instructor and the system.

2.6.5 Control Loader

To be changed

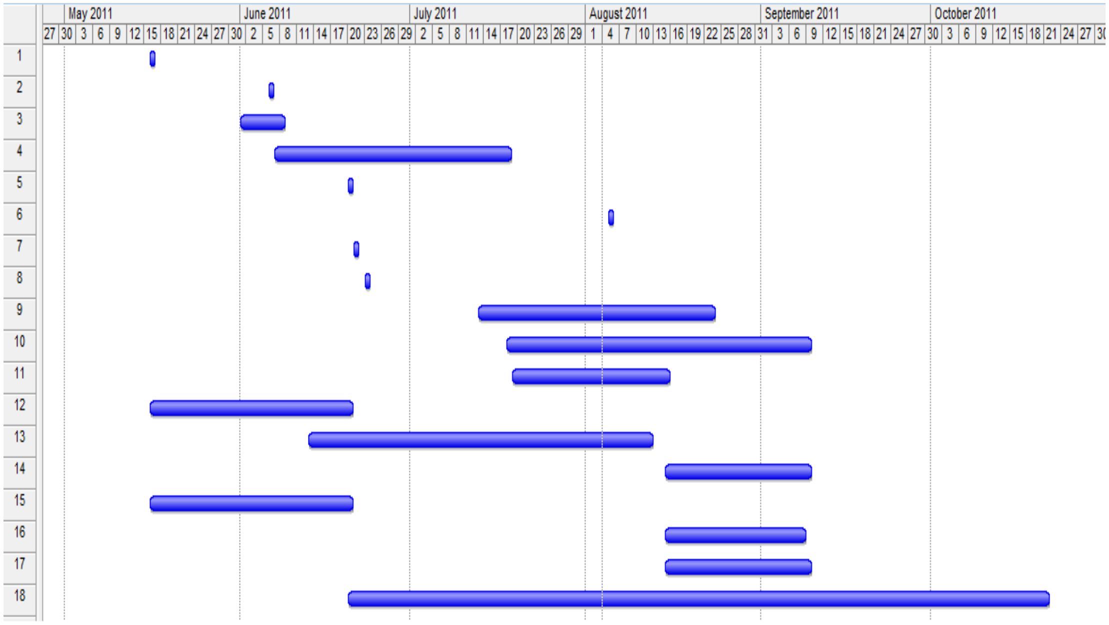
The Control Loader will have a development board with a USB port in it. Once a signal is received from the simulator a microcontroller will process it. A protocol has to be created to provide an interface between the USB port and two DACs (Digital-Analog Converters).The plan is to use either an Ultra-Low-Power MSP430 or a Stellaris Family Microcontrollers. The DACs then, will send a voltage value in the range of +/- 7V to the servo-motors. The individual parts can be seen on detail in Figure **2.6.5**

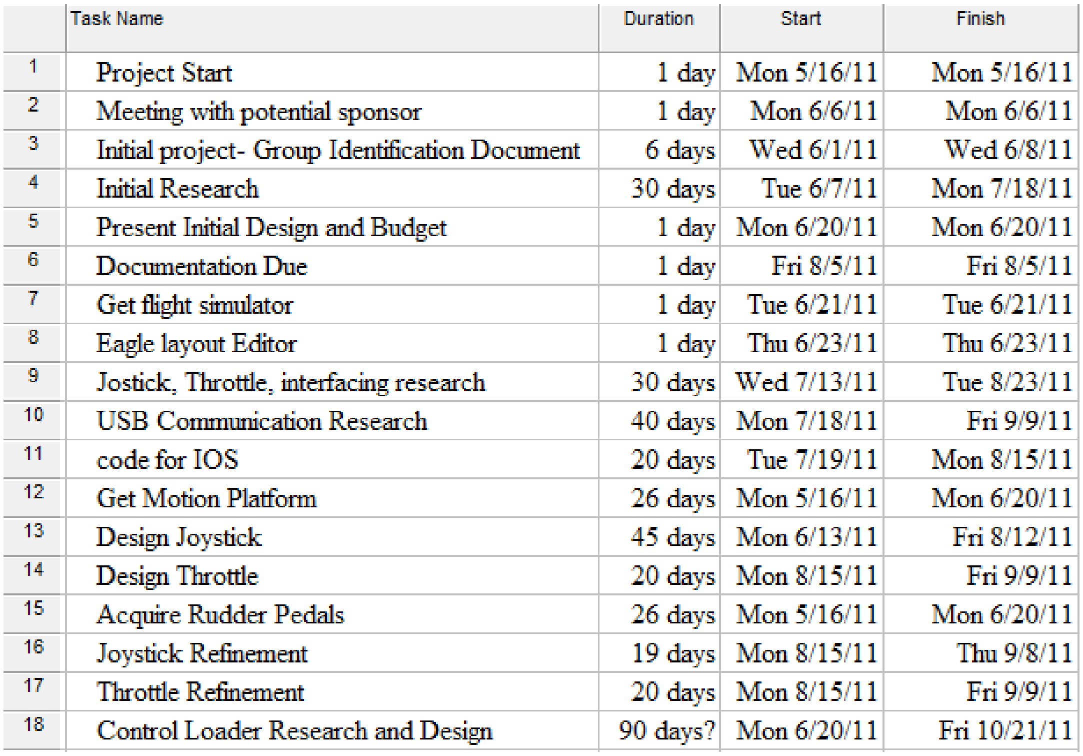
2.6.6 Motion Base

The Motion Base was leased from a local company in Longwood, which name is Servos and Simulation, Inc **[2.6.1]** Voltes Fly will incorporate this motion platform on the picture for many reasons. The first reason is, of course, cost. One of this motion platforms' price could easily be on the thousand dollars, so the group is saving a lot of money by using it without cost. The second reason is about the specifications. This unit meets all the requirements the group had in mind in order to build the system effectively.

2.7 Milestone Gantt chart

After taking into account all the tasks required to accomplish the goals set for this project and the time that every task would take, the Milestone Gant chart in **Figure 2.7.1** was developed.





	Task Name	Duration	Start	Finish
1	Project Start	1 day	Mon 5/16/11	Mon 5/16/11
2	Meeting with potential sponsor	1 day	Mon 6/6/11	Mon 6/6/11
3	Initial project- Group Identification Document	6 days	Wed 6/1/11	Wed 6/8/11
4	Initial Research	30 days	Tue 6/7/11	Mon 7/18/11
5	Present Initial Design and Budget	1 day	Mon 6/20/11	Mon 6/20/11
6	Documentation Due	1 day	Fri 8/5/11	Fri 8/5/11
7	Get flight simulator	1 day	Tue 6/21/11	Tue 6/21/11
8	Eagle layout Editor	1 day	Thu 6/23/11	Thu 6/23/11
9	Jostick, Throttle, interfacing research	30 days	Wed 7/13/11	Tue 8/23/11
10	USB Communication Research	40 days	Mon 7/18/11	Fri 9/9/11
11	code for IOS	20 days	Tue 7/19/11	Mon 8/15/11
12	Get Motion Platform	26 days	Mon 5/16/11	Mon 6/20/11
13	Design Joystick	45 days	Mon 6/13/11	Fri 8/12/11
14	Design Throttle	20 days	Mon 8/15/11	Fri 9/9/11
15	Acquire Rudder Pedals	26 days	Mon 5/16/11	Mon 6/20/11
16	Joystick Refinement	19 days	Mon 8/15/11	Thu 9/8/11
17	Throttle Refinement	20 days	Mon 8/15/11	Fri 9/9/11
18	Control Loader Research and Design	90 days?	Mon 6/20/11	Fri 10/21/11

Figure 2.7.1: Time-Activity relationship

3 Preliminary Research

3.1 Implementation Methodology

3.1.1 Project Management

Management can be a key factor for a project to either fail or success. A good management is definitely an essential part of every serious project, so the team feels that having a good strategy to approach all of the possible challenges on the project is not only critical but also essential for success. The team decided not to have a single project manager; instead, each person will have a set of responsibilities based on their previous knowledge and backgrounds. This does not mean that they will be by themselves or exclusively working on those specific tasks, but this is necessary in order to apply the divide and conquer mechanism. Every team member needs to achieve their goals keeping in mind the global scope of work, budget and due dates.

Since the team has a very short amount of time to accomplish its goals, it was decided to follow the four steps of an already proven methodology of traditional software development, which are **Requirements**, **Architecture and Design**, **Development**, and **Test and Feedback**. The team will follow these steps even when it is understood that some of the tasks are going to be running in parallel or with some level of overlap. The requirements stage is when the whole team gets together and find out the initial magnitude of the project and all possible ways to approach the problem. Next, during the architecture and design phase is when the team draws the project's overall layout; then, it is broken down into smaller pieces for a better understanding. Here is when the assignment of individual responsibilities comes into play. Basically the initial architectural and conceptual designs are defined during this stage. The development phase is one of the most interesting ones; here the ideas and thoughts discussed during the previous stage are materialized and a prototype is the result. During the final stage, the prototype goes through an exhaustive inspection of every subsystem and the interaction of them as a whole. Any problem detected will be categorized in one of these sections: functionality issues, interaction incompatibilities, and design issues. As it is seen, the previous sections are going from the least serious problems to the worst case scenario. After several iterations during the last three stages, a final prototype is presented.

3.1.2 Research Methodology

Once individual responsibilities were assigned, an exhaustive research of individual subsystems has to be conductive by the team member in charge of that specific functionality. The research emphasis has to be on not only hardware components but also on software functionalities. Motion platforms, joysticks, rudder pedals, throttles, USB/Serial communication ports, microcontrollers, DACs are just few examples of how intensive the research should be. Every team member should get different versions or models of specific parts in order to compare their technical specifications and if is following the cost-effective principle. By doing that, the group should determine the right part to do the job at the lowest price. The biggest percent of the information was gathered mostly from internet, our sponsors, and local stores dedicated to sell electronic parts and accessories. After the group had all the information needed, it was easier to decide what part was the better one based on cost, and technical specifications.

3.1.3 Design Methodology

Since the team realized from the beginning how important and crucial a good design is, it was decided to dedicate as long as the group could to this stage. During this time, personal skills and individual backgrounds played an important role. The whole team was immersed in a brainstorming figuring out the best design for individual part and the safest way to interact with others. The communication between the whole team was really important during this phase. Since most of the interfaces were designed during this time, several meeting were needed and a shorter time between them. In addition, the creation of several diagrams explaining in detail functionalities and data flows was really important since helped to understand the interaction between the parts and also was useful to recognize internal weaknesses.

After this phase was concluded, it was understood that without walking through these steps, the development process would have been longer. Since many developers or engineers feel the necessity to start building their creations, our group is encouraged them not to do it and instead to take the time to follow the steps required by this stage.

3.2 COTS Flight Simulation Software

Currently in the market there is a wide variety of commercially available off the shelf (COTS) flight simulation software available. The three most widely used options are: Microsoft Flight Simulator X, Flight Gear, and X-Plane. Each option has its pro’s and con’s for use.

Microsoft Flight Simulator X has in use for approximately 25 years and this is a testament to the quality of the software. One factor that really persuaded the group to focus on Microsoft Flight Simulator X was the length it was in production. The online support for this software is vast and there are plenty of resources present where if needed help could be obtained during the design/build phase. In order to extend the simulation software, Microsoft provides the SimConnect Software Development Kit, which will allow the IOS have control over environmental variables, along with allowing the use of the group's custom hardware. Unfortunately in 2010, Microsoft discontinued support and future production of the simulation software. Due to the cancellation, the group felt that this could impact support therefore was no longer a viable solution in providing a flight physics model and image generator (IG).

The second option is X-Plane which is popular among hobbyists and commercial companies. Some of the factors that make X-Plane enticing are the cost ($30), support, and positive feedback from current/past users. The low cost helped contribute to the popularity, which in turn contributes to the amount of software support to aid in interfacing the IOS with X-Plane. One of the downfalls is that X-Plane does not directly support the logic involved with provided force feedback to the joystick, however there is a third party plugin that will provide the functionality. This is not ideal since there is no guarantee regarding the level of functionality related to the force feedback. The initial design for the project is simulate force feedback by providing a static amount of force to the joystick.

Flight Gear the final option for simulation software. The major pro of using Flight Gear is that it is open source under the General Public License. This would allow the group to directly modify the anything in the simulation software in order to meet the required needs. Another factor that makes Flight Gear enticing is the cost which is free and this allows all team members the ability to work on it without having to purchase multiple copies which is the case with Microsoft Flight Simulator X and X-Plane. The con of Flight Gear is that it is that it is not very intuitive and quality of the scenery is lacking when compared to the other options.

The decision was narrowed down to X-Plane and Flight Gear mostly due to the high expense of Microsoft Flight Simulator X. After some testing it was decided that X-Plane was the simulation software of choice. It meets all the required criteria for the project and offers the most for the price along with a simple communication method via UDP.

3.3 Flight Computer

The flight computer is the most important COTS component in this project. Without it, there is no simulator at all. This is where the flight simulation software is run. It must have enough RAM to support both the flight simulator software and the IOS software application. In order to save money and meet our budget goals, a computer was donated by group member Hector Bermudez. This desktop computer has the following specifications: it has a dual core processor **AMD Athlon 64x2** running at a rate of **2.51 GHz**, with a **Nvidia Geforce 6150SE nForce430** graphics card, and **2 GB of RAM**. If necessary, additional memory will be added, which will cost 30-50 dollars.

3.4 Displays

Ideally four displays would be used to provide the optimal simulator experience to the pilot. Three displays would be used to provide the Out The Window (OTW) view to the pilot and the fourth display would be for the IOS. This setup can become very costly, because it requires the use of more components. If only a single flight computer is used, then in order to have four displays, either a new graphic card which supports four displays must be bought or a secondary graphic card that will supplement the pre-existing. Due to budgeting constraints it was decided the most efficient option would be to only support two displays, one for the IOS and the other for the OTW. The IOS display will be an LCD display while the OTW view will be a projector.

LCD screen: The screen for the instructor was picked for its cost effectiveness, the group will be using the **Acer- Factory- Refurbished 18.8 inch Widescreen TFT-LCD Monitor**. The color of the monitor purchased will be black so that it matches with the rest of the flight simulator. It can be purchased at Best Buy for $69.99.

Projector: The projector used to display the OTW view is the **Optoma- Factory refurbished XGA DLP** which can be found at Bestbuys' website for a total price of $399.99. If it is possible, the group would like to avoid this expense by finding a contributor who is willing to let us borrow one effectively cutting our expenses down.

3.5 Audio System

The audio system to be implemented is an important part of the flight simulator. Without it, the experience would not be the same. Not only must the computer speakers used along the flight computer provide an enhancement to the overall experience but they must also take into account the initial budget restrictions of the project. While it is a COTS item, research was done to ensure that the right choice was made when purchasing it.

The first audio system which was looked into was the **Raygo R12-41405 USB Multimedia Speakers**. It was found in the CompUSA website, and consists of two 2.5inch speakers which are USB powered. What stood out the most about these was their small size and price. Their lightweight property makes them easy to mount on each side of the pilot seat and at a price of $9.99 they get the job done. The con is that it uses USB connectivity, which will be the main source of connection between the other systems and the flight computer. The group doesn't want another peripheral getting in the way of these ports, just in case the final design needs more than one USB port to operate. Therefore, our next option was a **Corsair SP2500 Gaming Audio Series High-power 2.1 PC Speaker System** offered by the same website. The first positive aspect of this option is that it was designed specifically for gaming. It comes with a remote control with volume and bass control. This control would be operated by the "trainer" or supervisor using the IOS software to set up the flight computer. Instead of just two speakers, this system comes with a very big subwoofer. The group had the idea of placing the subwoofer as close to the mounting platform and the pilot seat as possible. If the placement is done correctly, the sound waves from the base can cause a noticeable amount of vibration into the pilot seat. This would create another way of adding realism to our simulator. The complete set is sold at a price of $229.99 which is a bit expensive for the groups liking. Also, the side speakers seem to be very heavy and might cause problems when trying to incorporate them with the platform and seat contraption. The same company offers a more economical option: **Corsair SP2200 Gaming Audio Series 2.1 PC Speaker System**. For $69.99, it offers a subwoofer of roughly the same magnitude as the more expensive option. While it does not output the same amount of power; several online reviews were read and offered positive feedback with regards to its quality. The two audio speakers are smaller and lighter, and seem to be just the right size for mounting and portability. This is important because the simulator's location might have to be changed several times during development and for the final presentation.

The possibility of using the group member's personal audio systems was considered; however, these speakers were for the most part worn out and did not offer the desired quality of sound. Therefore, the corsair SP2200 was chosen for this project. Its economical impact on the project is quite low, when compared to their quality of output, portability, and also their positive aesthetic impact on the simulator.

3.6 Pilot Seat

There are several goals in the minds of the group members with regards to the seat used for the project. The design of the seat must demonstrate that the following characteristics are taken care of:

- Enough cushioning and safety of user
- Aesthetically pleasing
- Durability

When choosing the seat (COTS item), the group is guided by their cost-effectiveness, and their dimensions, which are critical because they must be able to fit on the motion platform without any problem. The seat has to be safely secured and attached to this platform. Several options were explored, and the results are as follows:

A company called playseats .com was found to have several options for flight simulator seats. These are commercially used and have very high prices. One design that inspired the group consists of a cockpit style design with supports on each side of the chair, allowing the user to mount both the joystick and throttle. The bottom of the chair has steel supports with openings for bolting to the platform. It is being sold for 427 euros which equals a value of approximately 600 dollars. While this option is out of our reach, it gives the group an idea of how to mound the peripheral devices into the system. Playseat.com also offers a design that is derived from an office chair, which also sold at an expensive price. At this point, the group noticed that it will be rather easy to buy a generic office chair and adapt it for this project. OfficeMax's website was searched, and a variety of chairs were looked at. All one has to do, is disconnect the seat from its legs or wheel based support. Basically, the group can use the bolts and openings already manufactured into the bottom of the chair to attach it to the platform. The lowest priced office chair available on their website is being offered for $29.99 but does not offer any arm support. Looking around some more, the group was able to find the following option:

The OfficeMax **Adelfina High-Back Executive Chair** has an aesthetically pleasing design. Its black which will match the color of the motion platform, and it provides two layers of seat cushioning. Each side has arm support with leather padding, and it is available for less then a hundred dollars. It can be purchased for $69.99 on their website. The group came to the following conclusion: if there are no volunteers found that will donate an office style chair for the simulator, the Adelfina chair will be the one to be used as a pilot seat.

3.7 Integrated Development Environments

3.7.1 IOS

There are two choices of Integrated Development Environments (IDE) that can be used for development of the IOS software: Visual Studio 2010 Professional and Visual Studio 2010 Ultimate. It was narrowed down to these two choices because it has been decided that the IOS will be written in C# .NET which is ideally developed using Visual Studio (version independent), although it doesn't have to be. Using Visual Studio 2010 Professional provides all the functionality needed to develop the IOS software such as a robust debugger and an ideal interface which provides a clean organized overview of the project. All the features present in Visual Studio 2010 Professional are included with Visual Studio 2010 Ultimate, with the decision being the later, because it provides the ability to auto-generate Unified Modeling Language 2.0 (UML) compliant diagrams and Dependency Validation. This feature will be very useful during the build phase of the project. If any code needs to be changed, it allows for quicker updates to the documentation which results in a more streamlined development process.

3.7.2 Control Loader MSP430

After making the decision to develop and test the project using the MSP430 microcontroller, the next logical step is to take a look at the available software support. The software environment used for development must be one that all the group members feel comfortable with using. It must have features that can aid, and eventually cut time in the embedded software development process. This will ensure the best possible outcome. There are two main integrated development environments available directly from TI which are most popular with this controller: the IAR Embedded Workbench, and Code Composer Studio. These workbenches support a wide range of MSP430 controllers. This makes them ideal for this project, because the testing phase of the different subsystems take place in the Launch pad development board which contains a different MSP430 than the one to be used in the final version of the control loader.

The IAR embedded workbench has a C/C++ compiler for the controller, example projects, and run time libraries with plenty of online support. It has plenty of example files for different configurations of the controller **[3.7.1]**. Texas instruments provides a 38 page technical user guide that explains a typical project flow using this workbench, with a typical blinking LED project with step by step instructions. The kick start version comes with a simulator that can be used independently of the dev. Board or microcontroller. This would help the group with splitting the work and being able to work away from the hardware side. Once the code developed is tested and its output verified within the simulator, the code can be modified to be able to work on the actual piece of hardware. The testing phase of the project will be much easier with this simulator.

Code composer studio is now in version 4, and it provides a high level environment with the look and feel of the Eclipse IDE **[3.7.2].** This is already a feature which grabbed the attention of the group, due to the fact that all the members have experience and feel comfortable using Eclipse. Also, two of the members have already used this software for simple LED blinking projects using the Launchpad. Another factor which leads to this option is that there is a plugin called Grace which will be of tremendous help when developing the firmware. Grace is a graphical Peripheral Configuration tool which helps the developer set up the different hardware IC's to achieve whatever it is he/she needs to accomplish**[3.7.3]**. It provides a graphical user interface where the different components can be selected, showing the different configuration options available. It will then demonstrate how these configurations can be set up using code. For example, if the user wants to set up different pins for output or input; this tool can help generate code that will do so. Also, it will aid the user when there's a need to use analog to digital or digital to analog converters in the device. It displays all the options available and the steps needed to use the device, removing the need to read through pages and pages of PDF documents to learn the process. With this tool, the time spent in configuration is cut down and the developer can focus in the bulk and more important part of the firmware. It is this tool alone that convinced the team to use CCSv4 as the main software environment to develop the control loader's firmware.

3.8 Motion Platforms

Even though the motion platform does not need to be designed by our team, it is really important to choose the right one. The motion platform plays a decisive role on the system since the whole simulation experience is related to this platform. After the first Google searches, the team agreed to use platforms of two degrees of freedom. These platforms have the specifications required for the completion of the project as well as being simple enough to facilitate our work.

3.8.1 CKAS "T" Series 2DOF Motion Platform

The first option was to use the platform offered by **CKAS Mechatronics PTY LTD [3.8.1]**, a company located in Australia. After the company approved the copyright permission, the group obtained access to its specifications and a quote.

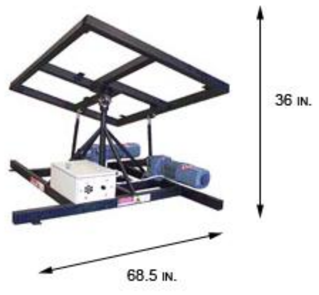


Figure 3.8.1(a) Standard units CKAS "T" Series 2DOF Motion Platform

System	CKAS T5 2DOF Motion System	CKAS T10 2DOF Motion System
Architecture	Pitch-Roll Crank Arm	
Actuation	Fully Electric	
PC Interface	USB 2.0	
PC – Controller Update	100 Hz	
Power Requirements	220 – 240 VAC 1ϕ – 7 amps	220 – 240 VAC 1ϕ – 12 amps
Approx Dim’s (parked)		
Width	1740mm (68.5”)	
Length	1740mm (68.5”)	
Height	915mm (36”)	915mm (36”)
Approx unit weight	220 kg	350 kg
Payload Mass	450 kg (1000lb)	900 kg (2000lb)
Payload Moment of Inertia	250 kg.m^2	400 kg.m^2
Angular Axes	Displacement , Velocity , Accel	Displacement / Velocity / Accel
Pitch	$\pm 18^o$, $\pm 30^o$/S , $\pm 300^o/s^2$	$\pm 12^o$, $\pm 20^o$/S , $\pm 200^o/s^2$
Roll	$\pm 18^o$, $\pm 30^o$/S , $\pm 300^o/s^2$	$\pm 12^o$, $\pm 20^o$/S , $\pm 200^o/s^2$

Figure 3.8.1(b) Standard units CKAS “T” Series 2DOF Motion Platform

Really soon the group found out that the price of the platform plus shipping and handling supersede the budget for the whole project, so even though this platform met all the requirements, the cost was excessive. Therefore, it was rejected immediately.

3.8.2 2DOF Motion Platform

After a few days making calls and searching the Internet, the group found this company located in Longwood that was engaged in this business for years, **Servos and Simulation Inc**. They had a slightly different model of two degrees of freedom platforms but with a more efficient design. As shown on **Figure 3.8.2** the position of the servos and its arms are located in parallel. With this design the Loader Controller can send the same pitch and roll signals to both servo-motors with different polarities, making possible a simpler and more robust implementation on the software side. For instance, if the aircraft is going a certain amount of degrees rolling to the right, the servos-motor on the right will have a negative value of -5; meanwhile the servos-motor on the left will have a positive value of 5.

This company was willing to provide us the platform until the end of our project in exchange for our services in the maintenance of their corporate website and in the assembly of some of the platforms that were under construction at that time. The group agreed to have several meetings with this company in order to work around our schedules, and to discuss the terms of use of the platform. But at the end of the day, based on the significant amount of money to be saved, the team decided to take this option and proceed.

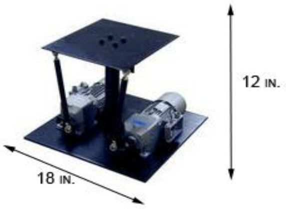


Figure 3.8.2: Motion Base Platform

	710-2-500-220	710-2-1000-220	710-2-2000-220	710-2-4000-220
Payload	500 Lbs (227 Kgs)	1000 Lbs (454 Kgs)	2000 Lbs (909 Kgs)	4000 Lbs (1818 Kgs)
Center of Gravity	Centered directly over the center post of the motion base			
Top Platform Dimensions	14" x 14"	48" x 39"	48" x 39"	48" x 39"
Floor Platform Dimensions	18" x 18"	60" x 36"	60" x 36"	60" x 36"
Height (std)	12"	24" or less	24"	24"
Weight (std ship weight)	15 Lbs	650 Lbs	750 Lbs	1000 Lbs

Table 3.8.1: Mechanical Specifications

	Units	710-2-500-220	710-2-1000-220	710-2-2000-220	710-2-4000-220
Max. Roll Angle	Degrees	±15˚	±20˚	±20˚	±20˚
Max. Roll Velocity	degrees/ sec	±60	±60	±60	±60
Max.Roll Acceleration	degrees/ sec^2	±300	±300	±300	±300
Max. Pitch Angle	Degrees	±15˚	±20˚	±20˚	±20˚
Max. Pitch Velocity	degrees/ sec	±60	±60	±60	±60
Max Pitch Acceleration	degrees/sec^2	±300	±300	±300	±300

Table 3.8.2: Dynamic Specifications

3.9 Development Boards

Because the control loader is such an important element to the overall system, it is crucial to take into account what are the features and components the team is looking for in the development board to be used:

- USB 2.0 support.
- Analog to digital conversion of at least 3 signals.
- Digital to analog conversion of at least 2 signals.
- Software development kit that preferably allows the microcontroller to be programed in a high level language.
- Cost-effective and with the least amount of extra features that are not needed by the project's design specifications.

3.9.1 EXP430G2 Launchpad for MSP430 16-Bit Microcontrollers

The Launch Pad Development Kit provided by Texas Instrument can help the group develop the interface between the microcontroller and the USB port and provide testing grounds for the input devices. **Figure 3.9.1** shows the development board and its main hardware components. One of the best things about this board is its price as it can be purchased for as low as $4.30 at the Texas Instrument website **[3.9.1]**.

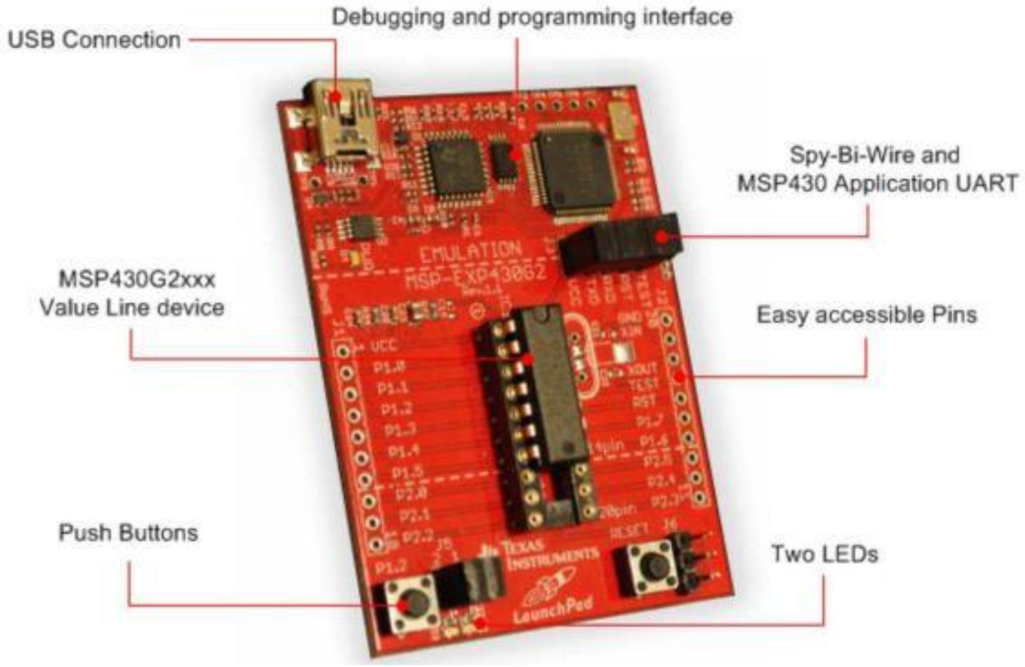


Figure 3.9.1. MSP-EXP430G2 Launchpad Overview

This development board has many features needed for this project; for instance, the Launch Pad's integrated emulator interface connects flash-based MSP430 Value Line devices to a PC for real-time and an in-system programming and debugging via USB; these two features are a fundamental part in the development of the CL since all communications will be through the USB port. In addition, it has a MSP430G2231IN14 - 2kB Flash, 128B RAM, 10 GPIO, 1x 16-bit timer, WDT, BOR, 1x USI (I2C/SPI) 8ch 10-bit ADC and a 14-/20-pin DIP (N) socket. It is also include a Built-in flash emulation for debugging and programming, which is a handful tool to trace the progress of the written software. The group has two options: using external DA and AD converters, or using a microcontroller which has these devices integrated internally.

3.9.2 MSP-EXP430F5529 Development Board

The MSP-EXP430F5529 Development board provides great features at a reasonable price, around $149.00. It is one of the latest MSP430 devices with integrated USB. The unit offers a 128KB Flash/8KB SRAM at a full speed USB 2.0. In addition, it has 16-Bits RISC architecture up to 25MHz. 16 bits are more than enough to handle the volume of information that is going to be used even in the worst case scenario, which will be the concurrent processing of multiple input signals from more than one part at a time. Moreover, the MSP430F5529 device on the development board can be debugged and powered using the integrated ezFET, or the TI Flash Emulation Tool; an example of this tool is the MSP-FET430UIF. **Figure 3.9.2** shows a visual perspective of the hardware distribution of this development board.

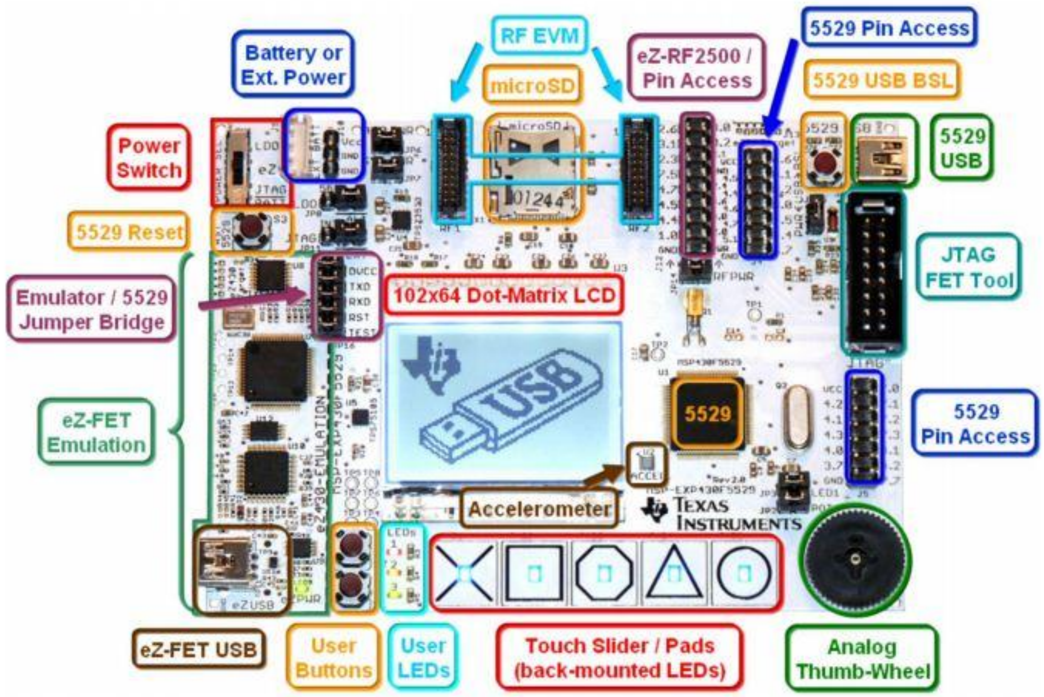


Figure 3.9.2. MSP-EXP430F5529 USB Experimenter's Board

It is very easy to set-up since it has a 102x64 Dot-Matrix LCD, which could be really helpful when reading the outcome results. Even though all the components and their functionalities provided would not be used, the group can always take them out and just use what it is needed for our project.

3.9.3 STELLARIS - LM3S3748 Development Board.

This micro controller bring to the table many interesting features like eight 10 bits ADC channels (inputs) when used as single-ended inputs, a USB flash memory port, which will be used to establish communication between the parts. Also it has four general – purpose 32-bit timers, a 64 KB single cycle SRAM and a 128 KB single cycle flash, which will be good enough to handle all the load in our project; **Figure 3.6.3** shows a more exhaustive hardware layout of this development board.

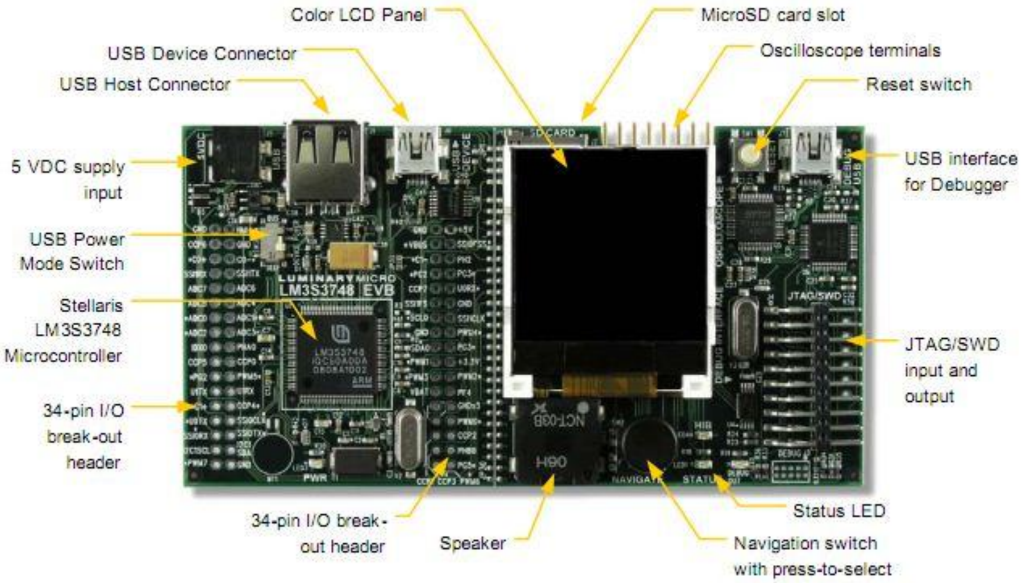

Color LCD Panel
MicroSD card slot
USB Device Connector
Oscilloscope terminals
USB Host Connector
Reset switch
5 VDC supply input
USB interface for Debugger
USB Power Mode Switch
Stellaris LM3S3748 Microcontroller
JTAG/SWD input and output
34-pin I/O break-out header
Status LED
34-pin I/O break-out header
Speaker
Navigation switch with press-to-select

Figure 3.9.3 Diagram of the LM3S3748 Development Board

As shown, the LM3S3748 has a liquid crystal graphics display with 128 x 128 pixel resolution that allows the developer to manage the board in an easier way. Another feature that will be really helpful is the oscilloscope feature, which has two differential measurement channels that provide waveform acquisition using the LM3S3748 microcontroller's Analog-to-Digital Converter.

3.9.4 MSP-TS430PM64 Development Boards

The MSP-TS430PM64 is one of the development board provided by Texas Instrument to program and configure the 64 pins microprocessor. As shown in **Figure 3.9.4,** this board does not have a USB port, so the team needs to find the way to provide an interface capable of matching the USB and the JTAG standards.

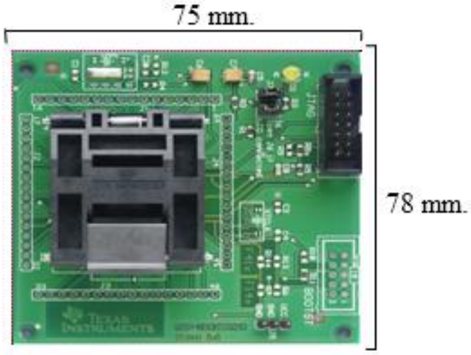
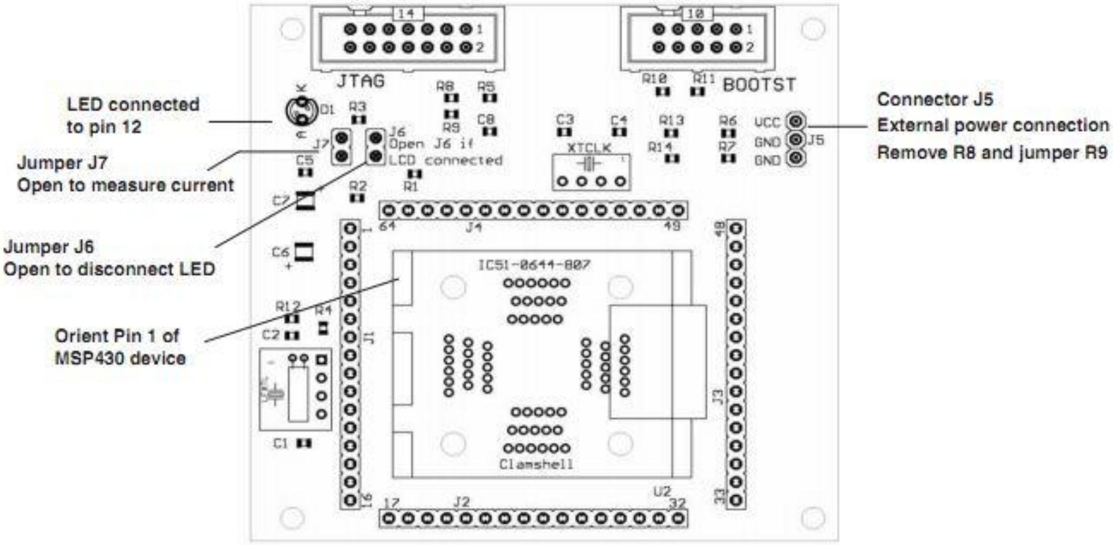


Figure 3.9.4 MSP-TS430PM64 Development Board

The board supports all the debugging interfaces through the JTAG male adapter. A more specific layout of the development board and the QFP ZIF Socket can be seen in **Figure**

3.9.5. One of the things that are highly noticeable on the board is the lack of a USB interface in it. Since all communication in our system is using the USB standard, the team needs to implement a solution for this problem. For instance, Texas Instrument offers the **MSP-FET430UIF**, which is a USB debugging interface, which will be discussed on the next pages.



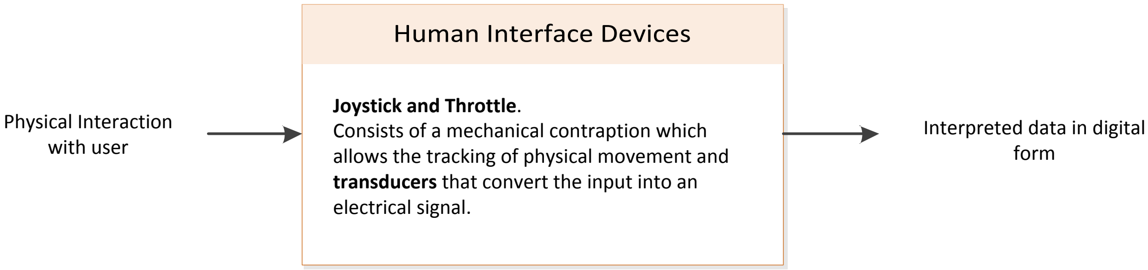
LED connected to pin 12
Jumper J7 Open to measure current
Jumper J6 Open to disconnect LED
Orient Pin 1 of MSP430 device
Connector J5 External power connection Remove R8 and jumper R9
JTAG
BOOTST
J6 Open J6 if LCD connected
XTCLK
IC51-0644-807
Clamshell
J1
J2
J3
J4
U2
VCC
GND
GND
J5

Figure 3.9.5 MSP-TS430PM64 Internal Layout

Add the development board that we used, ADuC841

3.10 Input Devices

The first question to be answered within our research for the peripherals is: how are the physical movements/signals going to be tracked? Whether the user is pulling the throttle's lever, changing the directional orientation of the joystick's arm, or pushing the pedals down; these are all physical signals which ultimately need to be translated to ones and zeros. The block diagram below shows the basic concept and serves as a guide during research and development in the area of input devices.



Human Interface Devices

Physical Interaction with user

Joystick and Throttle.
Consists of a mechanical contraption which allows the tracking of physical movement and **transducers** that convert the input into an electrical signal.

Interpreted data in digital form

Figure 3.10.1 HID Process

The joystick's mechanical composition can be yoke based, or have an arcade style design. The yoke is the common "w shaped" control for many aircraft. The Y axis is controlled by pushing away or towards the users body. In exchange, the x value is altered by rotating the handle. Arcade style joysticks consist of a cilindrical shaft which moves to the origin when released. In order to achieve this, A spring or magnet system can be implemented. The coordinate system for this structure is controlled by tilting the rod or lever left,right,forwards, or backwards.

Flight simulation turns out to be a very popular field for both home oriented hobbyists and small business hardware developers. Within these groups, there are people who have designed and built their own input devices. In the following paragraphs a few examples are presented that not only inspired the group but served as valuable reference for voltes fly.

A great design example for a flight simulation throttle is the Throttle Quadrant developed by Rob Barendregt **[3.10.1]**. His design consists of four throttles and two programmable input buttons. The throttles consist of plastic levers whose ends are connected to a big gear which is in contact with a smaller gear. This gear is then connected to rotary potentiometers (100Kohm) which allow for varying voltage levels in the circuit. The large gear allows the lever to have enough range of motion for the user; however, the rotary potentiometer has a max range of 275 degrees. In order to take as much advantage of this range as possible, Barendregt attached it to the smaller gear which has a 2.5 transmission factor. This parameter allows the potentiometer to be rotated at close to its maximum range. The circuit is then connected to a Game port connector, followed by a Game port to USB converter. Within the Host computer, the device is calibrated within the operating system and Microsoft's flight simulator. The user must specify which voltage levels correspond to the desired values in the simulator. This design is very inexpensive to build (he built it under $20), which is exactly what the group aimed for in the original Voltes Fly' component goals. What simplifies things even more is that our simulator is being built for simple planes like the Cessna so there's no need to implement more than one throttle. However, the rods used for levers appear very fragile and our final design should be more robust and be able to handle long hours of play.

Another interesting hobbyist is Mr. Igor Cesko and his interface device. This design called "IgorPlug" is an infrared computer remote control with USB support. While this design for an input device does not implement force feedback, it has a very good design and documentation provided on the website **[3.10.2]**. Potentiometers seem to be very popular in this field, used in this project as sensors for tracking movement. In order for the signals to be understood by the USB port, they had to be translated to the standard USB protocol. The AT90S23x3 microcontroller was used along with open source firmware to meet the specification. The designer notes how USB speed is really high compared to this controller, which caused some problems during the design. He points out the option of using a universal converter (serial to USB) but he wanted the challenge of implementing an USB hardware interface himself. Our project interfacing will be similar in nature: using a microcontroller and USB interface module to translate the incoming signals to the industry standard. The device driver was also written by him (which consisted of a .sys extension), using Microsoft Windows 2000 DDK environment.

Also, a testing environment was written for the controller. For our own joystick, a testing environment would seem very useful and this is a good procedure to mimic. While potentiometers seem to be the number one choice for the sensors, it is beneficial to explore the different technologies available to make an informed decision:

Induction- In order to track movement using induction, the handle is attached to a main magnet. Two sensor for each axis are placed around the origin of the shaft. As the main magnet gets closer or further away from each sensor, the magnetic flux going through them is affected. This change in magnetic flux triggers current flow and creates the desired signal, successfully translating the physical movement to an analog signal**[3.10.3]**. When the joystick is pushed to the right, the rightmost sensors will have the magnets in closer proximity. The changes in between the sensors can be used to calculate the coordinates of the joystick. For the design of a throttle, the number of sensors needed will depend on their accuracy and sensibility as well as the translational constraints of the lever.

Optical- An optical joystick has the same overall structure with only a change in the type of sensors used for each axis. As seen in U.S Patent 4,731,530**[3.10.4];** an optical joystick patented in 1988, this technology has the advantage having frictionless transducers**.** In order to track movement, a light source is used along with 2 phototransistors. The transistors react to the different levels of opacity to accurately measure the position of the stick. Since the signals generated by these transistors are so small in magnitude, the circuit requires the addition of operational amplifiers to get the proper output. The fact that there is less friction involved draws attention to this technology; however, optical potentiometers are not easily found for purchasing.

Resistive- As seen in the example devices shown, potentiometers are the most common option. The main idea of resistive joysticks is to use the varying resistance to track the movement. For example, to track the x and y position in the Yoke or directional joystick: 2 potentiometers would be needed to create an electrical signal for each directional axis. With regards to the potentiometers used, they usually have 3 terminals: one for voltage source, another terminal to ground, and the last one is the one is connected to a mechanical device that allows the person to adjust the resistance (knobs, levers, and handles are just some of the examples) **[3.10.5]**.

A wide range of potentiometers are available for different applications. These come in a variety of resistance ranges, and interfaces for adjusting the values. For example: sliding pots require the user or system to push a handle back and forth while rotary pots have a spinning shaft. For the purpose of our devices, rotary potentiometers seem to be the best choice. It can easily be attached to a gear or to the axis of a moving joystick. These components are also classified by their response to the input. There are some that respond in linear time, and others which react logarithmically. Log based potentiometers are mostly used in the audio industry because humans perceive sound in this manner therefore achieving a more realistic fading effect. However, linear function potentiometers will be accurate enough to do the trick for this project. Computer Joysticks often use 100kohm pots, or 470k (restricting its rotation to about ¼ thus achieving 100kOhm) **[3.10.5][3.10.6].**

3.10.1 Design Options

With regards to the game port connection, this method is very simple to implement. Also known as a DA-15 connection **(see Table 3.10.1)**; the interface was very popular in this type of design but are now obsolete with modern computer systems. However, a game port to USB convertor would easily solve this issue. The 15 pins support up to 2 joysticks and four buttons. Pins 3 and 6 would correspond to the x and y axis, the rest involve pins for 5v, ground, and additional inputs **[3.10.7]**.

pin	
1	5V
2	B1
3	X axis
4	GND
5	GND
6	Y axis
7	B2
8	5V
9	5V
10	B4
11	X axis 2
12	GND
13	Y axis 2
14	B3
15	5V

Table 3.10.1 DA-15 Gameport connector

The throttle signal could be considered as a 3rd axis and integrated into the same system. The three analog signals would go through the 15-pin connection and the hardware within the converter would take care of the analog to digital conversion. The software driver provided by this converter would take care of the interface with the operating system as well. This overall design option is shown below in **Figure 3.10.2**.

While this option would work, it does not have sufficient design for the groups liking. This block diagram does not include the electronics required for the force feedback system to be implemented (see later sections). The Voltes fly group wants to achieve analog to digital conversion and hardware/software interface through their own design. The final interface design to be used for both the throttle and joystick will be explained with greater detail in the design specification **section 4**.

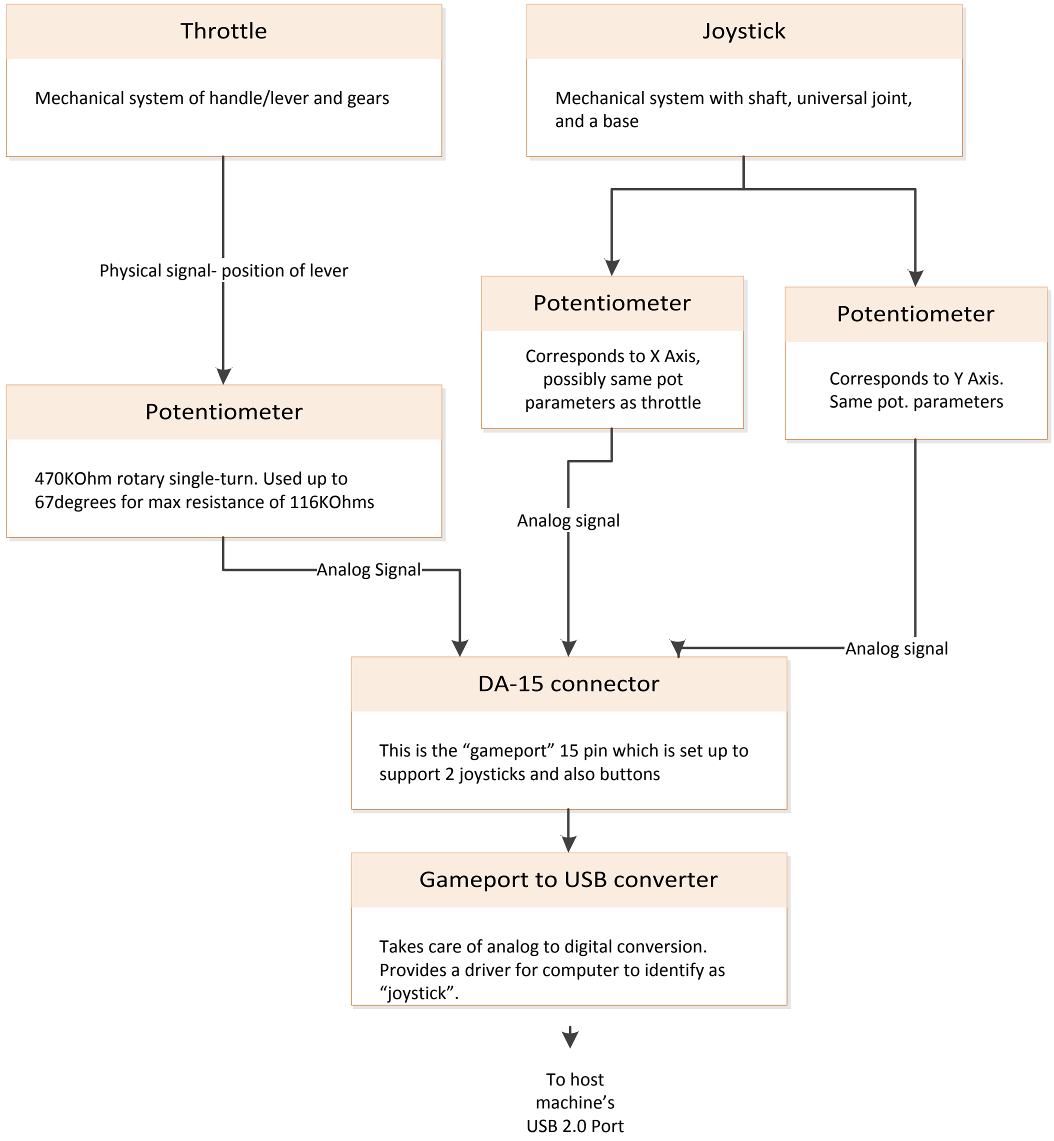

Throttle
Mechanical system of handle/lever and gears
Joystick
Mechanical system with shaft, universal joint, and a base
Physical signal- position of lever
Potentiometer
Corresponds to X Axis, possibly same pot parameters as throttle
Potentiometer
Corresponds to Y Axis. Same pot. parameters
Potentiometer
470KOhm rotary single-turn. Used up to 67degrees for max resistance of 116KOhms
Analog signal
Analog Signal
Analog signal
DA-15 connector
This is the "gameport" 15 pin which is set up to support 2 joysticks and also buttons
Gameport to USB converter
Takes care of analog to digital conversion. Provides a driver for computer to identify as "joystick".
To host machine's USB 2.0 Port

Figure 3.10.2 Input Devices design option

3.10.2 Force Feedback

Tactile feedback is used in wide range of devices: from medical applications, mobile phones, to game controllers. It provides a way of communication with the user and creates an overall enhanced experience. "Rumble" effects on game controllers have become somewhat of a standard in the industry, let's explore some designs and methodologies for achieving this effect. A great example is U.S Patent number 5,742,278**[3.10.7]**. It consists of a PC joystick with force feedback that can implement different intensities, waveforms, and patterns. The system is structured as follows: a link is made with the host computer through a serial port link; this is then connected to a digital signal processor. This processor has a memory device connected to it which has different force definitions stored for the processor to read. The processor is then connected to 2 DC drive motors, and an actuator which applies the effect on to the handle of the joystick. The drive motors are 24V DC carbon brush motors and can achieve a maximum of 10,000 rpm. Each motor is attached to each axis so that the rotational movement causes the haptic effect. Another related U.S Patent by Immersion Human Interface Corporation **[3.10.8]** demonstrates the use of pre-calculated force effects on a ROM. This is shown to save time because the peripheral's microcontroller does not have to perform as much mathematical computations. Circuit details are provided, which show the necessary connections between the microprocessor and the actuator which creates the feedback. Digital to analog converters are wired to an amplifier circuit. The amplifiers and resistor values can vary depending on how much voltage is needed for the type of actuator used.

The second option for a tactile feedback system is a motor for each axis that is triggered independently of the events in the flight simulator. Instead, they would be triggered in response to the user moving the joystick. This system needs a transducer that can detect whenever a force is applied by the user, and have the motors react accordingly. This said device can be a load cell. This device can measure force and outputs an electrical signal in the magnitude of millivolts. Therefore, circuitry for amplification must be implemented in order for the signal to be strong enough for the motors to react. Because the system is basically a linear controls problem, circuitry would be needed to maintain overall stability and accurate response to input signals. **Figure 3.10.4** gives a visual representation of the static feedback system using a load cell.

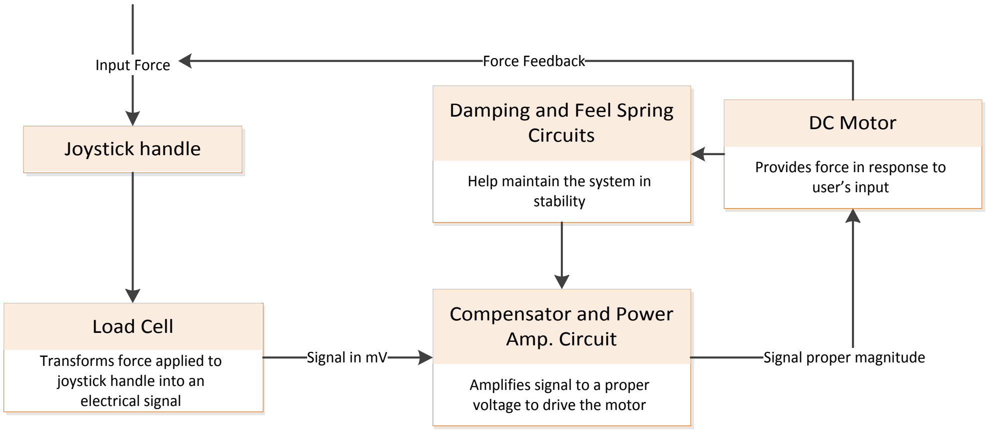

Input Force
Force Feedback
Damping and Feel Spring Circuits
Help maintain the system in stability
DC Motor
Provides force in response to user's input
Joystick handle
Load Cell
Transforms force applied to joystick handle into an electrical signal
Signal in mV
Compensator and Power Amp. Circuit
Amplifies signal to a proper voltage to drive the motor
Signal proper magnitude

Figure 3.10.3 Analog Force Feedback Model

3.11 Communication with Flight Computer

There are several ways in which the control loader, throttle, and joystick can communicate with the host computer. Let's look at the different ports and communication technologies available to accomplish this task.

USB 2.0

The universal serial bus is an industry standard, with 4 signal lines. One line for power, another one for ground, and 2 data signals. It has 3 modes of use which bit transfer rates of 480Mb/s, 12Mb/s, and 1.5Mb/s **[3.11.1]**. The joystick and throttle together would use 3 to 4 signals of maybe a byte long each. That is a total of at most 8 x 4 = 32 bits/second.

This means the system would only be using 32/ (10^6*1.5) = 2.133*10^-5 % of the total available bandwidth, running at the lowest transfer rate. Taking into account the output signals from the flight computer to the motion base would only increase the data packet by about a half word (16 bits) which would not make a big difference in the relative bandwidth used. The advantages of communicating to the host through USB are the following:

- Industry Standard with plug-and-play capabilities
- Having all our components communicate to the host through one high speed port
- Bidirectional support to effectively send values to motion base

Microsoft offers a windows driver development kit for different versions of their operating system. It provides sample drivers that can be used to send and receive raw data bytes between the host and its peripherals **[3.11.2]**. These features make it a good candidate for our component communication.

USB Module

These microcontrollers have many of the desired parts for our project, but unfortunately they do not have a USB module, the development boards for this family of microcontrollers does not have a USB port on them. In order to be loyal to our initial specification in terms of communication using USB, the gropu will create an independent USB interface and for debugging purposes the will use a JTAG adapter. This device makes compatible a USB signal with a JTAG interface allowing a proper debugging process. In order to build a USB interface, a USB development board would be used. A really interesting option is the Teensy Board, which is being used in several projects. **Figure 3.11** two possible form-factors for this board.

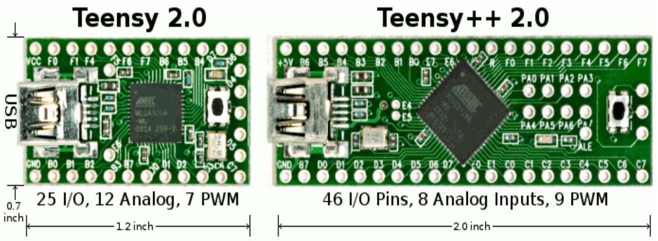


Figure 3.11 Teensy Development Board(waiting for Copyright Permissions)

RS-232

The standard protocol is intended for data transfer rates of 256kbs which is still enough for the amount of data being used by the peripheral components. Data is transmitted in pin 2 and received in pin 3 and the rest are used for acknowledging, handshaking, and other communication protocols **[3.11.3]**. A possible option is to use the USB port and have it behave as a serial port.

MSP-FET430UIF

The MSP-FET430UIF is a powerful flash emulation tool that includes a USB debugging interface, which result very handful when program and debug through the JTAG interface any microcontroller belonging to the MSP430 family. Some of its features are very attractive to our project like:

- Software configurable supply voltage between 1.8 and 3.6 volts at 100mA.
- Support JTAG Security Fuse blow to protect code.
- Support all MSP430 boards with JTAG header.
- No external Power Supply is required.
- The flash memory can be erased and programmed in seconds.

Add the Serial communication option

3.12 Hardware Interfacing

The CL is composed of several elements that perform different actions and each of them must be analyzed in detail. The CL is responsible for all communications between the platform, the IOS, the throttle and the joystick so it must be designed carefully because if an error is made on this system, it will create a chain reaction causing errors in the subsystems that depend on him. In order to handle all the inputs needed the group will have to use a microcontroller with at least three 12-bit Analog-to-Digital Converter (ADC12) channels and either a 12-bit Digital-to-Analog Converter (DAC12) with two outputs or two DAC12. Let's analyze some of the devices that caught our attention during the research process.

3.12.1 MSP430FG437

The MSP430FG437 is a really versatile device that has an ultra-low- power consumption and a wide range of peripherals. This microcontroller is optimized to achieve an extended battery life, so it can use in various portable applications. Its architecture has a powerful 16-bit RISC CPU, 16 bits registers, a digitally controlled oscillator (DCO), a 16-bit timer, and the capability to communicate using synchronous protocols like SPI or I2C, among others. The most important in this device, from our point of view, is that it has 12 channels ADC12 and two DAC12 built-in, which is much more than it is needed to achieve our technical specifications. In **Figure 3.12.1** it can be observe the internal layout of the MSP430FG437.

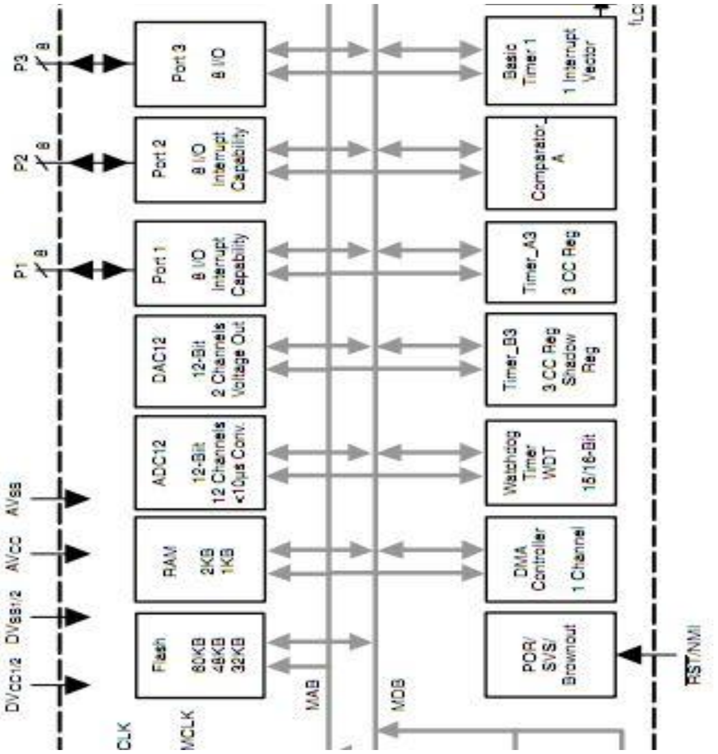


Figure 3.12.1 Functional Block Diagram for the MSP430FG437

In addition to the features exposed above, the MSP430FG437's individual I/O bits are programmable independently, which is very handful when you address independent inputs coming from different parts. For instance, the input from the throttle and the joystick are not functional dependent, so they can be handled independently by the processor. The Read/Write access to port-control registers is supported by all instructions and any combination of output, input and interruption is possible on this device giving you the ability and flexibility to control the values in any register or process execution at any time. **Figure 3.12.2** shows the external pin configuration of the MSP430FG437 microprocessor.

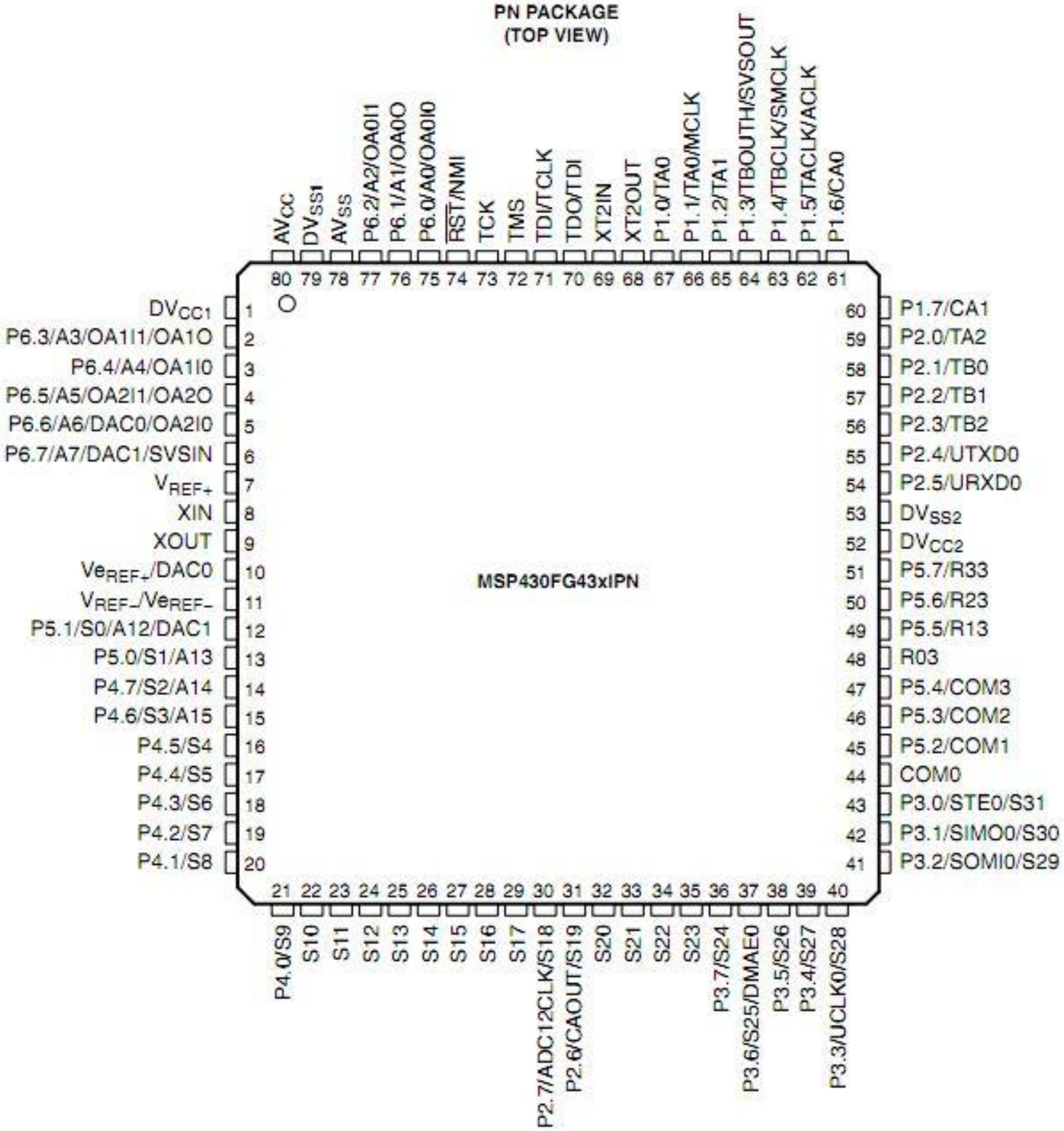
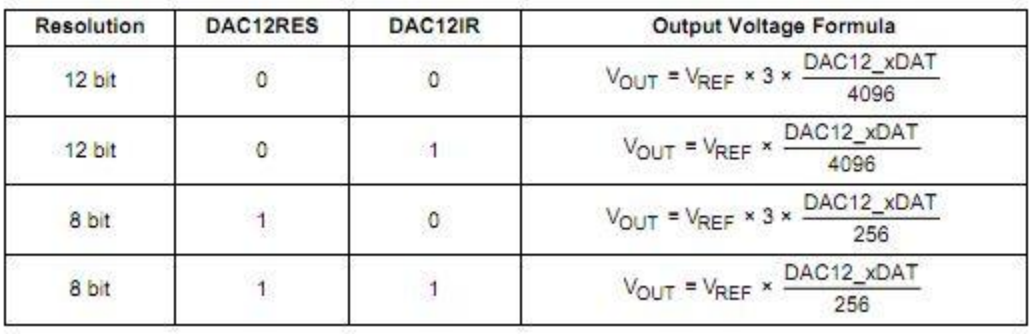


Figure 3.12.2 MSP430FG437 pins layout

The team arrived to the conclusion that this device is an excellent candidate for the final design based on the features and benefits that offers at an excellent price, around $6.50, which make it highly affordable.

3.12.2 MSP430FG437 internal DAC12s

As previously stated, the MSP430FG437 microcontroller has two 12 bits DAC built-in, which can be configured to operate in 8 or 12 bits mode using the DAC12RES bit. The greater the number of bits, better resolution is achieved, so 12 bits mode of operation seem fair enough. This device can be also configured to allow inputs selections between straight binary and 2s-compliment data. The team will use the first alternative, so the **Table 3.12.1** shows the outputs voltage for different resolutions. [reference number]



Resolution	DAC12RES	DAC12IR	Output Voltage Formula
12 bit	0	0	$V_{OUT} = V_{REF} \times 3 \times \frac{DAC12\_xDAT}{4096}$
12 bit	0	1	$V_{OUT} = V_{REF} \times \frac{DAC12\_xDAT}{4096}$
8 bit	1	0	$V_{OUT} = V_{REF} \times 3 \times \frac{DAC12\_xDAT}{256}$
8 bit	1	1	$V_{OUT} = V_{REF} \times \frac{DAC12\_xDAT}{256}$

Table 3.12.1 MSP430FG437's DAC12 Full Scale Range.

Some of the most remarkable features of this device are noted in **Table 3.12.2**.

Feature	Benefits
Low Power consumption (250mW max) and Small Size	Make the device ideal for automatic test equipment, data acquisition systems, DAC-per-pin programmers, and closed-loop servo-control.
12-bit parallel input data capabilities and double buffered DAC input logic.	These features allow simultaneous update of all DACs
Data Read back	Give the ability to load its registers at any time.

Table 3.12.2 MSP430FG437's DAC12 features.

3.12.3 MSP430F2618

The MSP430F2618 has some similarities with the MSP430FG437 like its low-power consumption and its internal layout, but it has differences that make this microcontroller one of the favorite's candidates among the teammates. This device has only 8 channels for its ADC12 which met our requirements since only three are needed, but it has two DAC12 and 8KB of RAM that supersede the 1KB of RAM from the MSP430FG437.

On top of that, the MSP430F2618 cost less and offers better performance since not only have more RAM but also more Flash Memory, which is 116 KB+256 B and runs at 16MHz instead of the 8MHz from the previous microcontroller. **Figure 3.12.3** shows the internal layout of this wonderful device.

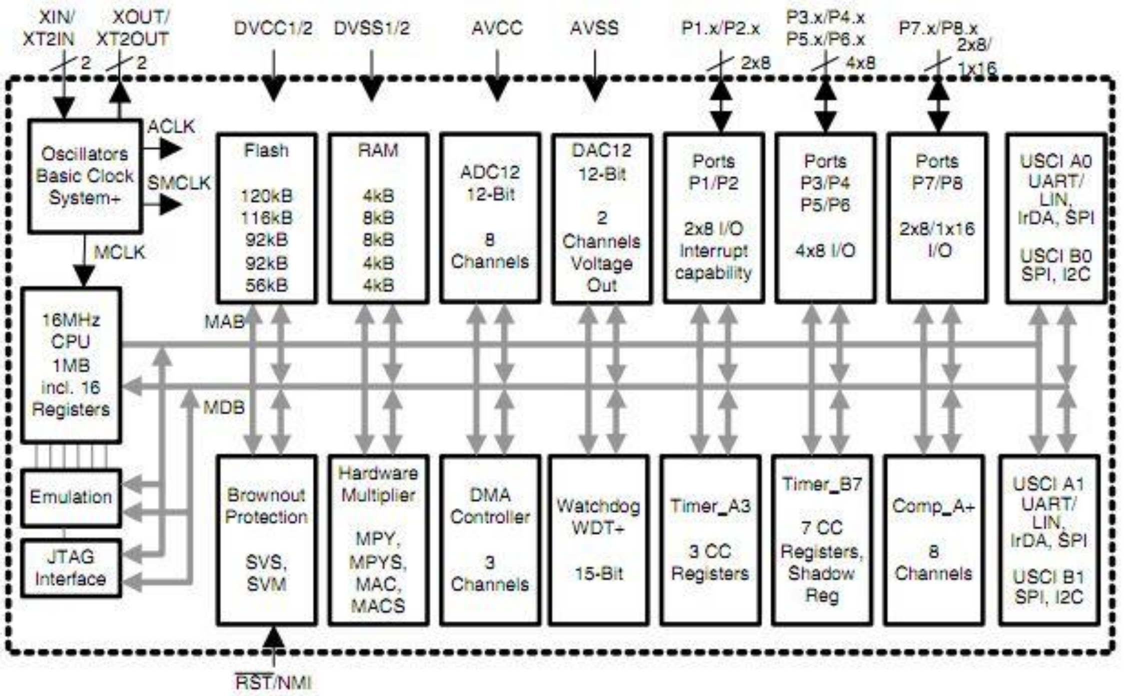


Figure 3.12.3 Functional Block Diagram for the MSP430F2618

On the functional block diagram it can be seen the block distribution within the MSP430F2618 and how they communicate with each other. It has plenty of I/O digital pins, for instance, pins from P7.0 to P8.5 are for general-purpose so they will be used to receive or send data as needed, pins from P6.0 to P6.4 are intended to be used by the ADC to process the incoming analog signals from the joystick and throttle. In addition, the DAC12 will use pins P6.5, P6.6 and P6.7 to communicate voltage values to the external circuitry. The assigned developer will use the DAC12s in combination with the DMA module, which increase the throughput between peripheral modules and reduces system power consumption since the CPU remains in sleep mode without having to awake to move data from one peripheral to other.

Just as the previous microcontroller, on the MSP430F2618 each pin can be programed independently and port from P7 to P8 can be accessed word-wise. Moreover, any combination of inputs, output and interruptions are possible and all instructions support Read/Write access to registers, which gives the developer the freedom to use every instruction available in the MSP430F2618 microprocessor. No wonder this processor is used by aerospace, defense, and medical companies in their high end applications.

In addition, the **comparator_A+** module is going to be used in monitoring external analog signals, battery voltage supervision and to support precision slope analog-to-digital conversions. By comparing both microcontrollers, based on the cost and benefits that this microprocessor offers, there is a consensus of the entire team that this is going to be the part to use in our first prototype.

Figure 3.12.4 shows the pins configuration for the MSP430F2618 microcontroller. As it is shown, the device has 64 pins and they have to be soldered in a 113-Pin Ball Grid

Array (BGA) [reference], which will make our prototype building process a little more complex than expected and for this reason, the team will probably use outsourcing services to build the printed circuit board. Despite this small inconvenience, the device looks like the perfect one to do the job.

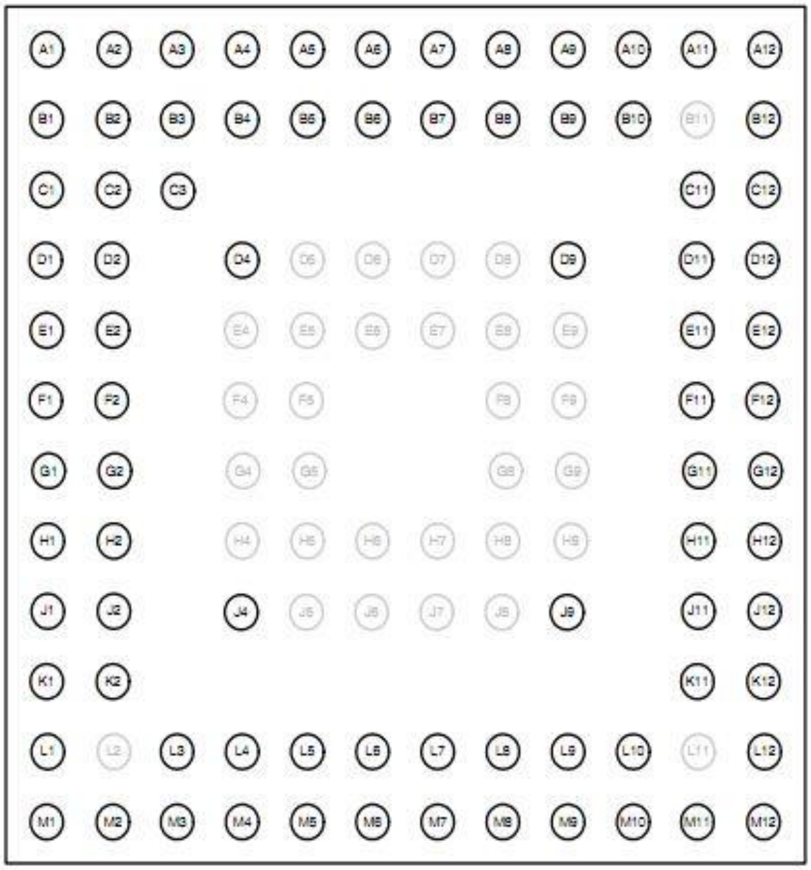


Figure 3.12.4 pins configuration for the MSP430F2618

3.12.4 MSP430F2618 internal DAC12s

The ADC12 can implement a 16 word conversion and control buffer and this buffer allows up to 16 independent ADC samples; this samples can be stored and converted without intervention from the CPU, which relieves the processor load in a large percentage. On top of that, the DAC12 can be used in conjunction with the DMA allowing the processor a faster access to any memory location. If needed, both DAC12 can be used together and the MSP430F2618 allow us to use them synchronously. Since this microcontroller functions at a small voltage range, between 1.8V – 3.6V and the DAC12 has a single polarity (positive), the design is going to need an operational amplifier circuit after the DAC12 in order to achieve not only a higher voltage gain but also both polarities. The polarities are needed to indicate the direction of movement to the servo-motors' arms and the voltage value will dictate the amount of angular momentum to the motion base. The **Table 3.12.3** indicates the relationship between voltage values and range of motion in our system.

Voltage Value	Range of Motion
0 V	0 degrees
+7V	90 degrees
+10V	120 degrees
+15V	180 degrees
+30V	360 degrees

Table 3.12.3 Relation between Voltage and Range of Motion

Add the 8052 microcontroller and its features

3.12.5 Digital to analog conversion

The team is planning to use two Digital- Analog Converters to convert, as the name states, the digital signals into analog ones, so they can be understood by analog devices like the servo-motors. Even though the MSP430 microcontroller has one 12 bits DAC built-in, two of them will be needed to use them in combination with development boards like the MSP-EXP430G2 Launchpad and the MSP-EXP430F5529. **Figure 3.12.5** shows the DAC7724N, which is a 12 bits quad voltage output digital-analog-converter with 12-bits monotonic performance over the specified temperature guaranteed. Some of the most remarkable features of this device are:

Feature	Benefits
Low Power consumption (250mW max) and Small Size	Make the device ideal for automatic test equipment, data acquisition systems, DAC-per-pin programmers, and closed-loop servo-control.
12-bit parallel input data capabilities and double buffered DAC input logic.	These features allow simultaneous update of all DACs
Data Readback	Give the ability to load its registers at any time.

Table 3.10.1 DAC7724N features.

Figure 3.12.5 DAC7724N Dimensions

3.13 Programming Languages

Not all parts of the project are built on the same platforms, so the team needs to use different programming languages for each part. For instance, the IOS should be developed in C# while the interface between the USB and the microcontroller could be developed in either C or C++.

C was developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories for use with the UNIX operating system. It is a general-purpose computer programming language that was designed for implementing system software, but it is also widely used for developing portable application software. It is widely used on many different software platforms, and there are few computer architectures for which a C compiler does not exist which makes it very useful in a product like ours.

C++ was developed at Bell Labs as an enhancement to the C language. It is a general purpose programming language with a bias towards systems programming that support data abstraction, support object-oriented programming and generic programming. C it's regarded as a middle level language since it's compose by both high level and low level language features. C++ expands the functionality of the C language to include classes, data encapsulation, stronger type checking, inheritance, exception handling, namespaces, and many other features.

C# is a simple, modern, object-oriented, and type-safe programming language. C# combines the high productivity of Rapid Application Development (RAD) languages and the power of C++. Visual C# .NET is Microsoft's C# development tool. It includes an interactive development environment, visual designers for building Windows and Web applications, a compiler, and a debugger. Visual C# .NET is part of a suite of products, called Visual Studio .NET, that also includes Visual Basic .NET, Visual C++ .NET, and the JScript scripting language. All of these languages provide access to the Microsoft .NET Framework, which includes a common execution engine and a rich class library. Even though C# is a new language, it has complete access to the same rich class libraries that are used by experienced tools such as Visual Basic .NET and Visual C++ .NET. **[3.13.1]**

3.14 Existing Similar Projects

3.14.1 Thano's Flight Simulator

Similar projects exist at all levels, from large companies to enthusiasts. For example, one of the most original projects developed by Thanos, a Greek guy and can be viewed on YouTube at this address **[3.14.1]**. Thano's project uses a pulley system in order to move the motion platform he used two wiper motors. The motion platform was built with PVC pipes and hardware easy to find in any hardware store at a really low price, so the project was not only relatively easy to build but also accessible to those with low budgets. On the **Figure 3.14.1** it can be seen an overall schematic of the electronic system used in this project. It is important to notice that Thano's website is no longer in service but he explicitly gave permission to everybody to download the schematics and specifications. Even though a different approach will be used, our project shares some similarities with this one in terms of simplicity and affordability.

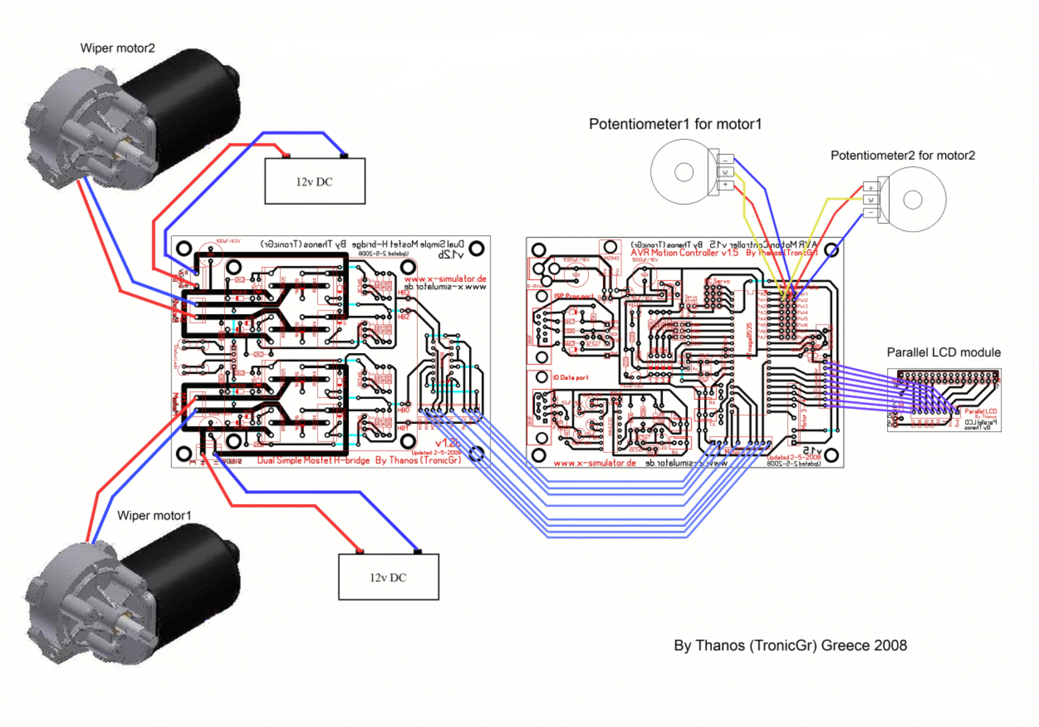

Wiper motor2
12v DC
Potentiometer1 for motor1
Potentiometer2 for motor2
Dual Simple Mosfet H-bridge By Thanos (TronicGr)
AVR Motion Controller v1.5 By Thanos (TronicGr)
www.x-simulator.de
Parallel LCD module
Wiper motor1
12v DC
By Thanos (TronicGr) Greece 2008

Figure 3.14.1 Connection Diagram of the AMC 1.5 and DSMhb

3.14.2 Gobosh G700s Flight Simulator

This project was developed by Christopher Dlugolinksi, Robert Gysi, Joseph Munera, and Lewis Vail during the Fall of 2009 and the Spring of 2010**[3.14.2]**. Even though our project shares a lot of technical functionalities with Gobosh G700s, our approach is significantly different. Since their project was intended to be used for educational purposes, they were more focused on the gauges' functionalities, the accuracy of their readings and on recreating the cockpit as close to reality as possible while our project put more emphasis not only on the recreation of the cockpit's details but also on the sensations that the pilot will experiment in a real experience flying an aircraft. **Figure 3.14.2** shows a screenshot of this project.

Figure 3.14.2. Gobosh G700s Flight Simulator (waiting copyright permission)

3.15 Washout Algorithm

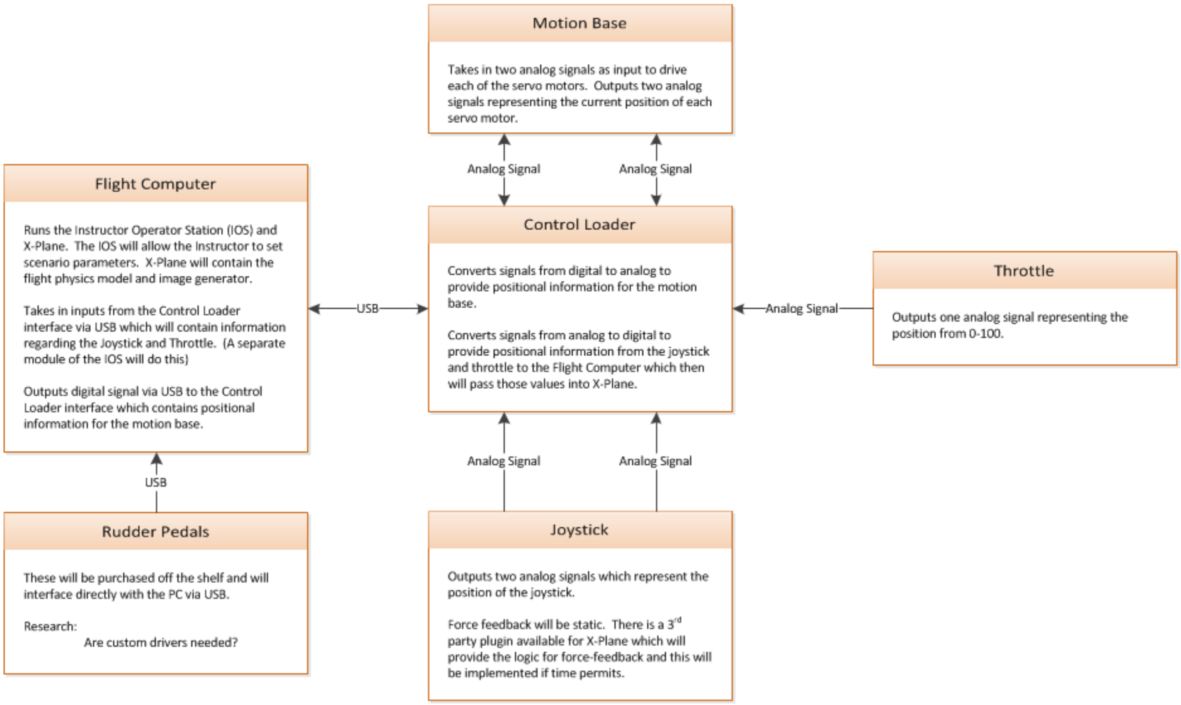
A primary component of modern simulators is the implementation of a washout algorithm. This algorithm is responsible for providing the realistic feel for the simulator. The need for it is due to the fact that most motion bases are unable to achieve the same flight angles as the actual aircrafts. For example, an aircraft can have a roll value of 22 degress while the motion base only has a roll range up to 15 degrees. The washout algorithm handles this by "faking" the pilot into thinking the bank angle is greater than it actually is. This algorithm is extremely complex and due to that it will not be implemented in the Voltes Fly due to timing constrants.

4 Design

4.1 Initial Design Architectures and Related Diagrams

Initially the group designed the system so that the different parts like the joystick, the throttle and rudder pedals establishes communication directly to the computer running the simulator, but anticipating synchronization issues it was decided that everything was handled by the Control Loader except for the rudder pedals, which are the only parts communicating directly to the Flight Computer. Every communication has to go through the CL, and all the synchronization will take place there. **Figure 4.1** shows the high level structure of the entire system and what kinds of signals, analog or digital are used between independent modules. The ultimate goal of the system is to gather a joystick position and translate that into movement on the motion base.

Update this diagram



Motion Base

Takes in two analog signals as input to drive each of the servo motors. Outputs two analog signals representing the current position of each servo motor.

Flight Computer

Runs the Instructor Operator Station (IOS) and X-Plane. The IOS will allow the Instructor to set scenario parameters. X-Plane will contain the flight physics model and image generator.

Takes in inputs from the Control Loader interface via USB which will contain information regarding the Joystick and Throttle. (A separate module of the IOS will do this)

Outputs digital signal via USB to the Control Loader interface which contains positional information for the motion base.

Control Loader

Converts signals from digital to analog to provide positional information for the motion base.

Converts signals from analog to digital to provide positional information from the joystick and throttle to the Flight Computer which then will pass those values into X-Plane.

Throttle

Outputs one analog signal representing the position from 0-100.

Rudder Pedals

These will be purchased off the shelf and will interface directly with the PC via USB.

Research:
Are custom drivers needed?

Joystick

Outputs two analog signals which represent the position of the joystick.

Force feedback will be static. There is a 3rd party plugin available for X-Plane which will provide the logic for force-feedback and this will be implemented if time permits.

Figure 4.1.1 Overall Design

The Control Loader sends the positional data for the Motion Base, Throttle, and Joystick to the Flight Computer. The Flight Computer then will receive the positional values from the Rudder Pedals. After the Flight Computer has received the positional data from all the systems, it will then pass those values into X-Plane. X-Plane will then take those values and compute the roll and pitch of the aircraft. Finally then, the Flight Computer will send the roll and pitch values back to the Control Loader, which in turn will pass the roll and pitch values to the Motion Base. Although each system runs asynchronously from each other, the Flight Computer ensures that the roll and pitch values from X-Plane match accordingly.

4.2 Signal Processing Unit (SPU) Update this

The Control Loader receives and responds to any incoming signal coming from the subparts. Just for simplification purposes, **Figure 4.2** was created; showing only the event when the flight computer transmits information to the CL.

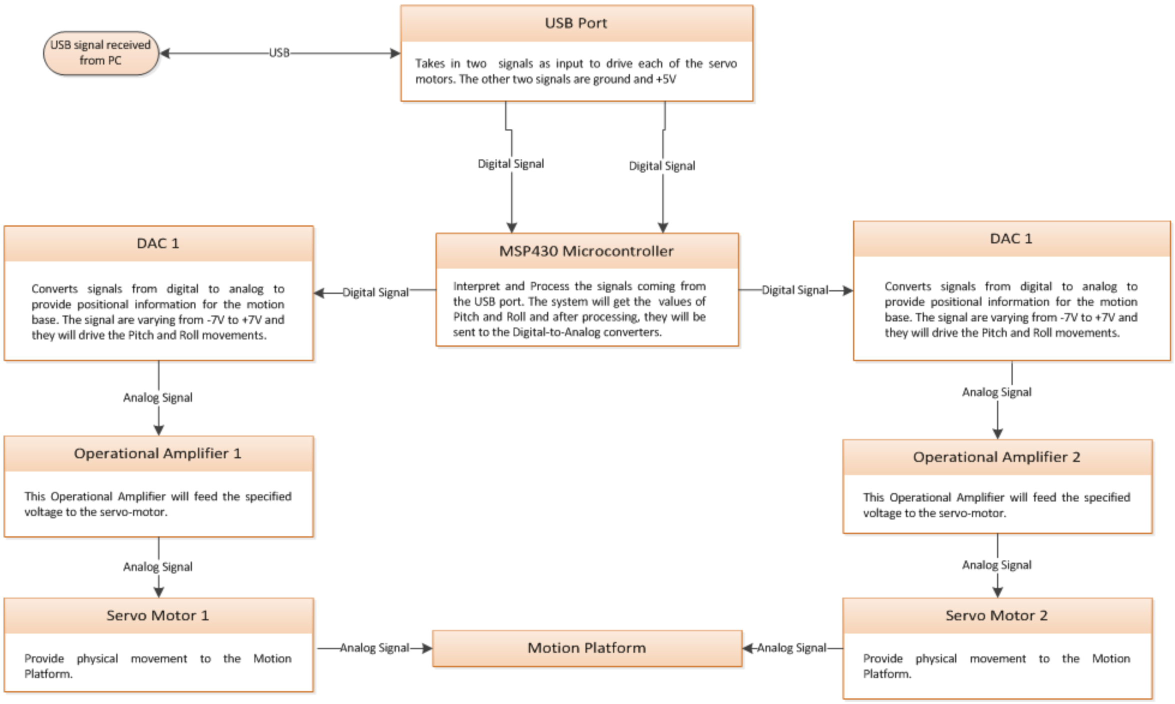

USB signal received from PC
USB
USB Port
Takes in two signals as input to drive each of the servo motors. The other two signals are ground and +5V
Digital Signal
Digital Signal
DAC 1
Converts signals from digital to analog to provide positional information for the motion base. The signal are varying from -7V to +7V and they will drive the Pitch and Roll movements.
Digital Signal
MSP430 Microcontroller
Interpret and Process the signals coming from the USB port. The system will get the values of Pitch and Roll and after processing, they will be sent to the Digital-to-Analog converters.
Digital Signal
DAC 1
Converts signals from digital to analog to provide positional information for the motion base. The signal are varying from -7V to +7V and they will drive the Pitch and Roll movements.
Analog Signal
Analog Signal
Operational Amplifier 1
This Operational Amplifier will feed the specified voltage to the servo-motor.
Operational Amplifier 2
This Operational Amplifier will feed the specified voltage to the servo-motor.
Analog Signal
Analog Signal
Servo Motor 1
Provide physical movement to the Motion Platform.
Analog Signal
Motion Platform
Analog Signal
Servo Motor 2
Provide physical movement to the Motion Platform.

Figure 4.2 CL and motion platform interaction

To be changed

The team decided to use the **MSP430F2618** microcontroller with the **MSP-TS430PM64** development board to build the control loader unit. Both the board and the microcontroller not only have all the parts needed for the project but also no external interface is needed between the essential parts like the ADC12, DAC12 and the processor. If the boards considered initially had been used, the development of the interface between the microcontroller and its peripherals would have been very difficult to develop. So, by having everything built-in on the same chip the complexity of the interface is reduced and the design is more compact and robust.

The voltage out of the DAC12 is around the 2.2V – 2.5V range, so an external circuit with two op-amps was designed in cascade in order to achieve not only the desired voltage; which is 10 V, but also to get both polarities: positive and negative. These polarities are needed to indicate the arm's directions; if a negative value is received, the arm will move down and if a positive value is received, the arm will move up. In the event that a Roll action occur to the right, the left arm will receive a positive voltage while the right arm will receive the same voltage but negative. On the other hand, if a pitch action occurs, both arms will receive the same voltage. Two LM348D op-amps were used in the circuit; this specific op-amp is very reliable, and it has very low power consumption. **Figure 4.1.1** shows the circuitry schematic of the external interface.

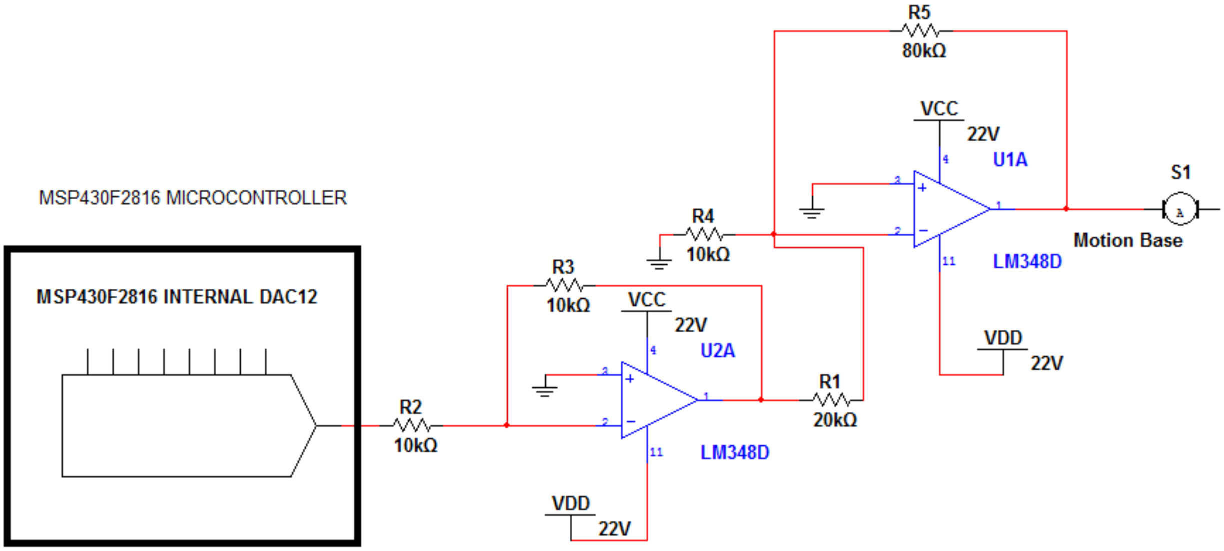


Figure 4.1.1 External Op. Amp Circuit

It was also determined that the full range of angles was not needed in the design. Instead, the design just needs to use from 90 degrees to minus 90 degrees (or 270 degrees. Therefore, the voltages range used will be varying from -7V to 7V.

Figure 4.1.2 shows the range of movement of the motion base's arm, which is represented by the shadow area on the figure. The team will use the JTAG adapter or the MSP-FET430UIF to debug the application code and for testing analysis. The wire used from the operational amplifiers to the motion base is a 12 gauge wire.

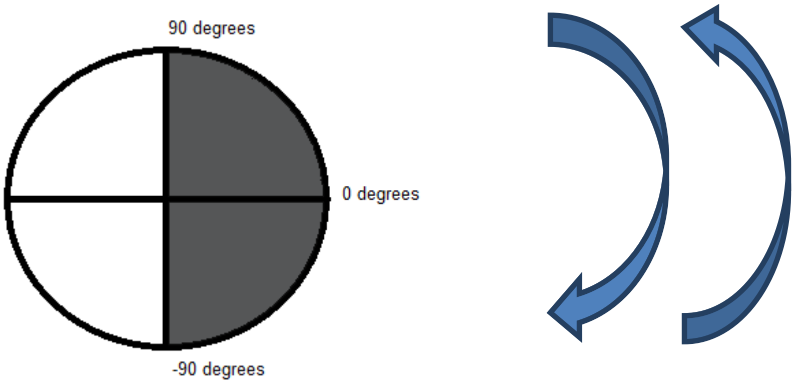


Figure 4.1.2 Range of Movement.

Several functions were needed to implement the Control Loader functionality. All functions were programmed on the **MSP430F2618** using Code Composer Studio V4 Core Edition with a plugging called Grace that made our developing process easier and more user friendly. The code inside the microcontroller has an infinite "while loop" and within all the functions resides; some of them are the following:

ReadADCChOne(): This function will constantly be checking the channel one of the ADC12, which belong to the throttle, for new analog voltage values at pin P6.5. If new information is detected those values are send to the DAC12 using its pin 12.1. The function needs to perform some validations on the input voltages to guarantee that they are in range, if not in range, the value should not pass since it could harm the system. This value will be sent directly to the Flight Computer.

ReadADCChTwo(): This function will constantly be checking the channel two of the ADC12, which belong to X value from the joystick, for new analog voltage values at pin P6.6. If new information is detected those values are send to the DAC12 using its pin 12.0. The function needs to perform some validations on the input voltages to guarantee that they are in range, if not in range, the value should not pass since it could harm the system. This value will be sent directly to the Flight Computer.

ReadADCChThree(): This function will constantly be checking the channel three of the ADC12, which belong to Y value from the joystick, for new analog voltage values at pin P6.7. If new information is detected those values are send to the DAC12 using its pin 12.1. The function needs to perform some validations on the input voltages to guarantee that they are in range, if not in range, the value should not pass since it could harm the system. This value will be sent directly to the Flight Computer.

ReadUSB(): This function will constantly be checking the info coming from the USB interface, carrying the Pitch and Roll values. If new information is detected those values are send to the DAC12. These values were already processed by a washout algorithm on the Flight Computer's IOS, so they are ready to be redirected to the next stage, which is the DAC12. The function needs to perform some validations on the X and Y values to guarantee that they are in range, if not in range, the value should not pass since it could harm the system.

SendUSB(): This function will constantly be sending information to the DACs the Pitch and Roll values gathered from the USB interface.

BuildPackets(): This function will be responsible of building the information packets needed by the output devices, so this function will be constantly invoked by the SendUSB() function.

CleanRegisters(): This function is at charge of cleaning the registers either when the application end or when it crash.

Since the designed board does not have a USB port in it, the group is going to use a USB development board to connect the microcontroller MSP430F2816. It will act as a man-in-the-middle board, which its only function is to transfer the USB signal from the Flight Computer to the MSP-TS430PM64 board. The board will be the Teensy 2.0 and its specifications can be seen in **Table 4.2**.

Features	Specification
Processor	ATMEGA32U4
Flash Memory	32256
RAM Memory	2560
EEPROM	1024
I/O	25
UART,I2C,SPI	1,1,1

Table 4.2.1 Teensy 2.0

4.3 Software and Hardware Interfacing

4.3.1 Flight Computer to Control Loader

The Flight Computer communicates to the Control Loader through the use of USB Data packets. The payload in that packet will contain the roll and pitch float values sent from the Flight Computer to the Control Loader. Figure 4.3.1.1 illustrates the packet structure that will be sent from the Flight Computer to the Control Loader. These values represent the roll and pitch of the platform, not of the simulation. Due to the limitations of the motion base, in order to achieve the full range, a washout algorithm would need to be implemented on the Flight Computer to modify the roll and pitch before sending them to the Control Loader.

Roll (float value)	Pitch (float value)

Figure 4.3.1.1 IOS to Control Loader Packet Structure

4.3.2 Control Loader to Flight Computer

The Control Loader communicates to the Flight Computer through the use of USB Data packets. The payload in that packet will contain the following values: Control Loader Heartbeat (integer value), Motion Base Status (integer value), Joystick Status (integer value), Throttle Status (integer value), Throttle Position (float value), Joystick X Position (float value), Joystick Y Position (float value), Motion Base Servo 1 Position (float value), and Motion Base Servo 2 Position (float value). **Figure 4.3.2** illustrates the payload packet structure.

Control Loader Heartbeat (integer value)	Motion Base Status (integer value, 1 or 0)	Joystick Status (integer value, 1 or 0)	Throttle Status (integer value, 1 or 0)	Throttle Position (float value)	Joystick X Position (float value)	Joystick Y Position (float value)	Motion Base Servo 1 Position (float value)	Motion Base Servo 2 Position (float value)

Figure 4.3.2 Control Loader to IOS Packet Structure

The Heartbeat value notifies the IOS that there is an active communication connection with the Control Loader itself. The Motion Base, Joystick, and Throttle Status values notify the IOS if those systems are online or not. A value of 0, is used if the Control Loader is unable to communicate with that device, while a value of 1 is used to designate the system is able to communicate. The health status of each system can be viewed at any time from the Health Page of the IOS.

4.3.3 X-Plane to IOS

The IOS and X-Plane software are both hosted on the same physical computer (Flight Computer). Communication between the two will be done utilizing UDP packets. **Figure 4.3.3** represents the data structure the IOS will receive from X-Plane. When X-Plane is configured on the Flight Computer, there is an internal setting that will allow it to send UDP packets to a specific Internet Protocol (IP) address and Port number. The only data values needed from X-Plane are the values for roll and pitch which are sent to the Control Loader in order to drive the Motion Base. There are two options to get this data from X-Plane: configure it to send data packets containing the roll and pitch which must be configured within X-Plane itself or configure it to send motion base data packets which contain the values of pitch, heading, roll, sideways acceleration, vertical acceleration, and longitudinal acceleration. It was decided to use motion base data packets even though it contains more information than required because, the extra values are needed for the implementation of a washout algorithm (please refer to **section 3.15** for further information), which if time permitting will be implemented in the project.

MOTI01 (header for motion base packet)	Pitch (float value)	Heading (float value)	Roll (float value)	Sideways Acceleration (float value)	Vertical Acceleration (float value)	Longitudinal Acceleration (float value)	1 (end of the packet)

Figure 4.3.3 X-Plane to IOS Packet Structure

The header of the packet accounts for bytes 1-6, the float values of the packet account for bytes 7-30, and the end of the packet accounts for byte 31, which results in a total byte count of 31 for each packet.

4.3.4 IOS to X-Plane

As mentioned in **Section 4.3.3** the IOS and X-Plane are both hosted on the Flight Computer. In order to communicate between the two applications, UDP packets will be used. **Figure 4.3.4.1** represents the data structure the IOS will send to X-Plane. This packet structure is able to be received and sent from X-Plane and it is possible to use in place of the packet structure described in **Section 4.3.3**, but was decided against in order to have the option to implement a washout algorithm (please refer to **section 3.15** for further information).

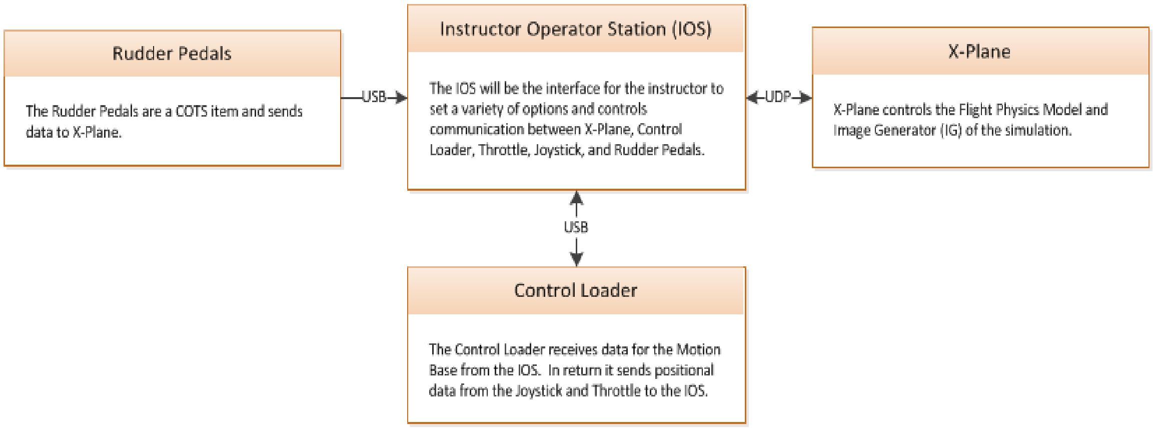
DATA01 (header for Data packet)	Index (integer value representing the index into the Data float array)	Data (float array of size 8, with each index representing settable data)	1 (end of the packet)

Figure 4.3.4 IOS to X-Plane Packet Structure

The header of the packet accounts for bytes 1-6, the index accounts for bytes 7-10, the data float array accounts for bytes 11-42, and the end of packets accounts for byte 43, which results in a total byte count of 43 for each packet.

4.4 Flight Computer

The Flight Computer will host the IOS and X-Plane as well as connect with the Rudder Pedals and Control Loader. **Figure 4.4.1** represents the communication connection for each of the systems. All communication to and from the Flight Computer will be handled by the IOS. The Rudder Pedals and Control Loader will interface with the Flight Computer via USB while X-Plane will interface via a local loop back utilizing UDP protocol.



Rudder Pedals

The Rudder Pedals are a COTS item and sends data to X-Plane.

Instructor Operator Station (IOS)

The IOS will be the interface for the instructor to set a variety of options and controls communication between X-Plane, Control Loader, Throttle, Joystick, and Rudder Pedals.

X-Plane

X-Plane controls the Flight Physics Model and Image Generator (IG) of the simulation.

Control Loader

The Control Loader receives data for the Motion Base from the IOS. In return it sends positional data from the Joystick and Throttle to the IOS.

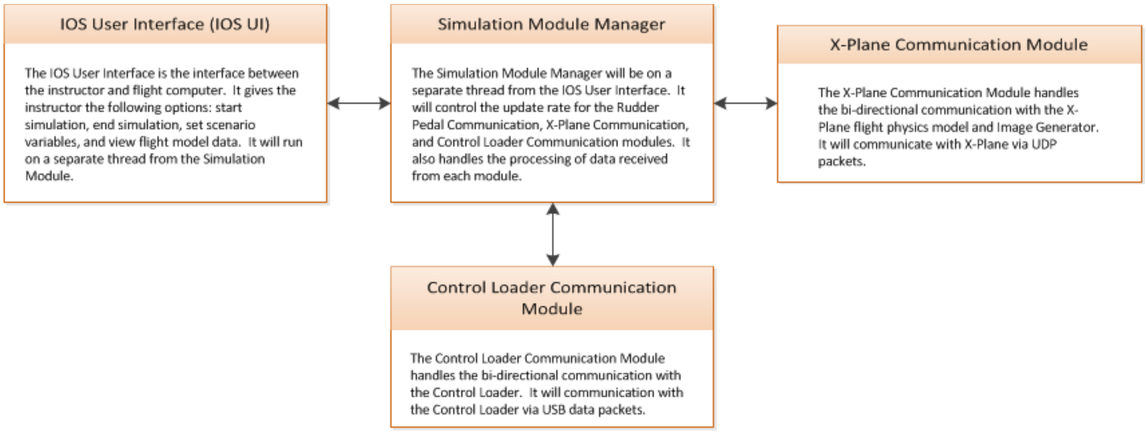
Figure 4.4.1 Flight Computer Systems

4.5 IOS – Instructor Operator Station

Overview

The IOS will be designed in modules to provide more maintainable code. Each communication module will be responsible for communication to a specific system in the IOS. The SimulationModuleManager will control the update rate of the modules to ensure that each module communicates at the same rate as the others. The update rate for the SimulationModuleManager will be in the range of 50 to 100 hertz, but is dependent upon the feel for the pilot. Since the update rate is set based on instructor feedback the design for the IOS will be implemented in such a way, the instructor will be able to change the update rate as needed.

Figure 4.5.1 represents the higher level architecture of the IOS software. The IOS User Interface and SimulationModuleManager will each run on a dedicated thread to allow for higher update rates (between 50 and 100 Hertz) without causing the IOS User Interface to become locked. The IOS has a clear distinction between the front end (IOS User Interface) and the back end (SimulationModuleManager). The front end will be implemented using an Object Oriented approach while the back end will follow a non-object oriented approach.



IOS User Interface (IOS UI)

The IOS User Interface is the interface between the instructor and flight computer. It gives the instructor the following options: start simulation, end simulation, set scenario variables, and view flight model data. It will run on a separate thread from the Simulation Module.

Simulation Module Manager

The Simulation Module Manager will be on a separate thread from the IOS User Interface. It will control the update rate for the Rudder Pedal Communication, X-Plane Communication, and Control Loader Communication modules. It also handles the processing of data received from each module.

X-Plane Communication Module

The X-Plane Communication Module handles the bi-directional communication with the X-Plane flight physics model and Image Generator. It will communicate with X-Plane via UDP packets.

Control Loader Communication Module

The Control Loader Communication Module handles the bi-directional communication with the Control Loader. It will communication with the Control Loader via USB data packets.

Figure 4.5.1 Main IOS modules

Application Flow

The overall flow of the IOS application is illustrated in **Figure 4.5.2**. When the Flight Computer is started, the IOS will be launched and will run on a dedicated monitor in full screen. When the IOS is launched, it first initializes the SimulationModuleManager, which will then have the XPlaneCommunicationModule and ControlLoaderCommunicationModule initialize communication with their associated system. Next the IOS will start X-Plane on a separate process and have it initialize into a waiting state. This was done to decrease start up time for X-Plane which can take up to approximately 15 minutes depending on user configuration for scenery fidelity of the virtual world. If X-Plane fails to launch, the user will be notified of the issue and the IOS will have to be restarted once the problem has been corrected. After the initial checks, the IOS will launch directly to the Scenario Page at which point it will be in a waiting state until the instructor starts the simulation. While the IOS is in a waiting state, the SimulationModuleManager will continue to run at a frame rate defined by a user configuration file and will send values of 0 for roll and pitch to the Control Loader. The positional values from the Control Loader will be ignored until the simulation session has been started. When the instructor clicks the Start Simulation button, the IOS will first check to make sure all scenario values are valid, if they are not, the IOS will notify the instructor and return to the waiting state. If the scenario values are valid, then the SimulationModuleManager StartSimulation method will be invoked, which sets its

SimulationActive property to true. Once this happens, the SimulationModuleManager will begin passing data between each of the systems until the instructor or pilots decides to end the current simulation session.

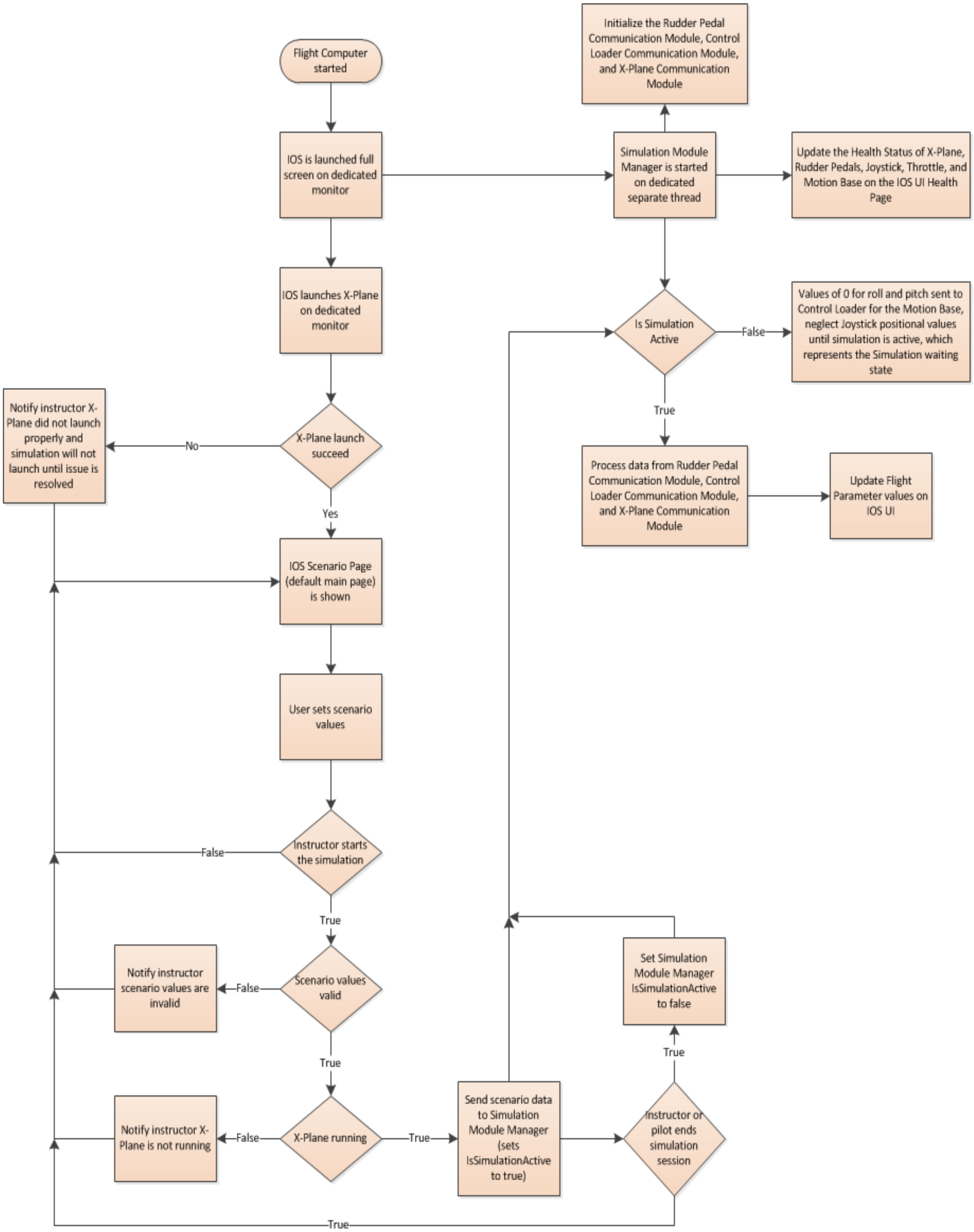

Flight Computer started
IOS is launched full screen on dedicated monitor
IOS launches X-Plane on dedicated monitor
X-Plane launch succeed
No
Notify instructor X-Plane did not launch properly and simulation will not launch until issue is resolved
Yes
IOS Scenario Page (default main page) is shown
User sets scenario values
Instructor starts the simulation
False
True
Scenario values valid
False
Notify instructor scenario values are invalid
True
X-Plane running
False
Notify instructor X-Plane is not running
True
Send scenario data to Simulation Module Manager (sets IsSimulationActive to true)
Instructor or pilot ends simulation session
True
Set Simulation Module Manager IsSimulationActive to false
True
Initialize the Rudder Pedal Communication Module, Control Loader Communication Module, and X-Plane Communication Module
Simulation Module Manager is started on dedicated separate thread
Update the Health Status of X-Plane, Rudder Pedals, Joystick, Throttle, and Motion Base on the IOS UI Health Page
Is Simulation Active
False
Values of 0 for roll and pitch sent to Control Loader for the Motion Base, neglect Joystick positional values until simulation is active, which represents the Simulation waiting state
True
Process data from Rudder Pedal Communication Module, Control Loader Communication Module, and X-Plane Communication Module
Update Flight Parameter values on IOS UI

Figure 4.5.2 Application Flow

Class Specifications

The IOS is developed with a clear distinction between the front end (IOS User Interface) and back end (Simulation data processing). In most simulator projects, the IOS software is a separate application from the simulation data processing software. Due to the budgeting limitations of the project, only a single computer is being used for the flight physics model, IG, and IOS, which classically would each run on a dedicated computer to allow the simulation to run at higher rates (between 50Hz and 100Hz). Since everything will be hosted on the same computer, it was decided for convenience to build the simulation data processing into the IOS. The choice to have a clear distinction between the two is due to the fact that the simulation data processing cannot be designed utilizing an object oriented approach due to the rate at which the computations most occur. Although the front end and back end follow different software architecture paradigms, all code will follow consistent conventions. All member level variables will be defined with a leading underscore character ("_"), all local variable declarations will be title case except for the leading character will always be lower case ("variableName"), and all public variables will be declared in title case ("VariableName"). The code will be organized from top to bottom in the following order: constant declarations, member variables, constructors, public read-only variables, public variables, public methods, member methods, and event handlers.

The entire IOS application is represented by a single Visual Studio 2010 Ultimate (VS2010) solution file. The solution is broken down into five separate projects; IOS_UserInterface, UserInterfaceControls, Simulator, CommunicationModules, and Utilities. The solution was broken up into the five projects to provide a more organized look of the software architecture, which will aide in the development phase of the project. Along with a more organized look, separate project files allow the option to break the software into separate individual applications if any unanticipated problems arise during development. It will provide the group with more flexibility and will allow the group to be better prepared to handle any unexpected issues that may arise during the development phase. The projects associated with the front end of the IOS software are: IOS_UserInterface, UserInterfaceControls, and Utilities. The projects associated with the back end of the IOS software are: Simulator, CommunicationModules, and Utilities.

Utilities Project

The Utilities project contains two separate directories: Communication and Simulation. This separation was done to provide higher organization for the Utilities project. The primary purpose of the Utilities project is to provide classes that can help reduce the complexity of code for the front and back end of the IOS software. Classes in the Communication directory are primarily used by the back end Communication Modules to provide an easy to use interface for communicating with peripheral devices. Classes in the Simulation directory are primarily used by the front end to provide an easy way to pass simulation data in the form of objects between modules of the user interface.

UDPConnection: The UDPConnection class provides an easy to use interface for communication with a single device via UDP. The only class to utilize the UDPConnection object is the XPlaneCommunication class. It was decided to exclude this functionality from the XPlaneCommunication class because there could be a future need for UDP communication. If during the testing phase, the Flight Computer is deemed not powerful enough to run all the systems, this class will allow the functions of the Flight Computer to be easily spread across multiple machines without requiring heavy re-coding. **Table 4.5.1** outlines the specifications of the UDPConnection class.

There are two overloaded constructors that are available, the first requires no parameter inputs and creates the new object using the default values as described earlier. The second takes in three parameters: address, sendPort, and receivePort, which set the _address, _sendPort, and _receivePort variables.

It contains five member properties: _address, _sendPort, _receivePort, _udpClient, and _active. The _address variable is of type String and represents the Internet Protocol (IP) address of the device to communicate with in the form of “xxx.xxx.xxx.xxx” and defaults to “127.0.0.1” which is for the local loop back and was chosen as the default value, because all systems requiring UDP communication will be hosted on the same physical computer. The _sendPort variable is of type Integer and represents the port number to use for sending data to the desired device and defaults to a value of “4”, because this is the first unassigned UPD port number as defined by the Internet Assigned Numbers Authority (IANA). The _receivePort variable is of type Integer and represents the port number to use for receiving data from the desired device and defaults to a value of “6”, because this is the second unassigned UDP port number as defined by the IANA. The _udpClient is of type UdpClient which is defined by the .NET framework and provides UDP network services. The _udpClient object is responsible for sending and receiving the data from the specified device at the defined _address, _sendPort, and _receivePort. The _active is of type Boolean and represents if the UDP connection is active. It is used for the Active property getter since that public property is read-only.

There is one public read-only property: Active. The Active property is of type Boolean and is implemented using a getter to ensure it is read-only. If the UDP connection is active, a value of true will be returned, otherwise false will be returned.

There are three public properties: Address, SendPort, and ReceivePort, which return the values of _address, _sendPort, and _receivePort. These public properties are represented in the code as getter and setters. When either of these public properties is set, the current established connection will be closed and a new connection will be established.

There are two public methods that can be called: GetData and SendData. The GetData method takes in no parameters as input and returns a byte array data structure which represents the data received from the device at the defined Address and ReceivePort. The SendData method takes in an input parameter data of type byte array which represents the data to send to the defined Address and SendPort of the device and does not have a return value.

There is one member method: InitConnection. The InitConnection method takes in three parameters as inputs: address, sendPort, and receivePort which are used to establish the UDP connection. When this method is called, a check to see if the _udpClient has been initialized (not equal to null), if it has not been initialized then it will be and the connection will be attempted. If _udpClient was already initialized, the current connection will be closed, then _udpClient will be re-created and the new connection will be attempted. The InitConnections member method gets called when a new instance of the UDPConnection class is instantiated, the Address is changed, the SendPort is changed, or the ReceivePort is changed. The setter for the Address property first checks to see if the new value is the same as _address, if they are not the same, then the InitConnection is called passing in the new value for the Address and the value of the Port property. The setter for the SendPort property first checks to see if the new value is the same as _sendPort, if they are not the same, then the InitConnection is called passing in the new value for SendPort and the value of the Address property. The setter for the ReceivePort property first checks to see if the new value is the same as _receivePort, if they are not the same, then the InitConnection is called passing in the new value for ReceivePort and the value of SendPort and Address.

Class Name: UDPConnection	**Description:** Provides an easy interface to communicate with UDP devices.
Instance Constructors	**Description**
UPDConnection	Requires no inputs and instantiates a new instance of the UDPConnection class using the default values.
UPDConnection	Requires three input parameters: address, sendPort, and receivePort.
Member Properties	**Description**
_address	String representing the IP address of the device to communicate with
_sendPort	Integer representing the port number to use for sending data
_receivePort	Integer representing the port number to use for receiving data
_active	Boolean value representing if the UDP connection is active. True if active, false otherwise.
_udpClient	UdpClient object which provides UDP network services
Public Read-Only Properties	**Description**
Active	Boolean value representing if the UDP connection is active. True if active, false otherwise.
Public Properties	**Description**
Address	String representing the IP address of the device to communicate with
SendPort	Integer representing the port number to use for sending data
ReceivePort	Integer representing the port number to use for receiving data
Public Methods	**Description**

SendData	Takes in a byte array parameter and transmits it over UDP to the specified IP address
GetData	Returns a byte array that was received from the specified IP address
Member Methods	**Description**
InitConnection	Requires three input parameters: address (String value representing the IP address), sendPort (Integer value representing the port to send on), and receivePort (Integer value representing the port to receive on). Establishes a connection with the device defined by the address, sendPort, and receivePort.

Table 4.5.1 UDPConnection Specifications

USBConnection: The USBConnection provides an easy to use interface to communicate with a single device via USB. The design purpose for this class is to provide an easy interface to communicate with generic USB devices. The ControlLoaderCommunication class is the only class to utilize the USBConnection class, but this design allows it to be re-used if there is a need during the development of the project. **Table 4.5.2** outlines the specifications of the USBConnection class.

There are two overloaded constructors that are available, the first requires no parameter input and creates the new object using the default values as described earlier. The second takes in two parameters: vendorId and productId of type Integer, which set the _vendorId and _productId properties.

It class contains six member properties: _vendorId, _productId, _usbDevice, _usbReader, _usbWriter, and _active. The _vendorId property is of type Integer and represents the vendor identification of the USB device to communicate with. The _productId property is of type Integer and represents he product identification of the USB device to communicate with. By default both the _vendorId and _productId are set to a value of 0, because the properties must be declared. The _usbDevice property is of type UsbDevice and is defined by the LibUsbDotNet 2.2.8 library and provides non-driver specific communication to USB devices. The _usbReader property is of type UsbEndPointReader and is defined by the LibUsbDotNet 2.2.8 library. It provides the ability to retrieve data by either a call to one of its overloaded Read methods or from a DataReceived event. In this implementation data will be retrieved using the Read method, since it is desired to reduce overhead involved with event handling. The _usbWriter property is of type UsbEndpointWriter and is defined by the LibUsbDotNet 2.2.8 library. It provides the ability to write data to an endpoint by using one of its overloaded Write methods. The _active property is of type Boolean and represents if the USB connection is active. True if the USB connection is active and false otherwise. Both the _usbReader and _usbWriter require the _usbDevice to be initialized first, because the _usbDevice initializes the _usbReader by calling its OpenEndPointReader method and initializes the _usbWriter by calling its OpenEndPointWriter method.

There is one public read-only property: Active. The Active property is of type Boolean and is implemented using a getter to ensure it is read-only. It represents whether there is an active USB communication connection and returns the value of the _active member property. If the USB connection is active, true is returned, otherwise false is returned.

There are two public parameters: VendorId and ProductId. The VendorId property is of type Integer and is implemented using a getter to ensure it is read-only. It represents the vendor identification of the USB device to communicate with and returns the value of the _vendorId member property. The ProductId property is of type Integer and is implemented using a getter to ensure it is read-only. It represents the product identification of the USB device to communicate with and returns the value of the _productId member property.

There are two public methods: GetData and SendData. The GetData method takes in no parameters as input and returns a byte array data structure which represents the data received from the USB device defined by the VendorId and ProductId. The SendData method takes in an input parameter of type byte array which represents the data to send to the USB device defined by the VendorId and ProductId and does not have a return value.

There is one member method: InitConnection. The InitConnection method takes in two parameters: vendorId and productId, which are used to establish the USB connection. When this method is called, a check to see if the _usbDevice, _usbReader, and _usbWriter have been initialized (not equal to null), if either of these have not been initialized, then first the _usbDevice will be initialized and it will be used to initialize the _usbReader and _usbWriter as described earlier. If all of these have been initialized, the USB connection will be closed, then _usbDevice will be re-created along with _usbReader and _usbWriter, and then the new USB connection will be attempted. The InitConnection member method gets called when a new instance of the USBConnection class is instantiated.

Class Name: USBConnection	**Description:** Provides an easy interface to communication with USB devices.
Instance Constructors	**Description**
USBConnection	Requires no inputs and instantiates a new instance of the USBConnection class using the default values.
USBConnection	Requires two input parameters: vendorId and productId.
Member Properties	**Description**
_vendorId	Integer value representing the vendor identification number of the connect to
_productId	Integer value representing the product identification number of the device to connect to
_active	Boolean value representing if the USB connection is active. True if active, false

	otherwise.
_usbDevice	UsbDevice object defined by LibUsbDotNet and is responsible for communicating with the specified USB device
_usbReader	UsbEndpointReader object defined by LibUsbDotNet and is responsible for reading from an USB endpoint
_usbWriter	UsbEndpointWriter object defined by LibUsbDotNet and is responsible for writing to an USB endpoint
Public Read-Only Properties	**Description**
Active	Boolean value representing if the USB connection is active. True if active, false otherwise.
Public Properties	**Description**
VendorId	String value representing the vendor identification number of the connect to
ProductId	String value representing the product identification number of the device to connect to
Public Methods	**Description**
SendData	Takes in a byte array parameters and transmits it over USB to the specified device
GetData	Returns a byte array that was received from the specified USB device
Member Methods	**Description**
InitConnection	Requires two inputs of type Integer: vendorId and productId. Establishes a connection with the USB device defined by vendorId and productId.

Table 4.5.2 USBConnection Specifications

Atmosphere: The Atmosphere class represents settable parameters related to atmospheric conditions for the instructor and inherits from the Object (.NET Framework) base class. The design purpose of this class was to lessen the code complexity of gathering atmospheric data from the user interface in order to have the option to save/load atmosphere settings from a file. **Table 4.5.3** outlines the specifications for the Atmosphere class.

There are three overloaded constructors available, the first requires no parameter inputs and creates the new object using the default values. The second takes in seven properties: thermalTops, thermalCoverage, thermalClimbRate, closestAirportTemperature, seaLevelBarometricPressure, visibility, and precipitation, which set the values for the public properties. The third constructor takes in a single parameter: fromString, which is of type String and contains the string representation of the Atmosphere class.

There are seven public properties: ThermalTops, ThermalCoverage, ThermalClimbRate, ClosestAirportTemperature, SeaLevelBarometricPressure, Visibility, and Precipitation. The ThermalTops property is of type Integer and represents the height above sea level in feet of the thermal tops. The ThermalCoverage property is of type Integer and represents the thermal coverage as a percent. The ThermalClimbRate property is of type Integer and represents the thermal climb rate in feet per minute. The ClosesAirportTemperature property is of type Integer and represents the temperature in Fahrenheit of the closest airport to the current position of the aircraft. The SeaLevelBarometricPressure property is of type Integer and represents the barometric pressure at sea level in inches of mercury. The Visibility property is of type Integer and represents the visibility as seen by the pilot. The Precipitation property is of type PrecipitationTypes and represents one of the defined PrecipitationTypes values correlating to the amount of desired precipitation.

There is one public method: ToString, which is an overridden method from the Object base class. This method returns the String representation of the Atmosphere class public properties formatted at "Atmosphere: ThermalTops value, ThermalCoverage value, ThermalClimbRate value, ClosestAirportTemperature value, SeaLevelBarometricPressure value, Visibility value, Precipitation value". This was done to allow the Atmosphere class be easily formatted as a String so that it could be saved to file without much effort required.

There is one member method: FromString, which takes in a single parameter fromString of type String that is formatted as described by the ToString method. This method is called by the overloaded constructor that has the fromString String input parameter. When this method is called, it will first check that the fromString String is not equal to an empty String ("") or equal to null. Next it will check that fromString contains the substring "Atmosphere:" and either of these checks fails, the method will return. If the checks pass, the FromString method will remove the substring "Atmosphere:" therefore leaving only the public property values separated by the "," character ("ThermalTops value, ThermalCoverage value, ThermalClimbRate value, ClosestAirportTemperature value, SeaLevelBarometricPressure value, Visibility value, Precipitation value"). Next the remaining String will be split on the "," character which then will represent the String as an array of Strings, which index holding a public property value. If length of this String array is not equal to seven, then it means there is invalid data and the FromString method will return. If this check then passes, indexes zero through five will be converted from String values to Integer values, and then will set their coordinating public properties. The last index (six) will be converted to a PrecipitationTypes value. If all the conversions from String to the required types pass, then the method has completed successfully.

Class Name: Atmosphere	**Description:** Represents all parameters related to atmosphere conditions and inherits from Object (.NET Framework)

Instance Constructors	**Description**
Atmosphere	Requires no inputs and instantiates a new instance of the Atmosphere class using the default values.
Atmosphere	Requires seven parameters: thermalTops, thermalCoverage, thermalClimbRate, closestAirportTemperature, seaLevelBarometricPressure, visibility, and precipitation
Atmosphere	Requires one input parameter: fromString.
Public Properties	**Description**
ThermalTops	Integer value representing the thermal tops in feet above sea level
ThermalCoverage	Integer value representing the thermal coverage as a percentage
ThermalClimbRate	Integer value representing the thermal climb rate in feet per minute
ClosestAirportTemperature	Integer value representing the temperature at the closet airport to the aircraft in Fahrenheit
SeaLevelBarometricPressure	Integer value representing the barometric pressure at sea level in inches of mercury
Visibility	Integer value representing the 57visibility as viewed by the pilot
Precipitation	Amount of precipitation, must be a PrecipitationTypes
Public Methods	**Description**
ToString	Overridden method from the Object base class that returns a String representing the Atmosphere class in the format “Atmosphere: ThermalTops value, ThermalCoverage value, ThermalClimbRate value, ClosestAirportTemperature value, SeaLevelBarometricPressure value, Visibility value, Precipitation value”
Member Methods	**Description**
FromString	Requires a String parameter: fromString that is formatted as described by the ToString method and sets the parameters based on the value in the String.

Table 4.5.3 Atmosphere Specifications

CloudCover: The CloudCover class represents all settable parameters related to clouds and inherits from the Object (.NET Framework) base class. The design purpose for this class was to lessen the code complexity of gathering cloud data from the user interface in order to have the option to save/load cloud settings from a file. **Table 4.5.4** outlines the specifications for the CloudCover class.

There are three overloaded constructors available, the first requires no parameter inputs and creates the new object using the default values. The second takes in four parameter: coverType, topMsl, baseMsl, and aboveGroundLevel, which set the values for the coordinating public properties. The third constructor takes in a single parameter: fromString, which is of type String and contains the string representation of the CloudCover class.

There are four public properties: CoverType, TopMsl, BaseMsl, and AboveGroundLevel. The CoverType property is of type CloudCoverTypes and represents one of the defined CloudCoverTypes values correlating to the possible types of cloud covering available. The TopMsl property is of type Integer and represents the height of the cloud tops above sea level in miles. The BaseMsl property is of type Integer and represents the height of the cloud base above sea level in miles. The AboveGroundLevel property is of type Integer and represents the height above ground level of the clouds in miles.

There is one public method: ToString, which is an overridden method from the Object base class. This method returns the String representation of the CloudCover class public properties formatted as “CloudCover: CoverType value, TopMsl value, BaseMsl value, AboveGroundLevel value”. This was done to allow the CloudCover class to be easily formatted as a String so that it could be saved to a file without much effort required.

There is one member method: FromString, which takes in a single parameter fromString of type String that is formatted as described by the ToString method. This method is called by the overloaded constructor that has the fromString String input parameter. When this method is called, it will first check that the fromString String is not equal to an empty String (“”) or equal to null. Next it will check that fromString contains the substring “CloudCover:” and if either of these checks fails, the method will return. If the checks pass, the FromString method will remove the substring “CloudCover:” therefore leaving only the public property values separated by the “,” character (“CoverType value, TopMsl value, BaseMsl value, AboveGroundLevel value”). Next the remaining String will be split on the “,” character which then will represent the String as an array of Strings, with each index holding a public property value. If length of this String array is not equal to four, then it means there is invalid data and the FromString method will return. If this check then passes, indexes one through three will be converted from String values to Integer values then will set their coordinating public properties. The first index (zero) will be converted to a CloudCoverTypes value. If all the conversions from String to the required types pass, then the method has completed successfully.

Class Name: CloudCover	**Description:** Represents all parameters involving clouds
Instance Constructors	**Description**
CloudCover	Requires no inputs and instantiates a new instance of the CloudCover class using the default values.
CloudCover	Requires four input parameters: coverType, topMsl, baseMsl, and aboveGroundLevel.
CloudCover	Requires one input parameter: fromString.

Public Properties	**Description**
CoverType	Type of cloud cover for the simulation, must be a CloudCoverTypes
TopMsl	Integer value representing the miles above sea level of the cloud tops
BaseMsl	Integer value representing the miles above sea level of the cloud base
AboveGroundLevel	Integer value representing the miles above ground level of the clouds
Public Methods	**Description**
ToString	Overridden method from the Object base class that returns a String representing the CloudCover class in the format "CloudCover: CoverType value, TopMsl value, BaseMsl value, AboveGroundLevel value"
Member Methods	**Description**
FromString	Requires a String parameter: fromString that is formatted as described by the ToString method and sets the parameters based on the value in the String.

Table 4.5.4 CloudCover Specifications

Runway: The Runway class represents all the settable parameters related to the runway and inherits from the Object (.NET Framework) base class. The design purpose of this class is to lessen the code complexity of gathering runway data from the user interface in order to have the option to save/load runway settings from a file. **Table 4.5.5** outlines the specifications for the Runway class.

There are three overloaded constructors available, the first requires no parameter inputs and creates the new object using the default values. The second takes in two parameters: runwayCondition and runwayName, which set the values for their coordinating public properties. The third constructor takes in a single parameter: fromString, which is of type String and contains the string representation of the Runway class.

There are two public properties: RunwayCondition and RunwayName. The RunwayCondition property is of type RunwayConditionTypes and represents one of the defined RunwayConditionTypes values correlating to the possible types of runway conditions available. The RunwayName property is of type String and represents the name of a runway.

There is one public method: ToString, which is an overridden method from the Object base class. This method returns the String representation of the Runway class public properties formatted as “Runway: RunwayCondition value, RunwayName value”. This was done to allow the Runway class to be easily formatted as a String so that it could be saved to a file without much effort required.

There is one member method: FromString, which takes in a single parameter fromString of type String that is formatted as described by the ToString method. This method is called by the overloaded constructor that has the fromString String input parameter. When this method is called, it will first check that the fromString String is not equal to an empty String (“”) or equal to null. Next it will check that fromString contains the substring “Runway:” and if either of these checks fails, the method will return. If the checks pass, the FromString method will remove the substring “Runway:” therefore leaving only the public property values separated by the “,” character (“RunwayCondition value, RunwayName value”). Next the remaining String will be split on the “,” character which then will represent the String as an array of Strings, with each index holding a public property value. If length of this String array is not equal to two, then there is invalid data and the FromString method will return. If this check passes, index zero will be converted to a RunwayConditionTypes value and set the RunwayCondition public property. Index one will be set as the RunwayName public property. If all the conversions from String to the required types pass, then the method has completed successfully.

Class Name: Runway	**Description:** Represents all parameters related to the runway
Instance Constructors	**Description**
Runway	Requires no inputs and instantiates a new instance of the Runway class
Runway	Requires two input parameters: runwayCondition and runwayName
Runway	Requires one input parameter: fromString
Public Properties	**Description**
RunwayCondition	Type of runway condition, must be a RunwayConditionTypes.
RunwayName	String representing the name of the runway
Public Methods	**Description**
ToString	Overridden method from the Object base class that returns a String representing the Atmosphere class in the format "Runway: RunwayCondition value, RunwayName value"
Member Methods	**Description**
FromString	Requires a String parameter: fromString that is formatted as described by the ToString method and sets the parameters based on the value in the String.

Table 4.5.5 Runway Specifications

Wind: The Wind class represents all the settable parameters related to wind and inherits from the Object (.NET Framework) base class. The design purpose of this class is to lessen the code complexity of gathering wind data from the user interface in order to have the option to save/load wind settings from a file. **Table 4.5.6** outlines the specifications for the Wind class.

There are three overloaded constructors available, the first requires no parameter inputs and creates the new object using the default values. The second takes in eighteen parameters: highAltitudeWindLayer, highAltitudeWindDirection, highAltitudeWindSpeed, highAltitudeWindSheer, highAltitudeWindSheerDirection, highAltitudeWindTurbulence, midAltitudeWindLayer, midAltitudeWindDirection, midAltitudeWindSpeed, midAltitudeWindSheer, midAltitudeWindSheerDirection, midAltitudeWindTurbulence, lowAltitudeWindLayer, lowAltitudeWindDirection, lowAltitudeWindSpeed, lowAltitudeWindSheer, lowAltitudeWindSheerDirection, and lowAltitudeWindTurbulence, which set the values for their coordinating public parameters. The third constructor takes in a single parameter: fromString, which is of type String and contains the string representation of the Wind class.

There are eighteen public properties: HighAltitudeWindLayer, HighAltitudeWindDirection, HighAltitudeWindSpeed, HighAltitudeWindSheer, HighAltitudeWindSheerDirection, HighAltitudeWindTurbulence, MidAltitudeWindLayer, MidAltitudeWindDirection, MidAltitudeWindSpeed, MidAltitudeWindSheer, MidAltitudeWindSheerDirection, MidAltitudeWindTurbulence, LowAltitudeWindLayer, LowAltitudeWindDirection, LowAltitudeWindSpeed, LowAltitudeWindSheer, LowAltitudeWindSheerDirection, and LowAltitudeWindTurbulence. The HighAltitudeWindLayer property is of type Integer and represents the height above sea level for high altitude winds in feet. The HighAltitudeWindDirection property is of type Integer and represents the direction of the high altitude winds in degrees. The HighAltitudeWindSpeed property is of type Integer and represents the velocity of high altitude winds in knots. The HighAltitudeWindSheer is of type Integer and represents the sheer of the high altitude winds in knots. The HighAltitudeWindSheerDirection is of type Integer and represents the direction of the HighAltitudeWindSheer in degrees. The HighAltitudeWindTurbulence is of type Integer and represents the intensity of the turbulence for the HighAltitudeWindLayer. The MidAltitudeWindLayer property is of type Integer and represents the height above sea level for high altitude winds in feet. The MidAltitudeWindDirection property is of type Integer and represents the direction of the high altitude winds in degrees. The MidAltitudeWindSpeed property is of type Integer and represents the velocity of high altitude winds in knots. The MidAltitudeWindSheer is of type Integer and represents the sheer of the high altitude winds in knots. The MidAltitudeWindSheerDirection is of type Integer and represents the direction of the MidAltitudeWindSheer in degrees. The MidAltitudeWindTurbulence is of type Integer and represents the intensity of the turbulence for the MidAltitudeWindLayer.

The LowAltitudeWindLayer property is of type Integer and represents the height above sea level for high altitude winds in feet. The LowAltitudeWindDirection property is of type Integer and represents the direction of the high altitude winds in degrees. The LowAltitudeWindSpeed property is of type Integer and represents the velocity of high altitude winds in knots. The LowAltitudeWindSheer is of type Integer and represents the sheer of the high altitude winds in knots. The LowAltitudeWindSheerDirection is of type Integer and represents the direction of the LowAltitudeWindSheer in degrees. The LowAltitudeWindTurbulence is of type Integer and represents the intensity of the turbulence for the LowAltitudeWindLayer.

There is one public method: ToString, which is an overridden method from the Object base class. This method returns the String representation of the Wind class public properties formatted as "Wind: HighAltitudeWindLayer value, HighAltitudeWindDirection value, HighAltitudeWindSpeed value, HighAltitudeWindSheer value, HighAltitudeWindSheerDirection value, HighAltitudeWindTurbulence value, MidAltitudeWindLayer value, MidAltitudeWindDirection value, MidAltitudeWindSpeed value, MidAltitudeWindSheer value, MidAltitudeWindSheerDirection value, MidAltitudeWindTurbulence value, LowAltitudeWindLayer value, LowAltitudeWindDirection value, LowAltitudeWindSpeed value, LowAltitudeWindSheer value, LowAltitudeWindSheerDirection value, LowAltitudeWindTurbulence value". This was done to allow the Wind class to be easily formatted as a String so that it can be saved to a file without much effort required.

There is one member method: FromString, which takes in a single parameter fromString of type String that is formatted as described by the ToString method. This method is called by the overloaded constructor that has the fromString String input parameter. When this method is called, it will first check that the fromString String is not equal to an empty String ("") or equal to null. Next it will check that fromString contains the substring "Wind:" and if either of these checks fails, the method will return. If the checks pass, the FromString method will remove the substring "Wind:" therefore leaving only the public property values separated by the "," character ("HighAltitudeWindLayer value, HighAltitudeWindDirection value, HighAltitudeWindSpeed value, HighAltitudeWindSheer value, HighAltitudeWindSheerDirection value, HighAltitudeWindTurbulence value, MidAltitudeWindLayer value, MidAltitudeWindDirection value, MidAltitudeWindSpeed value, MidAltitudeWindSheer value, MidAltitudeWindSheerDirection value, MidAltitudeWindTurbulence value, LowAltitudeWindLayer value, LowAltitudeWindDirection value, LowAltitudeWindSpeed value, LowAltitudeWindSheer value, LowAltitudeWindSheerDirection value, LowAltitudeWindTurbulence value"). Next the remaining String will be split on the "," character which then will represent the String as an array of Strings, with each index holding a public property value. If length of this String array is not equal to eighteen, then there is invalid data and the FromString method will return. If this check passes, indexes zero through seventeen will be converted from Strings to Integer values and set there coordinating public property values. If all the conversions from String to the required types pass, then the method has completed successfully.

Class Name: Wind	**Description:** Represents all parameters involving wind conditions
Instance Constructors	**Description**
Wind	Requires no input parameters and instantiates a new instance of the Wind class

Wind	Requires eighteen input parameters: highAltitudeWindLayer, highAltitudeWindDirection, highAltitudeWindSpeed, highAltitudeWindSheer, highAltitudeWindSheerDirection, highAltitudeWindTurbulence, midAltitudeWindLayer, midAltitudeWindDirection, midAltitudeWindSpeed, midAltitudeWindSheer, midAltitudeWindSheerDirection, midAltitudeWindTurbulence, lowAltitudeWindLayer, lowAltitudeWindDirection, lowAltitudeWindSpeed, lowAltitudeWindSheer, lowAltitudeWindSheerDirection, and lowAltitudeWindTurbulence
Wind	Requires one input parameter: fromString
Public Properties	**Description**
HighAltitudeWindLayer	Integer value representing the feet above sea level for High altitude winds
HighAltitudeWindDirection	Integer value representing the direction of the HighAltitudeWindLayer in degrees
HighAltitudeWindSpeed	Integer value representing the velocity of the HighAltitudeWindLayer in knots
HighAltitudeWIndSheer	Integer value representing the sheer of the HighAltitudeWindLayer in knots
HighAltitudeWindSheerDirection	Integer value representing the direction of the HighAltitudeWindSheer in degrees
HighAltitudeWindTurbulence	Integer value representing the intensity of turbulence for the HighAltitudeWindLayer
MidAltitudeWindLayer	Integer value representing the feet above sea level for Mid altitude winds
MidAltitudeWindDirection	Integer value representing the direction of the MidAltitudeWindLayer in degrees
MidAltitudeWindSpeed	Integer value representing the velocity of the MidAltitudeWindLayer in knots
MidAltitudeWindSheer	Integer value representing the sheer of the MidAltitudeWindLayer in knots
MidAltitudeWindSheerDirection	Integer value representing the direction of the MidAltitudeWindSheer in degrees
MidAltitudeWindTurbulence	Integer value representing the intensity of turbulence for the MidAltitudeWindLayer

LowAltitudeWindLayer	Integer value representing the feet above sea level for Low altitude winds
LowAltitudeWindDirection	Integer value representing the direction of the LowAltitudeWindLayer in degrees
LowAltitudeWindSpeed	Integer value representing the velocity of the LowAltitudeWindLayer in knots
LowAltitudeWindSheer	Integer value representing the sheer of the LowAltitudeWindLayer in knots
LowAltitudeWindSheerDirection	Integer value representing the direction of the LowAltitudeWindSheer in degrees
LowAltitudeWindTurbulence	Integer value representing the intensity of turbulence for the LowAltitudeWindLayer
Public Methods	**Description**
ToString	Overridden method from the Object base class that returns a String representing the Atmosphere class in the format "Wind: HighAltitudeWindLayer value, HighAltitudeWindDirection value, HighAltitudeWindSpeed value, HighAltitudeWindSheer value, HighAltitudeWindSheerDirection value, HighAltitudeWindTurbulence value, MidAltitudeWindLayer value, MidAltitudeWindDirection value, MidAltitudeWindSpeed value, MidAltitudeWindSheer value, MidAltitudeWindSheerDirection value, MidAltitudeWindTurbulence value, LowAltitudeWindLayer value, LowAltitudeWindDirection value, LowAltitudeWindSpeed value, LowAltitudeWindSheer value, LowAltitudeWindSheerDirection value, LowAltitudeWindTurbulence value"
Member Methods	**Description**
FromString	Requires a String parameter: fromString that is formatted as described by the ToString method and sets the parameters based on the value in the String.

Table 4.5.6 Wind Specifications

Scenario: The Scenario class represents all the settable parameters for a simulation and inherits from the Object (.NET Framework) base class. This class contains all the information from the Atmosphere, CloudCover, Runway, and Wind classes and was designed to provide a clean approach to be able to save or load a scenario from file. **Table 4.5.7** outlines the specifications for the Scenario class.

There are three overloaded constructors, the first requires no parameter inputs and creates the new object using the default values. The second constructor requires four parameters: runway, atmosphere, cloudCover, and Wind, which set values for their coordinating public parameters. The third constructor requires a single parameter: fromString, which is of type String and contains the string representation of the Scenario class.

There are four public properties: Runway, Atmosphere, CloudCover, and Wind. The Runway property is of type Runway and represents all the property related to a runway as discussed in the Runway class specification. The Atmosphere property is of type Atmosphere and represents all the properties related to the atmosphere as discussed in the Atmosphere class specification. The CloudCover property is of type CloudCover and represents all the parameters related to the clouds as discussed in the CloudCover class specification. The Wind property is of type Wind and represents all the parameters related to wind as discussed in the Wind class specification.

There is one public method: ToString, which is an overridden method from the Object base class. This method returns the String representation of the Scenario class. It invokes the ToString method on each of the public properties and inserts a new line ("\n") between each one, then returns the aggregated String. The purpose of having each of the public property object implement a ToString method was to decrease the code complexity in the Scenario ToString method, which will make testing the IOS software much more efficient. The return String from the ToString method is formatted as "Runway.ToString()\nAtmosphere.ToString()\nCloudCover.ToString()\nWind.ToString()".

There is one member method: FromString, which takes in a single parameter fromString of type String that is formatted as described by the ToString method. This method is called by the overloaded constructor that has the fromString String input parameter. When this method is invoked, it will first check that the fromString String is not equal to an empty String ("") or equal to null. Next it will split the fromString on the "\n" new line character which will return a String array. If the length of the String array is not equal to four, then there is invalid data and the method will then notify the user interface that an error has occurred loading the scenario data. If the check passes, then the FromString method will next instantiate new instances for each of the public properties by invoking their constructors that requires the fromString parameter. If any of the public property objects fails when attempting to create from a String, they will load with their default data. After all public property objects have been instantiated, the FromString method has completed.

Class Name: Scenario	**Description:** Represents all parameters related to the scenario for a simulation session
Instance Constructors	**Description**
Scenario	Requires no input parameters and instantiates a new instance of the Scenario class
Scenario	Requires four input parameters: runway, atmosphere, cloudCover, and

	Wind
Scenario	Requires one input parameter: fromString
Public Properties	**Description**
Runway	Runway object representing all the parameters for the runway
Atmosphere	Atmosphere object representing all the parameters related to atmosphere conditions
CloudCover	CloudCover object representing all the parameters related to cloud covering
Wind	Wind object representing all the parameters related to wind conditions
Public Methods	**Description**
ToString	Overridden method from the Object base class that returns a String representing the Atmosphere class in the format "Runway.ToString()\nAtmosphere.ToString()\nCloudCover.ToString()\nWind.ToString()"
Member Methods	**Description**
FromString	Requires a String parameter: fromString that is formatted as described by the ToString method and sets the parameters based on the value in the String.

Table 4.5.7 Scenario Specifications

CloudCoverTypes: The CloudCoverTypes class contains an enumeration that represents each of the possible cloud cover types as defined by X-Plane. The types are as follows: Clear, HiCirrus, LowCirrus, LowStratus, CumulosScattered, CumulosBroken, and CumulosOvercast. The purpose for this class is to ensure correctness of data, since the instructor has a finite number of options for setting the CoverType property in the CloudCover object, so that the IOS software is more guarded against errors due to corrupt data. **Table 4.5.8** outlines the enumeration types for the CloudCoverTypes class.

Class Name: CloudCoverTypes	**Description:** Contains the possible types of cloud covers for the simulation
Enumeration Types	**Description**
Clear	Represents clear cloud covering
HiCirrus	Represents high cirrus cloud covering
LowCirrus	Represents low cirrus cloud covering
LowStratus	Represents low stratus cloud covering
CumulosScattered	Represents cumulus scattered cloud covering
CumulosBroken	Represents cumulus broken cloud covering
CumulosOvercast	Represents cumulus overcast cloud covering

Table 4.5.8 CloudCoverTypes Specifications

PrecipitationTypes: The PrecipitationTypes class contains an enumeration that represents each of the possible precipitation options as defined by X-Plane. The types are as follows: Light, Moderate, Heavy, and Severe. The purpose for this class is to ensure correctness of data since the instructor has a finite number of options for setting the Precipitation property in the Atmosphere object, so that the IOS software is more guarded against errors due to corrupt data. **Table 4.5.9** outlines the enumeration types for the PrecipitationTypes class.

Class Name: PrecipitationTypes	**Description:** Represents the types of precipitation for the simulation
Enumeration Types	**Description**
Light	Represents light precipitation
Moderate	Represents moderate precipitation
Heavy	Represents heavy precipitation
Sever	Represents severe precipitation

Table 4.5.9 PrecipitationTypes Specifications

RunwayConditionTypes: The RunwayConditionTypes class contains an enumeration that represents each of the possible runway condition options as defined by X-Plane. The types are as follows: CleanAndDry, Damp, and Wet. The purpose for this class is to ensure correctness of data, since the instructor has a finite number of options for setting the RunwayCondition property in the Runway object, so that the IOS software is more guarded against errors due to corrupt data. **Table 4.5.10** outlines the enumeration types for the RunwayConditionTypes class.

Class Name: RunwayConditionTypes	**Description:** Represents the types of runways conditions for the simulation
Public Properties	**Description**
CleanAndDry	Represents a clean and dry runway for the simulation
Damp	Represents a damp runway
Wet	Represents a wet runway

Table 4.5.10 RunwayConditionTypes Specifications

CommunicationModules Project

The CommunicationModules project contains class's specific to communication with the peripheral devices of the simulator and provide data structures to represent the data being transmitted. The DataPackets directory contains classes that represent the data being transmitted and provides an easy to use interface to build data packets and manipulate them. The design purpose was to reduce code complexity by breaking this functionality into smaller pieces and by doing so, it will help reduce time in the testing phase of the project since each piece is less complicated therefore easier to test. The classes not located in the DataPackets directory are specific to the back end of the IOS software and favor a more non object oriented approach. Although objects are used within the classes, the classes themselves are all static and contain static methods/properties. The design purpose of this was to reduce the complexity during execution of the software by reducing overhead involved with instantiating new instances of an object. This portion of the IOS software must run at a rate between 50-100Hz and it is crucial to reduce as much overhead as possible to ensure that the data processing is able to run at such a high rate. The primary purpose of the CommunicationModules project is to provide a higher level of organization by grouping classes that have similar behavior together which makes the overall code easier to understand and modify.

XPlaneSendPacket: The XPlaneSendPacket class inherits from the Object (.NET Framework) base class. This class represents the packet structure to send to X-Plane from the IOS via UDP. The design purpose for this class is to provide an easy to use data structure that will handle converting to and from a byte array in order to be sent to X-Plane. **Table 4.5.11** outlines the specifications for the XPlaneSendPacket class.

There are two member constant properties: HEADER and TAIL. The HEADER constant property is of type String and holds the value "DATA10" which is the header required by X-Plane. The TAIL constant property is of type String and represents the ending of the data packet structure.

There are two overloaded constructors available; the first requires no input parameters and instantiates a new object with default values of zero for the JoystickPostionX, JoystickPositionY, and ThrottlePosition public properties. The second overloaded constructor requires in a single input parameter: data, which is of type byte array. When this constructor is used, invokes the ParseByteArray member method using the data byte array as the input parameter.

There are three public properties: JoystickPositionX, JoystickPositionY, and ThrottlePosition. The JoystickPositionX property is of type Float and represents the x-position of the Joystick. The JoystickPostionY property is of type Float and represents the y-position of the Joystick. The ThrottlePostion property is of type Float and represents the throttle position in the aircraft.

There are two public methods: ToByteArray and FromByteArray. The ToByteArray public method requires no input parameters. It takes each public property and passes their values into the BitConverter.GetBytes method (.NET Framework), which returns a byte array representation of each public property. Next it declares a new byte array to be the size of the combined byte array lengths for the public properties. Then it copies each of the public property byte arrays into the aggregated byte array. Since the packet definition for communication from the IOS to X-Plane (please refer to **section 4.3.4**) specifies more values than needed, the rest of the values will be represented by an empty byte array. Finally the aggregated byte array representing the public properties is returned to the caller. The FromByteArray public method requires a single input parameter: data of type byte array. The purpose for this method is so that the caller using the XPlaneSendPacket object doesn't have to instantiate a new instance every time the data is received, therefore reducing overhead. This method works similar to the overloaded constructor that requires the data byte array input in the fact that it passes its input parameter byte array to the ParseByteArray method.

There is one member method, ParseByteArray, which requires a single input parameter: data, which is of type byte array. The method first checks to ensure the byte array is not equal to null, then checks if the byte array length is equal to 43. If either of these checks fails, the method returns and does not set the public properties. If the checks pass, the method will iterate through the byte array and parse it based on the definition in **Section 4.3.4** and set the public properties accordingly.

Class Name: XPlaneSendPacket	**Description:** Represents the data structure sent to X-Plane via UDP packets and provides methods for converting it to and from a byte array. Inherits from the .NET Framework Object class
Member Constant Properties	**Description**
HEADER	String value representing the head for the packet ("DATA10")
TAIL	String value representing the head for the packet ("1")
Instance Constructors	**Description**
XPlaneSendPacket	Requires no input parameters and instantiates a new empty XPlaneSendPacket object
XPlaneSendPacket	Requires a single input parameter: data, which is of type byte array. Calls the ParseByteArray method and sets the public properties accordingly.
Public Properties	**Description**
JoystickPositionX	Float value representing the x position of the joystick
JoystickPositionY	Float value representing the y position of the joystick
ThrottlePosition	Float value representing the throttle position in the aircraft
Public Methods	**Description**
ToByteArray	Requires no input parameters and returns a byte array representing the values of JoystickPostionX and JoystickPostionY
FromByteArray	Requires a single input parameter: data, which is of type byte array. Calls the ParseByteArray method and sets the public properties accordingly.
Member Methods	**Description**
ParseByteArray	Requires a single input parameter: data, which is of type byte array. Iterates through the data byte array and sets the JoystickPositionX and JoystickPositionY accordingly

Table 4.5.11 XPlaneSendPacket Specifications

XPlaneReceivePacket: The XPlaneReceivePacket class inherits from the Object (.NET Framework) base class. This class represents the packet structure received by the IOS from X-Plane via UDP. The design purpose for this class is to provide an easy to use data structure that will handle converting to and from a byte array in order to be received by the IOS. **Table 4.5.12** outlines the class specifications for the XPlaneReceivePacket.

There are two member constant parameters: HEADER and TAIL. The HEADER constant property is of type String and holds the value “MOTI01” which is the header required by X-Plane. The TAIL member constant property is of type String and holds the value “1” which represents the ending of the data packet structure. These values are stored as constants, because they are not going to be changed dynamically in the code. They are predefined values used defined by X-Plane.

There are two overloaded constructors available; the first requires no input parameters and instantiates a new object with default values of zero for the Pitch, Heading, Roll, SidewaysAcceleration, VerticalAcceleration, and LongitudinalAcceleration public properties. The second overloaded constructor requires in a single input parameter: data, which is of type byte array. When this constructor is used, invokes the ParseByteArray member method using the data byte array as the input parameter.

There are six public properties: Pitch, Heading, Roll, SidewaysAcceleration, VerticalAcceleration, and LongitudinalAcceleration. The Pitch property is of type Float and represents the pitch of the aircraft in degrees. The Heading property is of type Float and represents the heading of the aircraft in degrees. The SidewaysAcceleration property is of type Float and represents the sideways acceleration of the aircraft. The VerticalAcceleration property is of type Float and represents the vertical acceleration of the aircraft. The LongitudinalAcceleration property is of type Float and represents the longitudinal acceleration of the aircraft.

There are two public methods: ToByteArray and FromByteArray. The ToByteArray public method requires no input parameters. It takes each public property and passes their values into the BitConverter.GetBytes method (.NET Framework), which returns a byte array representation of each public property. Next it generates a byte array representation of the HEADER and TAIL constant values. Then it declares a new byte array to be the size of the combined byte array lengths for the public properties, header, and tail. Then it copies each of the public property byte arrays into the aggregated byte array in the order: header, public properties, and tail. Finally the aggregated byte array representing the data packet for data sent from X-Plane to the IOS as defined in **Section 4.3.4** is returned to the caller. The FromByteArray public method requires a single input parameter: data of type byte array. The purpose for this method is so that the caller using the XPlaneReceivePacket object doesn’t have to instantiate a new instance every time the data is received, therefore reducing overhead. This method works similar to the overloaded constructor that requires the data byte array input in the fact that it passes its input parameter byte array to the ParseByteArray method.

There is one member method, ParseByteArray, which requires a single input parameter: data, which is of type byte array. The method first checks to ensure the byte array is not equal to null, then checks if the byte array length is equal to 43. If either of these checks fails, the method returns and does not set the public properties. If the checks pass, the method will iterate through the byte array and parse it based on the definition in **Section 4.3.4** and set the public properties accordingly.

Class Name: XPlaneReceivePacket	**Description:** Represents the data structure received from X-Plane via UDP packets and provides methods for converting it to and from a byte array. Inherits from the .NET Framework Object class
Member Constant Properties	**Description**
HEADER	String value representing the head for the packet ("MOTI01")
TAIL	String value representing the head for the packet ("1")
Instance Constructors	**Description**
XPlaneReceivePacket	Requires no input parameters and instantiates a new empty XPlaneReceivePacket object
XPlaneReceivePacket	Requires a single input parameter: data, which is of type byte array. Calls the ParseByteArray method and sets the public properties accordingly.
Public Properties	**Description**
Pitch	Float value representing the pitch of the aircraft in degrees
Heading	Float value representing the heading of the aircraft in degrees
Roll	Float value representing the roll of the aircraft in degrees
SidewaysAcceleration	Float value representing the sideways acceleration of the aircraft in feet per minute
VerticalAcceleration	Float value representing the vertical acceleration of the aircraft in feet per minute
LongitudinalAcceleration	Float value representing the longitudinal acceleration of the aircraft in feet per minute
Public Methods	**Description**
ToByteArray	Requires no input parameters and returns a byte array representing the values of Pitch, Heading, Roll, SidewaysAcceleration, VerticalAcceleration, and LongitudinalAcceleration
FromByteArray	Requires a single input parameter: data, which is of type byte array. Calls the ParseByteArray method and sets the public properties accordingly.
Member Methods	**Description**
ParseByteArray	Requires a single input parameter: data, which is of type byte array. Iterates through the data byte

	array and sets the Pitch, Heading, Roll, SidewaysAcceleration, VerticalAcceleration, and LongitudinalAcceleration accordingly

Table 4.5.12 XPlaneReceivePacket Specifications

ControlLoaderSendPacket: The ControlLoaderSendPacket class inherits from the Object (.NET Framework) base class. This class represents the packet structure sent to the Control Loader from the IOS as defined in **Section 4.3.1**. The design purpose for this class is to provide an easy to use data structure that will handle converting to and from a byte array in order to be sent to the Control Loader. **Table 4.5.13** outlines the specifications for the ControlLoaderSendPacket.

There are two overloaded instance constructors available; the first requires no input parameters and instantiates a new ControlLoaderSendPacket object with values of zero for the Roll and Pitch public properties. The second overloaded constructor requires a single input parameter: data, which is of type byte array. When this constructor is used, it invokes the ParseByteArray member method using the data byte array as the input parameter.

There are two public properties: Roll and Pitch. The Roll property is of type Float and represents the Roll of the aircraft in degrees. The Pitch property is of type Float and represents the Pitch of the aircraft in degrees. These are the only two values needed by the Control Loader in order to drive the motion based.

There are two public methods: ToByteArray and FromByteArray. The ToByteArray public method requires no input parameters. It takes each public property and passes their values into the BitConverter.GetBytes method (.NET Framework), which returns a byte array representation of each public property. Next it declares a new byte array to be the size of the combined byte array lengths of the public properties. Then it copies each of the public property byte arrays into the aggregated byte array in the order Roll and Pitch. Finally the aggregated byte array representing the data packet for data sent to the Control Loader as defined in **Section 4.3.1** is returned to the caller. The FromByteArray public method requires a single input parameter: data, which if of type byte array. The purpose for this method is so that the caller using the ControlLoaderSendPacket object does not have to instantiate a new instance every time the data is set, therefore reducing overhead. This method works similar to the overloaded constructor that requires the data byte array input in the fact that it passes its input parameter byte array to the ParseByteArray member method.

There is one member method: ParseByteArray, which requires a single input parameter: data, which is of type byte array. The method first checks to ensure the byte array is not equal to null, then checks if the byte array length is equal to 8. If either of these checks fails, the method returns and does not set the public parameters. If the checks pass, the method will iterate through the byte array and parse it based on the definition in **Section 4.3.1** and set the public properties accordingly.

Class Name: ControlLoaderSendPacket	**Description:** Represents the data structure sent to the Control Loader via USB data packets and provides methods for converting it to and from a byte array. Inherits from the .NET Framework Object class
Instance Constructors	**Description**
ControlLoaderSendPacket	Requires no input parameters and instantiates a new empty ControlLoaderSendPacket object
ControlLoaderSendPacket	Requires a single input parameter: data, which is of type byte array. Calls the ParseByteArray method and sets the public properties accordingly.
Public Properties	**Description**
Roll	Float value representing the roll of the aircraft in degrees
Pitch	Float value representing the pitch of the aircraft in degrees
Public Methods	**Description**
ToByteArray	Requires no input parameters and returns a byte array representing the values of Roll and Pitch
FromByteArray	Requires a single input parameter: data, which is of type byte array. Calls the ParseByteArray method and sets the public properties accordingly.
Member Methods	**Description**
ParseByteArray	Requires a single input parameter: data, which is of type byte array. Iterates through the data byte array and sets the Roll and Pitch accordingly

Table 4.5.13 ControlLoaderSendPacket Specifications

ControlLoaderReceivePacket: The ControlLoaderReceivePacket class inherits from the Object (.NET Framework) base class. This class represents the packet structure sent from the Control Loader to the IOS as defined in **Section 4.3.2**. The design purpose for this class is to provide an easy to use data structure that will handle converting to and from a byte array in order to be received from the Control Loader. **Table 4.5.14** outlines the specifications for the ControlLoaderReceivePacket class.

There are two overloaded instance constructors available; the first requires no input parameters and instantiates a new ControlLoaderReceivePacket object with values of zero for all public properties. The second overloaded constructor requires a single input parameter: data, which is of type byte array. When this constructor is used, it invokes the ParseByteArray member method using the data byte array as the input parameter.

There are nine public properties: Heartbeat, MotionBaseStatus, JoystickStatus, ThrottleStatus, ThrottlePosition, JoystickPositionX, JoystickPositionY, MotionBaseServoPositionOne, and MotionBaseServoPostionTwo. The Heartbeat property is of type Integer and is incremented each time a packet is sent to the IOS by the Control Loader to notify that it is active. The MotionBaseStatus property is of type Boolean and is true if the Control Loader is able to communicate with the Motion Base, false otherwise. The JoystickStatus is of type Boolean and is true if the Control Loader is able to communicate with the Motion Base, false otherwise. The ThrottleStatus property is of type Boolean and is true if the Control Loader is able to communicate with the Throttle. The ThrottlePosition property is of type Float and represents the position of the Throttle. The JoystickPositionX property is of type Float and represents the x-position of the Joystick. The JoystickPositionY property is of type Float and represents the y-position of the Joystick. The MotionBaseServoPositionOne property is of type Float and represents the position of servo labeled "Servo_1" on the Motion Base. The MotionBaseServoPostionTwo property is of type Float and represents the position of the servo labeled "Servo_2" on the Motion Base.

The ToByteArray public method requires no input parameters. It takes each public property and passes their values into the BitConverter.GetBytes method (.NET Framework), which returns a byte array representation of each public property. Next it declares a new byte array to be the size of the combined byte array lengths of the public properties. Then it copies each of the public property byte arrays into the aggregated byte array in the order: Heartbeat, MotionBaseStatus, JoystickStatus, ThrottleStatus, ThrottlePosition, JoystickPositionX, JoystickPositionY, MotionBaseServoPositionOne, and MotionBaseServoPostionTwo. Finally the aggregated byte array representing the data packet for data sent from the Control Loader to the IOS as defined in **Section 4.3.2** is returned to the caller. The FromByteArray public method requires a single input parameter: data, which is of type byte array. The purpose for this method is so that the caller using the ControlLoaderReceivePacket object does not have to instantiate a new instance every time the data is set, therefore reducing overhead. This method works similar to the overloaded constructor that requires the data byte array input in the fact that it passes its input parameter byte array to the ParseByteArray member method.

There is one member method: ParseByteArray, which requires a single input parameter: data, which is of type byte array. The method first checks to ensure the byte array is not equal to null, then checks if the byte array length is equal to 36. If either checks fail, the method returns and does not set the public properties. If the checks pass, the method will iterate through the byte array and parse it based on the definition in **Section 4.3.1** and set the public properties. Note that although the MotionBaseStatus, JoystickStatus, and ThrottleStatus public properties are of type Boolean, they are represented as an Integer value when being converted to a byte array, with true represented by a numerical 1 and false represented by a numerical 0.

Class Name: ControlLoaderReceivePacket	**Description:** Represents the data structure from the Control Loader via USB data packets and provides methods for converting it to and from a byte array. Inherits from the .NET Framework Object class
Instance Constructors	**Description**
ControlLoaderReceivePacket	Requires no input parameters and instantiates a new empty ControlLoaderReceivePacket object
ControlLoaderReceivePacket	Requires a single input parameter: data, which is of type byte array. Calls the ParseByteArray method and sets the public parameters accordingly.
Public Properties	**Description**
Heartbeat	Integer value that is incremented by the Control Loader to notify that it is running
MotionBaseStatus	Boolean value that represents if the Control Loader is able to communicate with the Motion Base (true if able to communication, false otherwise)
JoystickStatus	Boolean value that represents if the Control Loader is able to communicate with the Joystick (true if able to communication, false otherwise)
ThrottleStatus	Boolean value that represents if the Control Loader is able to communicate with the Throttle (true if able to communication, false otherwise)
ThrottlePosition	Float value representing the position of the Throttle
JoystickPositionX	Float value representing the x position of the Joystick
JoystickPositionY	Float value representing the y position of the Joystick
MotionBaseServoPositionOne	Float value representing the position of servo one for the Motion Base
MotionBaseServoPositionTwo	Float value representing the position of server two for the Motion Base
Public Methods	**Description**
ToByteArray	Requires no input parameters and returns a byte array representing the values of Heartbeat, MotionBaseStatus, JoystickStatus, ThrottleStatus, ThrottlePosition, JoystickPositionX, JoystickPositionY, MotionBaseServoOne, and MotionBaseServoTwo
FromByteArray	Requires a single input parameter: data, which is of type byte array. Calls the ParseByteArray method and sets the public properties

	accoordingly.
Member Methods	**Description**
ParseByteArray	Requires a single input parameter: data, which is of type byte array. Iterates through the data byte array and sets the Roll and Pitch accordingly

Table 4.5.14 ControlLoaderReceivePacket Specifications

ICommunicationModuleBase: The ICommunicationModuleBase class is an interface in which all communication modules must implement. This guarantees that every communication modules implements the necessary methods invoked by the SimulationModuleManager class of the Simulator project. The design purpose for using an interface is so that in the SimulationModuleManager, it is possible to iterate through each communication module using them as ICommunicationModuleBase type, which reduces code complexity. **Table 4.5.15** outlines the specifications for the ICommunicationModuleBase class.

There are three public static methods that each communication module must implement: Initialize, GetData, and SendData. The Initialize method requires no parameter inputs and is used to establish the communication connection with the desired device specific to the class in which it is implemented. The GetData method requires no input parameters and returns a byte array containing the data from the desired device. The SendData method requires an input data of type byte array, which is sent to the desired device.

Class Name: ICommunicationModuleBase	**Description:** Interface that communication modules inherit from
Public Methods	**Description**
Initialize	Initializes the communication
GetData	Returns data from the device the module is communicating with
SendData	Sends data passed into the device the module is communication with

Table 4.5.15 ICommunicationModuleBase Specifications

ControlLoaderCommunication: The ControlLoaderCommunication class implements the ICommunicationModuleBase class and encapsulates the functionality of the USBConnection class located in the Utilities project for a specific rather than general purpose in order to communicate with the Control Loader. The design purpose of this class is to provide an easy to use interface to send/receive data from the Control Loader in order to reduce code complexity. This class is declared as static and doesn’t contain any constructors. **Table 4.5.16** outlines the specifications for the ControlLoaderCommunication class.

There are two constant static member properties: VENDOR_ID and PRODUCT_ID. Both of these are of type Integer and represent the vendor identification and product identification numbers of the Control Loader.

There is one static member property: _usbConnection. The _usbConnection is of type USBConnection and provides the interface in order to communication with the Control Loader.

There is one public static read-only property: Active, which returns the value of the _usbConnection.Active property that represents if the connection is active (true if active, false otherwise). This is implemented as a getter to ensure it is read-only.

Since the ControlLoaderCommunication class implements the ICommunicationModuleBase interface, it contains three public static methods: Initialize, GetData, and SendData. The Initialize static method takes in no input parameters and is responsible for initializing the USB connection with the Control Loader. This method will invoke the InitConnection method of the _usbConnection object which requires two input parameters: vendorId and productId, which will be the values defined by the VENDOR_ID and PRODUCT_ID constant values. The GetData static method requires no inputs and returns a value of type Object (defined by the ICommunicationModuleBase) which is the base class of the ControlLoaderReceivePacket. The method invokes the _usbConnection GetData method which returns a byte array containing the data. The structure of this byte array is outlined in **Section 4.3.2**. When the byte array is received, the GetData method will then instantiate a new ControlLoaderReceivePacket by calling the overloaded constructor that takes in a byte array parameter. Finally the method casts the ControlLoaderReceivePacket to its base class type (Object) and returns it to the caller. The SendData method requires a single input parameter: data, which is of type Object, the base class of the ControlLoaderSendPacket. This method first attempts to cast the data parameter to a ControlLoaderSendPacket object, if this cast fails, the method will return without sending any data. The reason for no error notification is because the assumption is that the data is corrupt and specific to that one instance and an attempt to continue the active simulation session will be made. Next the method will invoke the ControlLoaderSendPacket ToByteArray method which returns the byte array representation of the ControlLoaderSendPacket data structure. It then invokes the SendData method of the _usbConnection static member property which sends the byte array to the Control Loader and after this the SendData method has completed.

Class Name: ControlLoaderCommunicationModule	**Description:** Implements the ICommunicationModuleBase interface and controls communication via USB with the Control Loader
Member Constant Properties	**Description**
VENDOR_ID	The vendor identification used for the Control Loader
PRODUCT_ID	The product identification used for the Control Loader
Member Static Properties	**Description**
_usbConnection	Provides the interface in order to communicate with the Control Loader
Public Static Read-Only Properties	**Description**

Active	Returns the _usbConnection.Active property which represents if the connection is active (true if active, false otherwise)
Public Static Methods	**Description**
Initialize	Initializes the USB communication to the Control Loader
GetData	-Returns data from the device the module is communicating with
SendData	-Sends data passed into the device the module is communication with

Table 4.5.16 ControlLoaderCommunicationModule Specifications

XPlaneCommunication: The XPlaneCommunication class implements the ICommunicationModuleBase class and encapsulates the functionality of the UDPConnection class located in the Utilities project for a specific rather than general purpose in order to communicate with X-Plane. The design purpose of this class is to provide an easy to use interface to send/receive data from X-Plane in order to decrease code complexity. This class is declared as static and does not contain any constructors. **Table 4.5.17** outlines the specifications for the XPlaneCommunication class.

There are three constant static member properties: ADDRESS, SEND_PORT, and RECEIVE_PORT. The ADDRESS constant static property is of type String and represents the IP address of X-Plane, which will be defined as the local loop back ("127.0.0.1") since the IOS and X-Plane are hosted on the Flight Computer. The SEND_PORT constant static property is of type Integer and represents the port X-Plane uses to receive data, which is defined by X-Plane as 4900. The RECEIVE_PORT constant static property is of type Integer and represents the port X-Plane uses to send data, which is defined by X-Plane as 4901.

There is one static member property: _udpConnection. The _udpConnection is of type UDPConnection and provides the interface for communications with X-Plane.

There is one public static read-only property: Active, which returns the value of the _udpConnection.Active property that represents if the connection is active (true if active, false otherwise). This is implemented as a getter to ensure it is read-only.

Since the XPlaneCommunication class implements the ICommunicationModuleBase interface, it contains three public static methods: Initialize, GetData, and SendData. The Initialize static method requires no input parameters and is responsible for initializing the UDP connection with X-Plane. This method will invoke the InitConnection method of the _udpConnection object which requires three input parameters: address, sendPort, and receivePort, which will be the values defined by the ADDRESS, SEND_PORT, and RECEIVE_PORT constant values. The GetData static method requires no inputs and returns a value of type Object (defined by the ICommunicationModuleBase) which is the base class of the XPlaneReceivePacket. The method invokes the _udpConnection GetData method which returns a byte array containing the received data. The structure of this byte array is outlined in **Section 4.3.3**. When the byte array is received, the GetData method will then instantiate a new XPlaneReceivePacket by calling the overloaded constructor that takes in a byte array input parameter. Finally the method casts the XPlaneReceivePacket to its base class type (Object) and returns it to the caller. The SendData static method requires a single input parameter: data, which is of type Object, the base class of the XPlaneSendPacket. This method first attempts to cast the data parameter to a XPlaneSendPacket object, if this cast fails, the method will return without sending any data. The reason for no error notification is because the assumption is that the data is corrupt and specific to that one instance and an attempt to continue the active simulation will be made. Next the method will invoke the XPlaneSendPacket ToByteArray method which returns the byte array representation of the XPlaneSendPacket data structure. It then invokes the SendData method of the _udpConnection static property which sends the byte array to X-Plane and after this the SendData method has completed.

Class Name: XPlaneCommunication	**Description:** Implements the ICommunicationModuleBase interface and controls communication via UDP with X-Plane
Member Constant Properties	**Description**
ADDRESS	String value containing the IP address to use ("127.0.0.1")
SEND_PORT	Integer value representing the port X-Plane receives on (1400)
RECEIVE_PORT	Integer value representing the port X-Plane receives on (1401)
Member Static Properties	**Description**
_udpConnection	Provides the interface in order to communicate with X-Plane
Public Static Read-Only Properties	**Description**
Active	Returns the _udpConnection.Active property which represents if the connection is active (true if active, false otherwise)
Public Static Methods	**Description**
Initialize	Initializes the UDP communication with X-Plane
GetData	-Returns data from the device the module is communicating with
SendData	-Sends data passed into the device the module is communication with

Table 4.5.17 XPlaneCommunication

Simulator Project

The Simulator project contains classes' specific to processing data required for the simulator. They represent the back end of the IOS software and are implemented in a relatively non object oriented fashion. The design purpose behind creating this project was to provide a higher level of organization to the VS2010 solution. Grouping classes that deal specifically with simulation data processing allows for the intention of the code to be better understood.

SimulationModuleManager: The SimulationModuleManager class is responsible for processing all simulation data. It in effect is the "glue" that pieces the concept of the back end in regards to the IOS software together. This class is declared as static contains no constructors. **Table 4.5.18** outlines the specifications of the SimulationModuleManager class.

There are five static member properties: _simulationActive, _updateTimer, _updateInterval, _updateRate, _datapool. The _simulationActive property is of type Boolean and represents whether the simulation is active or not (true if active, false otherwise). The _updateTimer property is of type Timer (.NET Framework, System.Timers.Timer). This property allows the back end to specify an interval at which it will raise its Tick event, which is handled by the UpdateModules member method and allows the SimulationModuleManager to have more control over rate which for simulators of this type is between 50Hz and 100Hz. The _updateInterval property is of type Integer and represents the time in milliseconds between each Tick event fired by the _updateTimer property. The _updateRate property is of type Integer and represents the rate at which the _updateTimer will run in Hertz. The _datapool property is of type Datapool and allows the GetData and SetData member methods to utilize reflection to access variables by using their String representation.

There is a single read-only public static property: SimulationActive, which is of type Boolean. The SimulationActive property is implemented using a getter to ensure it is read-only and returns the value of the _simulationActive member property.

There is one public static property: UpdateRate, which is of type Integer and represents the rate at which the _updateTimer runs in Hertz. It is implemented using a getter and setter. When a new value is set for the UpdateRate property, first a check will be made against the _updateRate to see if the values, match, it they do, then the rate is the same, if not, then the new rate is set to _updateRate. After this, then the _updateRate is converted to the time interval in which an update is to occur and set to the _updateInterval property. Then the _updateTimer is restarted with its Interval property set to the _updateInterval property.

There are seven public static methods: Initialize, StartSimulation, EndSimulation, GetInt, SetInt, GetFloat, and SetFloat. The Initialize method invokes the Initialize methods for the ControlLoaderCommunication and XPlaneCommunication modules. The StartSimulation method sets the _simulationActive property to true which allows the UpdateModules method to send the Roll and Pitch from X-Plane to the Control Loader. The EndSimulation method sets the _simulationActive property to false which allows the UpdateModules method to send a value of 0 for Roll and Pitch to the Control Loader. The GetInt method requires a single input parameter: variable, which of type String. This method invokes the GetData method and casts the returned Object to type Integer before returning it to the caller. The SetInt method requires two input parameters: variable of type String and data which is of type Integer. This method invokes the SetData member method by passing in the variable and data input parameters and then returns the Boolean value. GetFloat method requires a single input parameter: variable, which of type String. This method invokes the GetData method and casts the returned Object to type Float before returning it to the caller. The SetFloat method requires two input parameters:

variable of type String and data which is of type Float. This method invokes the SetData member method by passing in the variable and data input parameters and then returns the Boolean value. The purpose of the GetInt, SetInt, GetFloat, and SetFloat member methods is to provide a type safe interface for getting and setting variables in the IOS. It also acts adds an additional layer between the variables in memory and the pieces accessing them, which allows gives the option to be able to easily change the structure for how memory is stored without having to re-code the entire IOS software.

There are three static member methods: UpdateModules, GetData, and SetData. The UpdateModules method is invoked each time the _updateTimer raises its Tick event at an interval defined by the _updateInterval property which is set by the UpdateRate public property. This method is responsible for effectively driving the simulation. It first invokes the ControlLoaderCommunication classes GetData method which has a return value of type Object and casts it to a ControlLoaderReceivePacket. It then sets the correlating variables in memory as defined by the Datapool class for the ControlLoaderReceivePacket public properties. Then it creates a new XPlaneSendPacket object using the JoystickPositionX, JoystickPositionY, and ThrottlePosition from the ControlLoaderReceivePacket to set the correlating public properties. Next the XPlaneCommunication classes SendData method is invoked passing in the newly created XPlaneSendPacket as the input parameter. Then the XPlaneCommunication classes GetData method is invoked which has a return value of type Object and casts it to an XPlaneReceivePacket type. It then sets the correlating variables in memory. After which the method creates a new ControlLoaderSendPacket and sets the Roll and Pitch public properties to their correlating values in memory. If the _simulationActive property is set as false, then the ControlLoaderSendPacket will have its Roll and Pitch properties set to a value of 0, so that the Motion Base is guaranteed to be in a neutral position. Finally it invokes the ControlLoaderCommunication classes SendData method passing in the newly created ControlLoaderSendPacket as the input parameter. The order this method gets and sends data was not chosen arbitrarily. The flow of data starts with the Control Loader, goes to the IOS, the goes to X-Plane at which point X-Plane uses the Joystick positional values and Throttle value to update the flight model, then the IOS receives the Roll and Pitch values from the X-Plane, and then finally those values are sent to the Control Loader. This flow ensures that the Motion Base and X-Plane are closely synchronized. The GetData method requires a single input parameter: variable, which is of type String. First the method checks that variable is a valid String, next it uses reflection with the _datapool variable to get the PropertyInfo object for the variable String, then it invokes the PropertyInfo object’s GetValue method which is returned to the caller of the GetData method. The SetData method requires two input parameters: variable of type String and data of type Object. First the method checks that variable and data are both valid, next it uses reflection with the _datapool member property to get the PropertyInfo object for the variable String, then it invokes the PropertyInfo object’s SetValue method passing in data as the parameter which sets data as the value for the parameter represented by the variable input parameter.

Class Name: SimulationModuleManager	**Description:** Controls the update rate for the ControlLoaderCommunication and XPlaneCommunication. Processes data from each module and updates the health status of the Joystick, Motion Base, Throttle, Rudder Pedals, and X-Plane.
Member Static Properties	**Description**
_simulationActive	Boolean value representing if the simulation session is active or not(true if active, false otherwise)
_updateTimer	Timer object which will run at a rate defined by the UpdateRate property.
_updateInterval	Integer value representing the interval in which the _updateTimer will raise its Tick event
_updateRate	Integer value representing the rate at which the _updateTimer will run
_datapool	Datapool object that contains all the variables specific to the simulation
Read-Only Public Static Properties	**Description**
SimulationActive	Boolean value representing if the simulation session is active or not(true if active, false otherwise)
Public Static Properties	**Description**
UpdateRate	Integer value representing the update rate in hertz for the ControlLoaderCommunication and XPlaneCommunication
Public Static Methods	**Description**
Initialize	Initializes the ControlLoaderCommunication and XPlaneCommunication. Starts the _updateTimer Timer to run at a rate defined by the UpdateRate property (default is 50 hertz)
StartSimulationFor	Sets the _simulationActive member property to true.
EndSimulation	Sets the _simulationActive member property to false
GetInt	Requires a String parameter: variable, which is the String representation of the variable. This is a wrapper for the GetData member method and converts the returned value from GetData to type Integer
GetFloat	Requires a String parameter: variable, which is the String representation of the variable. This is a wrapper for the GetData member method and converts the returned value from GetData to type Float

SetInt	Requires two input parameters: variable of type String and data of type Integer. This is a wrapper for the SetData member method.
SetFloat	Requires two input parameters: variable of type String and data of type Integer. This is a wrapper for the SetData member method.
Member Static Methods	**Description**
UpdateModules	Updates the modules at a rate defined by the UpdateRate property.
GetData	Requires an input parameter of type String named variable. First the method checks if variable is valid, next it uses reflection on the _datapool object to get the property represented by the variable String, finally it returns the value of the property as type object.
SetData	Requires two input parameters: variable of type String and data of type Integer. First the method checks if both variable and data are valid, next it uses reflection on the _datapool object to get the property represented by the variable String, finally it attempts to set the value to the data input parameter. A value of true is returned if the value is set and false otherwise.

4.5.18 SimulationModuleManager Specifications

Datapool: The Datapool class contains all the variables needed for the project. It allows the variables to be accessed utilizing reflection and will store all the variables in a memory mapped file. The design purpose for this class is to provide a single location where all variables can be accessed. There were two ways to do this, either by implementing all the variables within a struct then using pointers to access the variables or to create a memory mapped file with random access capabilities. Utilizing a struct would force any methods that need to access the variables via a pointer to run as unmanaged code and could lead to a producer-consumer problem. The second approach of using a memory mapped file allows all the variables to be accessed through managed code and as an added benefit, it also gives the ability to share the memory mapped file between separate processes. Although all functionality is implemented within the IOS, the ability to use a memory mapped file for multiple processes could be used if during development the IOS has to be broken into separate applications (front end and back end) because both applications would have access to the same variables in memory. Note that all the public properties are mapped to memory using pre-defined memory offsets to designate the location. **Table 4.5.19** outlines the specifications of the Datapool class.

There is one instance constructor, which requires no input parameters and instantiates a new instance of the Datapool class. When invoked, it will instantiate the _memoryMapFile member property then invokes its CreateViewAccessor method in which the return value is set to the _accessor member property.

There are two member properties: _memoryMapFile and _accessor. The _memoryMapFile property is of type MemoryMappedFile (.NET Framework) and allows for the reading/writing directly into memory and provides the interface for mapping variable values to memory. The _accessor property is of type MemoryMappedViewAccessor and provides the random access view into the _memoryMapFile.

There are fifty public properties: JoystickPotPositionX, JoystickPotPositionY, Roll, Pitch, CommandedRoll, CommandedPitch, ThrottlePotPosition, MotionBaseServo1PotPosition, MotionBaseServo2PotPosition, Heading , MagneticHeading, SidewaysAcceleration, VerticalAcceleration, LongitudinalAcceleration, ControlLoaderHeartbeat, ThermalTops, ThermalCoverage, ThermalClimbRate, ClosestAirportTemperature, SeaLevelBarometricPressure, Visibility, TopMsl, BaseMsl, AboveGroundLevel, HighAltitudeWindLayer, HighAltitudeWindDirection, HighAltitudeWindSpeed, HighAltitudeWIndSheer, HighAltitudeWindSheerDirection, HighAltitudeWindTurbulence, MidAltitudeWindLayer, MidAltitudeWindDirection, MidAltitudeWindSpeed, MidAltitudeWindSheer, MidAltitudeWindSheerDirection, MidAltitudeWindTurbulence, LowAltitudeWindLayer, LowAltitudeWindDirection, LowAltitudeWindSpeed, LowAltitudeWindSheer, LowAltitudeWindSheerDirection, LowAltitudeWindTurbulence, SimulationActive, MotionBaseStatus, JoystickStatus, ThrottleStatus, Precipitation, CoverType, RunwayCondition, and RunwayName. JoystickPotPositionX property is of type Float and represents the x position of the joystick. JoystickPotPositionY property is of type Float and represents the y position of the joystick. Roll property is of type Float and represents the roll of the aircraft in degrees. Pitch property is of type Float and represents the pitch of the aircraft in degrees. CommandedRoll property is of type Float and represents the commanded roll of the aircraft. CommandedPitch property is of type Float and represents the commanded pitch of the aircraft. ThrottlePotPosition property is of type Float and represents the position of the Throttle potentiometer. MotionBaseServo1PotPosition property is of type Float and represents the position of the Motion Base Servo 1 potentiometer. MotionBaseServo2PotPosition property is of type Float and represents the position of the Motion Base Servo 2 potentiometer. Heading property is of type Float and represents the heading of the aircraft in degrees. MagneticHeading property is of type Float and represents the magnetic heading of the aircraft in degrees. SidewaysAcceleration property is of type Float and represents the sideways acceleration of the aircraft in feet per minute.

VerticalAcceleration property is of type Float and represents the vertical acceleration of the aircraft in feet per minute. LongitudinalAcceleration property is of type Float and represents the longitudinal acceleration of the aircraft in feet per minute. ControlLoaderHeartbeat property is of type Integer value that is incremented by the Control Loader to notify that it is running. ThermalTops property is of type Integer and represents the thermal tops in feet above sea level. ThermalCoverage property is of type Integer and represents the thermal coverage as a percentage. ThermalClimbRate property is of type Integer and represents the thermal climb rate in feet per minute. ClosestAirportTemperature property is of type Integer and represents the temperature at the closet airport to the aircraft in Fahrenheit. SeaLevelBarometricPressure property is of type Integer and represents the barometric pressure at sea level in inches of mercury. Visibility property is of type Integer and represents the visibility as viewed by the pilot. TopMsl property is of type Integer and represents the miles above sea level of the cloud tops. BaseMsl property is of type Integer and represents the miles above sea level of the cloud base. AboveGroundLevel property is of type Integer and represents the miles above ground level of the clouds. HighAltitudeWindLayer property is of type Integer and represents the feet above sea level for High altitude winds. HighAltitudeWindDirection property is of type Integer and represents the direction of the HighAltitudeWindLayer in degrees. HighAltitudeWindSpeed property is of type Integer and represents the velocity of the HighAltitudeWindLayer in knots. HighAltitudeWIndSheer property is of type Integer and represents the sheer of the HighAltitudeWindLayer in knots. HighAltitudeWindSheerDirection property is of type Integer and represents the direction of the HighAltitudeWindSheer in degrees. HighAltitudeWindTurbulence property is of type Integer and represents the intensity of turbulence for the HighAltitudeWindLayer. MidAltitudeWindLayer property is of type Integer and represents the feet above sea level for mid altitude winds. MidAltitudeWindDirection property is of type Integer and represents the direction of the MidAltitudeWindLayer in degrees. MidAltitudeWindSpeed property is of type Integer and represents the velocity of the MidAltitudeWindLayer in knots. MidAltitudeWindSheer property is of type Integer and represents the sheer of the MidAltitudeWindLayer in knots. MidAltitudeWindSheerDirection property is of type Integer and represents the direction of the MidAltitudeWindSheer in degrees. MidAltitudeWindTurbulence property is of type Integer and represents the intensity of turbulence for the MidAltitudeWindLayer. LowAltitudeWindLayer property is of type Integer and represents the feet above sea level for Low altitude winds. LowAltitudeWindDirection property is of type Integer and represents the direction of the LowAltitudeWindLayer in degrees. LowAltitudeWindSpeed property is of type Integer and represents the velocity of the LowAltitudeWindLayer in knots. LowAltitudeWindSheer property is of type Integer and represents the sheer of the LowAltitudeWindLayer in knots. LowAltitudeWindSheerDirection property is of type Integer and represents the direction of the LowAltitudeWindSheer in degrees. LowAltitudeWindTurbulence property is of type Integer and represents the intensity of turbulence for the LowAltitudeWindLayer.

SimulationActive property is of type Boolean and represents if in an active simulation state true if active, false otherwise). MotionBaseStatus property is of type Boolean and represents if the Motion Base is communicating with the Control Loader. JoystickStatus property is of type Boolean and represents if the Joystick is communicating with the Control Loader. ThrottleStatus property is of type Boolean and represents if the Throttle is communicating with the Control Loader. Precipitation property is of type PrecipitationTypes/String. CoverType property is of type CloudCoverTypes/String. RunwayCondition property is of type RunwayConditionTypes/String. RunwayName property is of type String. Each of the public properties is implemented using getter and setters. When the setter is invoked, it will invoke the WriteBytes method passing in the appropriate values for the input parameters depending upon its pre-defined location in memory. When the getter is invoked, it will invoke the ReadBytes method passing in the appropriate values for the input parameters depending upon its pre-defined location in memory.

There are two member methods: ReadBytes and WriteBytes. The ReadBytes method requires two input parameters: offset and length. The offset is of type Unsigned Integer and represents the offset into memory to start reading from. The length is of type Unsigned Integer and represents the number of bytes to read beginning at the position defined by the offset. The method first declares a new byte array named returnData which will hold the bytes read from memory. Then it will create a local variable named position that will be set to the value of offset. Next it will iterate from 0 to the length and invoked the ReadByte method of the _accessory member property passing in the position variable as the input parameter. At the end of each iteration the position variable is incremented by the size of a byte (2) so that when the loop starts again it will read the next byte from memory. Each iteration of the loop stores the read bytes into the returnData byte array, and then when complete, the returnData byte array is returned to the caller.

Class Name: Datapool	**Description:** Contains all the variables used for the simulation.
Instance Constructors	**Description**
Datapool	Instantiates a new instance of the Datapool class. This also initializes the _memoryMapFile member parameters, then invokes it's CreateViewAccessor method in which the return value is set to the _accessor member parameter
Member Properties	**Description**
_memoryMapFile	MemoryMappedFile (.NET Framework) object allows for the reading and writing directly into memory and provides the interface for mapping parameter values to memory. This provides random access which is preferred.
_accessor	MemoryMappedViewAccessor (.NET Framework) object provides the random access view into the _memoryMapFile
Public Properties	**Description**
JoystickPotPositionX	Float value representing the x position of the joystick
JoystickPotPositionY	Float value representing the y position of the joystick
Roll	Float value representing the roll of the aircraft in degrees
Pitch	Float value representing the pitch of the aircraft in degrees
CommandedRoll	Float value representing the commanded roll of the aircraft
CommandedPitch	Float value representing the commanded pitch of the aircraft
ThrottlePotPosition	Float value representing the position of the Throttle

	potentiometer
MotionBaseServo1PotPosition	Float value representing the position of the Motion Base Servo 1 potentiometer
MotionBaseServo2PotPosition	Float value representing the position of the Motion Base Servo 2 potentiometer
Heading	Float value representing the heading of the aircraft in degrees
MagneticHeading	Float value representing the magnetic heading of the aircraft in degrees
SidewaysAcceleration	Float value representing the sideways acceleration of the aircraft in feet per minute
VerticalAcceleration	Float value representing the vertical acceleration of the aircraft in feet per minute
LongitudinalAcceleration	Float value representing the longitudinal acceleration of the aircraft in feet per minute
ControlLoaderHeartbeat	Integer value that is incremented by the Control Loader to notify that it is running
ThermalTops	Integer value representing the thermal tops in feet above sea level
ThermalCoverage	Integer value representing the thermal coverage as a percentage
ThermalClimbRate	Integer value representing the thermal climb rate in feet per minute
ClosestAirportTemperature	Integer value representing the temperature at the closet airport to the aircraft in Fahrenheit
SeaLevelBarometricPressure	Integer value representing the barometric pressure at sea level in inches of mercury
Visibility	Integer value representing the visibility as viewed by the pilot
TopMsl	Integer value representing the miles above sea level of the cloud tops
BaseMsl	Integer value representing the miles above sea level of the cloud base
AboveGroundLevel	Integer value representing the miles above ground level of the clouds
HighAltitudeWindLayer	Integer value representing the feet above sea level for High altitude winds
HighAltitudeWindDirection	Integer value representing the direction of the HighAltitudeWindLayer in degrees
HighAltitudeWindSpeed	Integer value representing the velocity of the HighAltitudeWindLayer in knots
HighAltitudeWIndSheer	Integer value representing the sheer of the HighAltitudeWindLayer in knots
HighAltitudeWindSheerDirection	Integer value representing the direction of the HighAltitudeWindSheer in degrees
HighAltitudeWindTurbulence	Integer value representing the intensity of turbulence

	for the HighAltitudeWindLayer
MidAltitudeWindLayer	Integer value representing the feet above sea level for Mid altitude winds
MidAltitudeWindDirection	Integer value representing the direction of the MidAltitudeWindLayer in degrees
MidAltitudeWindSpeed	Integer value representing the velocity of the MidAltitudeWindLayer in knots
MidAltitudeWindSheer	Integer value representing the sheer of the MidAltitudeWindLayer in knots
MidAltitudeWindSheerDirection	Integer value representing the direction of the MidAltitudeWindSheer in degrees
MidAltitudeWindTurbulence	Integer value representing the intensity of turbulence for the MidAltitudeWindLayer
LowAltitudeWindLayer	Integer value representing the feet above sea level for Low altitude winds
LowAltitudeWindDirection	Integer value representing the direction of the LowAltitudeWindLayer in degrees
LowAltitudeWindSpeed	Integer value representing the velocity of the LowAltitudeWindLayer in knots
LowAltitudeWindSheer	Integer value representing the sheer of the LowAltitudeWindLayer in knots
LowAltitudeWindSheerDirection	Integer value representing the direction of the LowAltitudeWindSheer in degrees
LowAltitudeWindTurbulence	Integer value representing the intensity of turbulence for the LowAltitudeWindLayer
SimulationActive	Boolean value representing if in an active simulation state. (true if active, false otherwise)
MotionBaseStatus	Boolean value representing if the Motion Base is communicating with the Control Loader
JoystickStatus	Boolean value representing if the Joystick is communicating with the Control Loader
ThrottleStatus	Boolean value representing if the Throttle is communicating with the Control Loader
Precipitation	PrecipitationTypes/String
CoverType	CloudCoverTypes/String
RunwayCondition	RunwayConditionTypes/String
Member Methods	**Description**
ReadBytes	Requires two input parameters: offset and length. The offset is of type uint and represents the offset into memory to start reading from. The length is of type uint and represents the number of bytes to read beginning at the offset.
WriteBytes	Requires three input parameters: offset, data, length. The offset is of type uint and represents the offset into memory where writing will begin. The data is of type byte array and contains the byte representation

	of the data to be written to memory. The length is of type uint and represents the length of bytes to be written to memory

Table 4.5.19 Datapool Specifications

UserInterfaceControls Project

The UserInterfaceControls project contains class's specific to the front end user interface portion of the IOS software. Each class represents a user control that can be re-used, which leads to faster development of the user interface and cleaner less complex code. The design purpose for this is that if a change needs to be made the way a particular user control works on the user interface, the change can be made in one spot rather than having to make the change in multiple places which could make it very difficult to maintain the code. This also allows for the VS2010 solution to be better organized visually which will make it easier for other group members to understand and make changes easily if necessary. Within this project there are two directories: Display and Input. The Display directory contains user controls that have the specific purpose of displaying information on the user interface. The Input directory contains user controls that have the specific purpose of allowing the instructor to enter data into the user interface.

IParameter: The IParameter class is an interface and is implemented by any user controls that will either display a parameter or allow a value to be entered for that parameter. The purpose to have all user controls of this functionality implement this interface is so that on the user interface, it will be easier to iterate through a controls collection by only looking for type IParameter. This leads to less complex and more efficient code. **Table 4.5.20** outlines the specifications for the IParameter class.

There are three public properties: DisplayName, VariableType, and VariableName. The DisplayName property is of type String and is the user friendly name that will be displayed to the instructor. The VariableType property is of type String and represents the type of the VariableName (either an Integer or Float). The VariableName property is of type String and represents the String representation of the variable in memory that will be used by the DisplayValue property.

Class Name: IParameter	**Description:** Interface that all Parameter controls must derive from
Public Properties	**Description**
DisplayName	String representing the name of the parameter to be displayed to the instructor
VariableType	String representing the type for the variable defined by VariableName
VariableName	String representing the variable name of the parameter

Table 4.5.20 IParameter Specifications

InputParameter: The InputParameter class implements the IParameter interface and allows the instructor to input parameter values. The purpose for this was to provide a re-usable control for building the user interface that is responsible for allowing the instructor to enter a value, validating the input, and setting it in memory. **Table 4.5.21** outlines the specifications for the InputParameter class.

There is one instance constructor that requires no inputs and instantiates a new instance of the InputParameter user control. When invoked, the constructor will initialize all of the member properties and display elements that were auto-generated by VS2010.

There are two member properties: _displayNameLabel and _displayValueTextField. The _displayNameLabel is of type Label (.NET Framework), which will display the value of the DisplayName public property. The _displayValueTextBox is of type TextBox (.NET Framework) and allows the instructor to enter a value.

There are five public properties: DisplayName, VariableName, VariableType, MaximumValue, and MinimumValue. The DisplayName property is of type String and represented the display String for the _displayNameLabel. The VariableName property is of type String and is the String representation of the variable to set in memory. The VariableType is of type String and represents the type for the VariableName String for use in the Validate member method. The MaximumRange is of Integer and represents the maximum value the instructor can input. The MinimumRange is of type Integer and represents the minimum value the instructor can input. The public properties are set during development time when adding the control to a Window Form (.NET Framework).

There is one member method: Validate, which is the event handler for the _displayValueTextField.TextChanged event. This method will check the instructor inputted value against the MaximumValue and MinimumValue public properties, if the input value is within the range, then the value is accepted. Next the method will check the ValueType public property and will invoke either the SetFloat or SetInt method from the SimulationModuleManager. This allows each control to be responsible for setting the value in memory which reduces the code involved with gathering data from the user interface in order to start a new simulation session.

Class Name: InputParameter	**Description:** Implements the IParameter interface and provides a user control for the instructor to input data.
Instance Constructors	**Description**
InputParameter	Requires no input parameters and instantiates a new instance of the InputParameter class.
Member Properties	**Description**
_displayNameLabel	Label component which will display the value of DisplayName
_displayValueTextBox	TextBox component which will display and allow the entering of data
Public Properties	**Description**
DisplayName	String value which will set the Text property of the _displayNameLabel

VariableName	String value representing the String representation of the variable in memory this control is attached to
VariableType	String value representing the type for the variable in memory reference by VariableName
MaximumValue	Value representing the maximum range of the value entered
MinimumValue	Value representing the minimum range of the value entered
Member Methods	**Description**
Validate	Validates that the value entered is between the MaximumValue and MinimumValue. Validates that the value entered is the proper type. Notifies the user if the value is invalid.

4.5.21 InputParameter Specifications

ParameterDisplay: The ParameterDisplay class implements the IParameter interface and displays parameter values on the user interface. The design purpose for this class was to provide a re-usable user control for use in the user interface in order to reduce code complexity and allow for changes to be easily made without requiring the IOS software to be re-written. **Table 4.5.22** outlines the specifications for the ParameterDisplay class.

There is one instance constructor that will instantiate a new instance of the ParameterDisplay class. When this constructor is invoked, it will initialize the _updateTimer member property and set it to the UpdateRate public property.

There are three member properties: _displayNameLabel, _displayValueTextBox, and _updateTimer. The _displayNameLabel is of type Label (.NET Framework), which will display the value of the DisplayName public property. The _displayValueTextBox is of type TextBox (.NET Framework) and displays the value designated by the VariableName property. The _updateTimer property is of type Timer (.NET Framework) and runs at a rate defined by the UpdateRate property. The event handler for the _updateTimer Tick event is where the value displayed by the _displayValueTextBox gets updated.

There are four public properties: DisplayName, VariableName, VariableType, and UpdateRate. The DisplayName is of type String and represents the display String for the _displayNameLabel member property. The VariableName property is of type String and represents the String representation of the variable to set in memory. The VariableType property is of type String and represents the type for the VariableName String for use in the Update member method.

There are three member methods: Update, Start, and Stop. The Update method is the event handler for the _updateTimer Tick event. First the method checks the VariableType property and invokes either the GetFloat or GetInt method of the SimulationModuleManager class. This allows the control to be responsible for setting the value in memory, which decreases the development time, because all values can be set during that period. There are two public methods: Start and Stop. The Start method invokes the Start method of the _updateTimer. The Stop method invokes the Stop method of the _updateTimer.

Class Name: ParameterDisplay	**Description:** Inherits from the IParameter interface and displays parameter data to the instructor
Instance Constructors	**Description**
ParameterDisplay	Requires not input parameters and instantiates a new instance of the ParameterDisplay class
Member Properties	**Description**
_displayNameLabel	Label that will display the Parameter name
_displayValueTextBox	TextBox set to read-only that will display the value of the Parameter
_updateTimer	Timer object that runs at a rate defined by the UpdateRate property
Public Properties	**Description**
DisplayName	String representing the display name of the parameter, this value updates the Text properties of the _displayNameTextField
VariableType	String representing the type for the variable defined by VariableName
VariableName	String representing the parameter variable name that will be displayed
UpdateRate	Integer value representing the update rate for the _updateTimer in Hertz
Public Methods	**Description**
Start	Invokes the Start method of the _updateTimer
Stop	Invokes the Stop method of the _updateTimer
Member Methods	**Description**
Update	Event handler for the _updateTimer Tick event and is responsible for updating the Text property of _displayValueTextBox with the value in memory represented by the VariableName property

Table 4.5.22 ParameterDisplay Specifications

InputParameterComboBox: The InputParameterComboBox class implements the IParameter interface and allows the instructor to choose a value from a drop down box. The design purpose for this class was to provide a re-usable user control for use in the user interface in order to reduce the code complexity and allow for changes to be made easily without requiring the IOS software to be re-written. **Table 4.5.23** outlines the specifications for the InputParameterComboBox class.

There is one instance constructor that instantiates a new instance of the InputParameterComboBox. When invoked, it will initialize all of the member properties.

There are two member properties: _comboBox and _comboBoxLabel. The _comboBox property is of type ComboBox (.NET Framework) and allows the instructor to select one of the Items from the drop down list. The _comboBoxLabel property is of type Label (.NET Framework) and will display the value of the DisplayName public property.

There are four public properties: DisplayName, VariableName, VariableType, and Items. The DisplayName property is of type String and represents the display String for the _comboBoxLabel member property. The VariableName property is of type String and represents the String representation of the variable to set in memory. The VariableType property is of type String and represents the type for the VariableName String for use in the ComboBox_SelectionChangeCommitted member method. The Items property is of type String Array and holds the items that will be listed by the _comboBox. This is represented as a getter and setter in the code. When Items is set, the new value will be checked to see if it matches the current value held by the _comboBox.Items property, if the two values match, then the setter returns. If the two values are different, the setter will set the new value to the _comboBox.Items property.

The ComboBox_SelectionChangeCommitted method is the only member method for this class and is the event handler for the _comboBox SelectionChangeCommitted event. When this method is invoked, it will invoke the SimulationModuleManager SetInt method passing in the VariableName for the variable input parameter and the _comboBox.SelectedIndex for the data input parameter.

Class Name: InputParameterComboBox	**Description:** Inherits from the IParameter interface and allows the instructor to choose an item from a ComboBox
Instance Constructors	**Description**
InputParameterComboBox	Requires no input parameters and instantiates a new instance of the InputParameterComboBox class
Member Properties	**Description**
_comboBox	ComboBox component that displays the options that can be chosen
_comboBoxLabel	Label component that displays the name associated with the _comboBox to display
Public Properties	**Description**
DisplayName	String used to set the Text property of the _comboBoxLabel component
VariableName	String representing the parameter variable name
VariableType	String value representing the type for the variable in memory

	referenced by VariableName
Items	String Array
Member Methods	**Description**
ComboBox_SelectionChangeCommitted	Occurs when an item is chosen from the drop-down list and the drop-down list is closed, updates the DisplayValue with the SelectedValue property of the _comboBox

Table 4.5.23 InputParameterComboBox Specifications

MenuBar: The MenuBar class inherits from the UserControl (.NET Framework) base class. This user control provides a static menu for the instructor across the top of the IOS window. The design purpose for implementing this as a user control rather than build it into the main form was for modularity. It reduces code complexity by breaking the user interface into smaller pieces which will reduce the amount of code per class, which in turn will reduce the amount of time during the testing phase. **Table 4.5.24** outlines the specifications for the MenuBar class.

There is one instance constructor that instantiates a new instance of the MenuBar class. When this constructor is invoked, all member properties are then initialized.

There are five member properties: _sessionTimer, _startStopSimulationButton, _activeSimulationDurationLabel, _currentDateTimeLabel, and _menu. The _sessionTimer property is of type Timer (.NET Framework) and is responsible for tracking the time duration of an active simulation. The _startStopSimulationButton property is of type Button (.NET Framework) and is the start/stop button for a simulation session. The _activeSimulationDurationLabel property is of type Label (.NET Framework) and displays the active simulation time duration that is calculated in the SessionTimer_Tick event handler of the _sessionTimer. The _menu property is of type MenuStrip (.NET Framework) and provides the following menu options to the instructor: Help, About, and Quit. If the Help option is chosen, a message window will be shown that contains basic operational instructions for Voltes Fly Simulator. The About option will show a message window with a description and short history of the Voltes Fly. The Quit option will first cause the IOS to shutdown X-Plane than close itself.

There are three member methods: SessionTimer_Tick, StartStopSimulationButton_Click, and Menu_Click. The SessionTimer_Tick method is the event handler for the _sessionTimer Tick event. This method is only invoked during an active simulation session and each time this happens, it updates _activeSimulationDurationLabel with the elapsed time since the active simulation was started. The StartStopSimulationButton_Click method is the event handler for the _startStopSimulationButton Click event. When this method is invoked, it will first check the text of the _startStopSimulationButton, if the text is “Start”, then it will invoke the Start method of the _sessionTimer, and then change the text to “Stop”. After this the method will invoke the StartSimulation method of the SimulationModuleManager class. If the text is “Stop”, then it will invoke the Stop method of the _sessionTimer, then change the text to “Start”, set the _activeSimulationDurationLabel to a value of 0, since the active simulation has been stopped, then finally invoke the StopSimulation method of the SimulationModuleManager class. The Menu_Click method is the event handler for the _menu Click event. When invoked, it will check which menu option was chosen and do the appropriate action as described earlier.

Class Name: MenuBar	**Description:** Contains static options to the instructor while the IOS is active
Instance Constructors	**Description**
MenuBar	Requires no input parameters and instantiates a new instance of the MenuBar class
Member Properties	**Description**
_sessionTimer	Timer object started when the simulation session is active to track the duration of the session
_startStopSimulationButton	Button component that when the simulation session is not active displays the label "Start". When clicked, the _sessionTimer is started and notifies that the simulation session has been started, and then displays the label "Stop. When click, the _sessionTimer is stopped and notifies that the simulation session has been stopped.
_activeSimulationDurationLabel	Label component which displays the duration of the active simulation session, when the active simulation session ends, this will reset to 0
_currentDateTimeLabel	Label component which displays the current date and time
_menu	MenuStrip component which provides the following menu options to the instructor: About, Help, and Quit
Member Methods	**Description**
SessionTimer_Tick	Invoked each time the _sessionTimer ticks as defined by the Interval property of the _sessionTimer
StartStopSimulationButton_Click	Click event handler for the _startStopSimulationButton. Changes the display label of the button, starts/stops the _sessionTimer, and notifies when the state (active or not active) of the simulation session changes.

Menu_Click	Invoked when the instructor chooses one of the following options: Help, About, and Quit. The Help and About options display a popup dialog with desired information. The Quit option will notify that the simulation session is ending and exit the IOS application

Table 4.5.24 MenuBar Specifications

StatusBar: The StatusBar class inherits from the UserControl (.NET Framework) base class. This user control displays commonly looked at flight parameter values to the instructor and is provided for convenience so that the instructor is able to view different pages in the IOS software while still being able to view specific flight parameters. **Table 4.5.25** outlines the specifications for the StatusBar class.

There is one instance constructor that instantiates a new instance of the StatusBar class and when invoked will initialize all of the member properties.

There are eight member properties: _barometricPressureDisplay, _windDirectionDisplay, _windVelocityDisplay, _visibilityDisplay, _altitudeDisplay, _magneticHeadingDisplay, _fuelQuantityDisplay, and _updateRate. The _barometricPressureDisplay is of type ParameterDisplay and displays the barometric pressure of the atmosphere. The _windDirectionDisplay property is of type ParameterDisplay and displays the wind direction of the high altitude winds. The _windVelocityDisplay property is of type ParameterDisplay and displays the velocity of the high altitude winds in knots. The _visibilityDisplay property is of type ParameterDisplay and displays the visibility value as a percentage of how opaque the atmosphere is in reference to the pilot. The _altitudeDisplay property is of type ParameterDisplay and displays the current altitude of the aircraft during an active simulation session. The _magneticHeadingDisplay is of type ParameterDisplay and displays the magnetic heading of the aircraft in degrees. The _fuelQuantityDisplay is of type ParameterDisplay and displays the remaining fuel quantity of the aircraft. The _updateRate property is of type Integer and represents the rate in Hertz. The _updateRate property is used to set the UpdateRate property for each member property of type ParameterDisplay.

There is one public property: UpdateRate, which is represented in the code as a getter and setter. When the UpdateRate is set, it first checks to see if the new value is the same as the _updateRate and if they are equal, it will exit. If the new value is different, the setter will then set the new value to _updateRate, and then set it to the UpdateRate property for each member property of type ParameterDisplay.

There are two public methods: Start and Stop. The Start method iterates through all member properties of type ParameterDisplay and invokes their Start methods. The Stop method iterates through all member properties and invokes the Stop method for all that are of type ParameterDisplay.

Class Name: StatusBar	**Description:** Displays the basic flight, weather, and aircraft parameters in real time during an active simulation session
Instance Constructors	**Description**
StatusBar	Requires no input parameters and instantiates a new instance of the StatusBar class
Member Properties	**Description**
_barometricPressureDisplay	ParameterDisplay which displays the barometric pressure in inches of mercury
_windDirectionDisplay	ParameterDisplay which displays the wind direction
_windVelocityDisplay	ParameterDisplay which displays the wind velocity
_visibility	ParameterDisplay which displays the visibility
_altitude	ParameterDisplay which displays the altitude of the aircraft
_magneticHeading	ParameterDisplay which displays the magnetic heading of the aircraft
_fuelQuantity	ParameterDisplay which displays the fuel quantity of the aircraft
_updateRate	Integer representing the rate at which to update each of the displays
Public Properties	**Description**
UpdateRate	Integer representing the rate at which to update each of the displays
Public Methods	**Description**
Start	Starts the _updateTimer
Stop	Stops the _updateTimer and resets all ParameterDisplay objects to default values

Table 4.5.25 StatusBar Specifications

IOS_UserInterface Project

The IOS_UserInterface project is the front end of the IOS software and is responsible for providing an easy to use interface to the instructor. The design purpose for breaking this into a separate project is due to the fact that the back end (Simulator project) is built in a non-object oriented fashion; because of the computational rates that must take place. This creates a good separation between the two and helps to show this distinction in an organized view to reduce the complexity of the IOS software. Within this project there is a single directory, Pages, which contains the individual pages that the instructor can interact with. The classes in this directory deal with taking inputs and display values to the instructor. This is the startup project for VS2010 and contains the IOS software's main method.

IOS_Main: The IOS_Main class is the main entry point for the IOS application. It contains a single method Main which is invoked with the IOS application is executed. This will then instantiate a new instance of the user interface which starts the entire application. VS2010 auto-generates this class when a new solution is created. **Table 4.5.26** outlines the specifications for the IOS_Main class.

Class Name: IOS_Main	**Description:** Contains the application main method for the IOS software
Public Static Methods	**Description**
Main	Main method for the IOS application, invokes the Application. Run method using a new instance of the PageBase class as the input

Table 4.5.26 IOS_Main Specifications

HealthPage: The HealthPage class inherits from the UserControl (.NET Framework) base class. This class displays the health status of all system components of the Voltes Fly Simulator. The design purpose for this class is due to the fact that the PageBase will host this user control in a TabPage so that it can use a TabControl to switch between different pages. **Table 4.5.27** outlines the specifications for the HealthPage class.

There is one instance constructor that instantiates a new instance of the HealthPage class and when invoked will initialize all of the user controls.

There are six member properties: _rudderPedalStatus, _joystickStatus, _motionBaseStatus, _xPlaneStatus, _throttleStatus, and _updateRate. The _rudderPedalStatus property is of type ParameterDisplay and displays the health status of the rudder pedals, which is set by the RudderPedalCommunication class. The _joystickStatus property is of type ParameterDisplay and displays the health status of the Joystick which is set in memory by the SimulationModuleManager. The _motionBaseStatus property is of type ParameterDisplay and displays the health status of the Motion Base which is set in memory by the SimulationModuleManager. The _xPlaneStatus property is of type ParameterDisplay and displays the health status of X-Plane which is set in memory by the SimulationModuleManager. The _throttleStatus property is of type ParameterDisplay and displays the health status of the Throttle which is set in memory by the SimulationModuleManager. The _updateRate property is of type Integer and represents the rate in Hertz to set the member properties of type ParameterDisplay UpdateRate public property too.

There is one public property: UpdateRate, which is of type Integer and is represented in the code as a getter and setter. When a new value is set, the UpdateRate will first check if the value is the same as _updateRate, if so, then the setter will exit. If the values are different, then it will iterate through all member properties of type ParameterDisplay and set their UpdateRate properties accordingly.

There are two public methods: Start and Stop. The Start method iterates through each member property of type ParameterDisplay and invokes their Start method. The Stop method iterates through each member property of type ParameterDisplay and invokes their Stop method.

Class Name: HealthPage	**Description:** Displays the health status of the simulation systems
Instance Constructors	**Description**
HealthPage	Requires no input parameters and instantiates a new instance of the HealthPage class
Member Properties	**Description**
_rudderPedalStatus	ParameterDisplay control that displays the health status of the Rudder Pedals
_joystickStatus	ParameterDisplay control that displays the health status of the Joystick
_motionBaseStatus	ParameterDisplay control that displays the health status of the Motion Base
_xplaneStatus	ParameterDisplay control that displays the health status of X-Plane
_throttleStatus	ParameterDisplay control that displays the health status of the Throttle
_updateRate	Integer value representing the rate to update the ParameterDisplay controls
Public Properties	**Description**
UpdateRate	Integer value representing the rate to update the ParameterDisplay controls
Public Methods	**Description**
Start	Iterates through each member property of type ParameterDisplay and invokes its Start method
Stop	Iterates through each member property of the type ParameterDisplay and invokes its Stop method

Table 4.5.27 HealthPage Specifications

FlightParametersPage: The FlightParametersPage inherits from the UserControl (.NET Framework) base class. This class displays all the parameters relevant to the operation of the simulator. The design purpose for this class is to provide an interface to view what the parameter values are during an active simulation in order to aide in the testing phase of development. This also allows for the other systems to be testing using the IOS software as the view into what is happening during an active simulation session. **Table 4.5.28** outlines the specifications for the FlightParametersPage class.

There are forty-six member properties: _joystickPotPositionX, _joystickPotPositionY, _roll, _pitch, _commandedRoll, _commandedPitch, _throttlePotPosition, _motionBaseServo1PotPosition, _motionBaseServo2PotPosition, _heading , _magneticHeading, _sidewaysAcceleration, _verticalAcceleration, _longitudinalAcceleration, _controlLoaderHeartbeat, _thermalTops, _thermalCoverage, _thermalClimbRate, _closestAirportTemperature, _seaLevelBarometricPressure, _visibility, _topMSL, _baseMSL, _aboveGroundLevel, _highAltitudeWindLayer, _highAltitudeWindDirection, _highAltitudeWindSpeed, _highAltitudeWindSheer, _highAltitudeWindSheerDirection, _highAltitudeWindTurbulence, _midAltitudeWindLayer, _midAltitudeWindDirection, _midAltitudeWindSpeed, _midAltitudeWindSheer, _midAltitudeWindSheerDirection, _midAltitudeWindTurbulence, _lowAltitudeWindLayer, _lowAltitudeWindDirection, _lowAltitudeWindSpeed, _lowAltitudeWindSheer, _lowAltitudeWindSheerDirection, _lowAltitudeWindTurbulence, _simulationActive, _motionBaseStatus, _joystickStatus, _throttleStatus, and _updateRate. The _joystickPotPositionX property is of type ParameterDisplay and displays the x position of the joystick. The _joystickPotPositionY property is of type ParameterDisplay and displays the y position of the joystick. The _roll property is of type ParameterDisplay and displays the roll of the aircraft in degrees. The _pitch property is of type ParameterDisplay and displays the pitch of the aircraft in degrees. The _commandedRoll property is of type ParameterDisplay and displays the commanded roll of the aircraft. The _commandedPitch property is of type ParameterDisplay and displays the commanded pitch of the aircraft. The _throttlePotPosition property is of type ParameterDisplay and displays the position of the Throttle potentiometer. The _motionBaseServo1PotPosition property is of type ParameterDisplay and displays the position of the Motion Base Servo 1 potentiometer. The _motionBaseServo2PotPosition property is of type ParameterDisplay and displays the position of the Motion Base Servo 2 potentiometer. The _heading property is of type ParameterDisplay and displays the heading of the aircraft in degrees. The _magneticHeading property is of type ParameterDisplay and displays the magnetic heading of the aircraft in degrees. The _sidewaysAcceleration property is of type ParameterDisplay and displays the sideways acceleration of the aircraft in feet per minute. The _verticalAcceleration property is of type ParameterDisplay and displays the vertical acceleration of the aircraft in feet per minute. The _longitudinalAcceleration property is of type ParameterDisplay and displays the longitudinal acceleration of the aircraft in feet per minute. The _controlLoaderHeartbeat property is of type ParameterDisplay and displays the Control Loader Heartbeat. The _thermalTops property is of type ParameterDisplay and displays the thermal tops in feet above sea level. The _thermalCoverage property is of type ParameterDisplay and displays the thermal coverage as a percentage. The _thermalClimbRate property is of type ParameterDisplay and displays the thermal climb rate in feet per minute. The _closestAirportTemperature property is of type ParameterDisplay and displays the temperature at the closet airport to the aircraft in Fahrenheit. The _seaLevelBarometricPressure property is of type ParameterDisplay and displays the barometric pressure at sea level in inches of mercury. The _visibility property is of type ParameterDisplay and displays the visibility as viewed by the pilot.

The _topMSL property is of type ParameterDisplay and displays the miles above sea level of the cloud tops. The _baseMSL property is of type ParameterDisplay and displays the miles above sea level of the cloud base. The _aboveGroundLevel property is of type ParameterDisplay and displays the miles above ground level of the clouds. The _highAltitudeWindLayer property is of type ParameterDisplay and displays the feet above sea level for High altitude winds. The _highAltitudeWindDirection property is of type ParameterDisplay and displays the direction of the HighAltitudeWindLayer in degrees. The _highAltitudeWindSpeed property is of type ParameterDisplay and displays the velocity of the HighAltitudeWindLayer in knots. The _highAltitudeWindSheer property is of type ParameterDisplay and displays the sheer of the HighAltitudeWindLayer in knots. The _highAltitudeWindSheerDirection property is of type ParameterDisplay and displays the direction of the HighAltitudeWindSheer in degrees. The _highAltitudeWindTurbulence property is of type ParameterDisplay and displays the intensity of turbulence for the HighAltitudeWindLayer. The _midAltitudeWindLayer property is of type ParameterDisplay and displays the feet above sea level for mid altitude winds. The _midAltitudeWindDirection property is of type ParameterDisplay and displays the direction of the MidAltitudeWindLayer in degrees. The _midAltitudeWindSpeed property is of type ParameterDisplay and displays the velocity of the MidAltitudeWindLayer in knots. The _midAltitudeWindSheer property is of type ParameterDisplay and displays the sheer of the MidAltitudeWindLayer in knots. The _midAltitudeWindSheerDirection property is of type ParameterDisplay and displays the direction of the MidAltitudeWindSheer in degrees. The _midAltitudeWindTurbulence property is of type ParameterDisplay and displays the intensity of turbulence for the MidAltitudeWindLayer. The _lowAltitudeWindLayer property is of type ParameterDisplay and displays the feet above sea level for Low altitude winds. The _lowAltitudeWindDirection property is of type ParameterDisplay and displays the direction of the LowAltitudeWindLayer in degrees. The _lowAltitudeWindSpeed property is of type ParameterDisplay and displays the velocity of the LowAltitudeWindLayer in knots. The _lowAltitudeWindSheer property is of type ParameterDisplay and displays the sheer of the LowAltitudeWindLayer in knots. The _lowAltitudeWindSheerDirection property is of type ParameterDisplay and displays the direction of the LowAltitudeWindSheer in degrees. The _lowAltitudeWindTurbulence property is of type ParameterDisplay and displays the intensity of turbulence for the LowAltitudeWindLayer. The _simulationActive property is of type ParameterDisplay and displays if in an active simulation state. (True if active, false otherwise). The _motionBaseStatus property is of type ParameterDisplay and displays if the Motion Base is communicating with the Control Loader. The _joystickStatus property is of type ParameterDisplay and displays if the Joystick is communicating with the Control Loader. The _throttleStatus property is of type ParameterDisplay and displays if the Throttle is communicating with the Control Loader. The _updateRate property is of type Integer and represents the rate in Hertz to set the member properties of type ParameterDisplay UpdateRate public property too.

There is instance constructor that instantiates a new instance of the FlightParametersPage class. When invoked it will initialize all the member properties and set each ParameterDisplay UpdateRate property to the _updateRate member property.

There is one public property: UpdateRate, which is of type Integer and is represented in the code as a getter and setter. When a new value is set, the UpdateRate will first check if the value is the same as _updateRate, if so, then the setter will exit. If the values are different, then it will iterate through all member properties of type ParameterDisplay and set their UpdateRate properties accordingly.

There are two public methods: Start and Stop. The Start method iterates through each member property of type ParameterDisplay and invokes their Start method. The Stop method iterates through each member property of type ParameterDisplay and invokes their Stop method.

Class Name: FlightParametersPage	**Description:** Displays all data relative to the operation of the simulator.
Instance Constructors	**Description**
FlightParametersPage	Requires no input parameters and instantiates a new instance of the FlightParametersPage class
Member Properties	**Description:**
_joystickPotPositionX	ParameterDisplay control that displays the x position of the joystick
_joystickPotPositionY	ParameterDisplay control that displays the y position of the joystick
_roll	ParameterDisplay control that displays the roll of the aircraft in degrees
_pitch	ParameterDisplay control that displays the pitch of the aircraft in degrees
_commandedRoll	ParameterDisplay control that displays the commanded roll of the aircraft
_commandedPitch	ParameterDisplay control that displays the commanded pitch of the aircraft
_throttlePotPosition	ParameterDisplay control that displays the position of the Throttle potentiometer
_motionBaseServo1PotPosition	ParameterDisplay control that displays the position of the Motion Base Servo 1 potentiometer
_motionBaseServo2PotPosition	ParameterDisplay control that displays the position of the Motion Base Servo 2 potentiometer
_heading	ParameterDisplay control that displays the heading of the aircraft in degrees
_magneticHeading	ParameterDisplay control that displays the magnetic heading of the aircraft in degrees
_sidewaysAcceleration	ParameterDisplay control that displays the sideways acceleration of the aircraft in feet per minute
_verticalAcceleration	ParameterDisplay control that displays the vertical acceleration of the aircraft in feet per minute
_longitudinalAcceleration	ParameterDisplay control that displays the longitudinal acceleration of the aircraft in feet per minute
_controlLoaderHeartbeat	ParameterDisplay control that displays the Control Loader Heartbeat

_thermalTops	ParameterDisplay control that displays the thermal tops in feet above sea level
_thermalCoverage	ParameterDisplay control that displays the thermal coverage as a percentage
_thermalClimbRate	ParameterDisplay control that displays the thermal climb rate in feet per minute
_closestAirportTemperature	ParameterDisplay control that displays the temperature at the closet airport to the aircraft in Fahrenheit
_seaLevelBarometricPressure	ParameterDisplay control that displays the barometric pressure at sea level in inches of mercury
_visibility	ParameterDisplay control that displays the visibility as viewed by the pilot
_topMSL	ParameterDisplay control that displays the miles above sea level of the cloud tops
_baseMSL	ParameterDisplay control that displays the miles above sea level of the cloud base
_aboveGroundLevel	ParameterDisplay control that displays the miles above ground level of the clouds
_highAltitudeWindLayer	ParameterDisplay control that displays the feet above sea level for High altitude winds
_highAltitudeWindDirection	ParameterDisplay control that displays the direction of the HighAltitudeWindLayer in degrees
_highAltitudeWindSpeed	ParameterDisplay control that displays the velocity of the HighAltitudeWindLayer in knots
_highAltitudeWindSheer	ParameterDisplay control that displays the sheer of the HighAltitudeWindLayer in knots
_highAltitudeWindSheerDirection	ParameterDisplay control that displays the direction of the HighAltitudeWindSheer in degrees
_highAltitudeWindTurbulence	ParameterDisplay control that displays the intensity of turbulence for the HighAltitudeWindLayer
_midAltitudeWindLayer	ParameterDisplay control that displays the feet above sea level for Mid altitude winds
_midAltitudeWindDirection	ParameterDisplay control that displays the direction of the MidAltitudeWindLayer in degrees
_midAltitudeWindSpeed	ParameterDisplay control that displays the velocity of the MidAltitudeWindLayer in knots

_midAltitudeWindSheer	ParameterDisplay control that displays the sheer of the MidAltitudeWindLayer in knots
_midAltitudeWindSheerDirection	ParameterDisplay control that displays the direction of the MidAltitudeWindSheer in degrees
_midAltitudeWindTurbulence	ParameterDisplay control that displays the intensity of turbulence for the MidAltitudeWindLayer
_lowAltitudeWindLayer	ParameterDisplay control that displays the feet above sea level for Low altitude winds
_lowAltitudeWindDirection	ParameterDisplay control that displays the direction of the LowAltitudeWindLayer in degrees
_lowAltitudeWindSpeed	ParameterDisplay control that displays the velocity of the LowAltitudeWindLayer in knots
_lowAltitudeWindSheer	ParameterDisplay control that displays the sheer of the LowAltitudeWindLayer in knots
_lowAltitudeWindSheerDirection	ParameterDisplay control that displays the direction of the LowAltitudeWindSheer in degrees
_lowAltitudeWindTurbulence	ParameterDisplay control that displays the intensity of turbulence for the LowAltitudeWindLayer
_simulationActive	ParameterDisplay control that displays if in an active simulation state. (true if active, false otherwise)
_motionBaseStatus	ParameterDisplay control that displays if the Motion Base is communicating with the Control Loader
_joystickStatus	ParameterDisplay control that displays if the Joystick is communicating with the Control Loader
_throttleStatus	ParameterDisplay control that displays if the Throttle is communicating with the Control Loader
_updateRate	Integer value representing the rate to update the ParameterDisplay controls
Public Properties	**Description**
UpdateRate	Integer value representing the rate to update the ParameterDisplay controls
Public Methods	**Description**

Start	Iterates through each member property of type ParameterDisplay and invokes its Start method
Stop	Iterates through each member property of the type ParameterDisplay and invokes its Stop method

Table 4.5.28 FlightParametersPage Specifications

ScenarioPage: The ScenarioPage inherits from the UserControl (.NET Framework) base class. This class allows the instructor to set scenario data for a simulation session. The design purpose for this class is to provide a clean interface to the instructor for entering scenario data for a simulation session. This class was developed to reduce the code complexity needed for the PageBase and allows for the look and feel of the ScenarioPage to be modified as needed without requiring the entire user interface to be re-written, therefore will speed up the development and testing phase of the project. **Table 4.5.29** outlines the specifications for the ScenarioPage class.

There is one instance constructor that will instantiate a new instance of the ScenarioPage class and when invoked will initialize all of the member properties.

There are thirty one member properties: _cloudCoverTypeComboBoxParam, _topMSLInputParam, _baseMSLInputParam, _aboveGroundLevelInputParam, _highAltitudeWindLayerInputParam, _highAltitudeWindLayerDirectionInputParam, _highAltitudeWindSpeedInputParam, _highAltitudeWindSheerInputParam, _highAltitudeWindSheerDirectionInputParam, _highAltitudeWindTurbulenceInputParam, _midAltitudeWindLayerInputParam, _midAltitudeWindLayerDirectionInputParam, _midAltitudeWindSpeedInputParam, _midAltitudeWindSheerInputParam, _midAltitudeWindSheerDirectionInputParam, _midAltitudeWindTurbulenceInputParam, _lowAltitudeWindLayerInputParam, _lowAltitudeWindLayerDirectionInputParam, _lowAltitudeWindSpeedInputParam, _lowAltitudeWindSheerInputParam, _lowAltitudeWindSheerDirectionInputParam, _lowAltitudeWindTurbulenceInputParam, _thermalTopsInputParam, _thermalCoverageInputParam, _thermalClimbRateInputParam, _closestAirportTemperatureInputParam, _seaLevelBarometricPressureInputParam, _visibilityInputParam, _precipitationInputParam, _runwayConditionComboBoxParam, and _runwayNameComboBoxParam. The _cloudCoverTypeComboBoxParam property is of type InputParameterComboBox and allows the instructor to choose one of the CloudCoverTypes. The __topMSLInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the miles above sea level for the cloud tops. The _baseMSLInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the miles above sea level for the cloud base. The _aboveGroundLevelInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the miles above ground level for the clouds. The _highAltitudeWindLayerInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the feet above sea level for the high altitude winds. The _highAltitudeWindLayerDirectionInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the direction of the high altitude winds in degrees. The _highAltitudeWindSpeedInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the velocity of the high altitude winds in knots. The _highAltitudeWindSheerInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the sheer of the high altitude winds in knots. The _highAltitudeWindSheerDirectionInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the direction of the high altitude wind shear in degrees. The _highAltitudeWindTurbulenceInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the intensity for the high altitude wind. The _midAltitudeWindLayerInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the feet above sea level for the mid altitude winds. The _midAltitudeWindLayerDirectionInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the direction of the mid altitude winds in degrees.

The _midAltitudeWindSpeedInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the velocity of the mid altitude winds in knots. The _midAltitudeWindSheerInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the sheer of the mid altitude winds in knots. The _midAltitudeWindSheerDirectionInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the direction of the mid altitude wind shear in degrees. The _midAltitudeWindTurbulenceInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the intensity for the mid altitude wind. The _lowAltitudeWindLayerInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the feet above sea level for the low altitude winds. The _lowAltitudeWindLayerDirectionInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the direction of the low altitude winds in degrees. The _lowAltitudeWindSpeedInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the velocity of the low altitude winds in knots. The _lowAltitudeWindSheerInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the sheer of the low altitude winds in knots. The _lowAltitudeWindSheerDirectionInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the direction of the low altitude wind shear in degrees. The _lowAltitudeWindTurbulenceInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the intensity for the low altitude wind. The _thermalTopsInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the height of thermal tops above sea level in feet. The _thermalCoverageInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the thermal coverage as a percentage. The _thermalClimbRateInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the thermal climb rate in feet per minute. The _closestAirportTemperatureInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the temperature at the closest airport to the initial position of the aircraft in Fahrenheit. The _seaLevelBarometricPressureInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the barometric pressure at sea level in inches of mercury. The _visibilityInputParam property is of type InputParameter and allows the instructor to enter an Integer value representing the visibility as a percentage. The _precipitationComboBoxParam property is of type InputParameterComboBox and allows the instructor to choose one of the PrecipitationTypes. The _runwayConditionComboBoxParam property is of type InputParameterComboBox and allows the instructor to choose one of the available runways.

Class Name: ScenarioPage	**Description:** Allows the instructor to set scenario values for the simulation session
Instance Constructors	**Description**
ScenarioPage	Requires no input parameters and instantiates a new instance of the ScenarioPage class
Member Properties	**Description**
_cloudCoverTypeComboBoxParam	InputParameterComboBox control to choose one of the CloudCoverTypes
_topMSLInputParam	InputParameter control to enter in an integer value representing the miles above sea level for the cloud tops
_baseMSLInputParam	InputParameter control to enter in an integer value representing the miles above sea level for the cloud base
_aboveGroundLevelInputParam	InputParameter control to enter an integer value representing the miles above ground level for the clouds
_highAltitudeWindLayerInputParam	InputParameter control to enter the feet above sea level for the high altitude winds
_highAltitudeWindLayerDirectionInputParam	InputParameter control to enter the direction of the high altitude winds in degrees
_highAltitudeWindSpeedInputParam	InputParameter control to enter the velocity of the high altitude winds in knots
_highAltitudeWindSheerInputParam	InputParameter control to enter the sheer of the high altitude

	wind in knots
_highAltitudeWindSheerDirectionInputParam	InputParameter control to enter the direction of the high altitude wind shear in degrees
_highAltitudeWindTurbulenceInputParam	InputParameter control to enter the intensity of turbulence for the high altitude wind
_midAltitudeWindLayerInputParam	InputParameter control to enter the feet above sea level for the mid altitude winds
_midAltitudeWindLayerDirectionInputParam	InputParameter control to enter the direction of the mid altitude winds in degrees
_midAltitudeWindSpeedInputParam	InputParameter control to enter the velocity of the mid altitude winds in knots
_midAltitudeWindSheerInputParam	InputParameter control to enter the sheer of the mid altitude wind in knots
_midAltitudeWindSheerDirectionInputParam	InputParameter control to enter the direction of the mid altitude wind shear in degrees
_midAltitudeWindTurbulenceInputParam	InputParameter control to enter the intensity of turbulence for the mid altitude wind
_lowAltitudeWindLayerInputParam	InputParameter control to enter the feet above sea level for the low altitude winds
_lowAltitudeWindLayerDirectionInputParam	InputParameter control to enter the direction of the low altitude winds in degrees
_lowAltitudeWindSpeedInputParam	InputParameter control to enter the velocity of the low altitude winds in knots
_lowAltitudeWindSheerInputParam	InputParameter control to enter the sheer of the low altitude wind in knots
_lowAltitudeWindSheerDirectionInputParam	InputParameter control to enter the direction of the low altitude wind shear in degrees
_lowAltitudeWindTurbulenceInputParam	InputParameter control to enter the intensity of turbulence for the low altitude wind
_thermalTopsInputParam	InputParameter control to enter the thermal tops in feet above sea level

_thermalCoverageInputParam	InputParameter control to enter the thermal coverage's as a percentage
_thermalClimbRateInputParam	InputParameter control to enter the thermal climb rate in feet per minute
_closestAirportTemperatureInputParam	InputParameter control to enter the temperature at the closest aircraft in Fahrenheit
_seaLevelBarometricPressureInputParam	InputParameter control to enter the barometric pressure at sea level in inches of mercury
_visibilityInputParam	InputParameter control to enter the visibility as a percentage
_precipitationInputParam	InputParameterComboBox control to choose one of the PrecipitationTypes
_runwayCondition	InputParameterComboBox control to choose one of the RunwayConditionTypes
_runwayName	InputParameterComboBox control to choose a runway

Table 4.5.29 ScenarioPage Specifications

PageBase: The PageBase class inherits from the Form (.NET Framework) base class. This class represents the primary Windows Form that will host all the different pages in which an instructor will interact with. The design purpose for this class is to provide a base where pages could be viewed while allowing the ability to add and remove pages without much programming effort. **Table 4.5.30** outlines the specifications for the PageBase class.

There is one instance constructor that instantiates a new instance of the PageBase class and initializes all of the visual components.

There are eight member properties: _verticalSplitterPanel, _horizontalSplitterPanel, _tabPageContainer, _scenarioTabPage, _healthTabPage, _flightParametersTabPage, _statusBar, and _menuBar. The _verticalSplitterPanel property is of type SplitterPanel (.NET Framework) and provides separation between the _statusBar user control from the _tabPageContainer, which makes it easier to organize the member property user controls. The _horizontalSplitterPanel property is of type SplitterPanel (.NET Framework) and provides separation between the _verticalSplitterPanel and the _menuBar use control, which will make it easier to organize the member property user controls. The _tabPageContainer property is of type TabControl (.NET Framework) and contains a separate tab for the _flightParametersPage, _healthPage, and _scenarioPage. The _scenarioTabPage property is of type TabPage (.NET Framework) and is the container for the _scenarioPage user control. The _healthTabPage property is of type TabPage (.NET Framework) and is the container for the _healthPage user control. The _flightParametersTabPage property is of type TabPage (.NET Framework) and is the container for the _flightParametersPage user control. The _statusBar property is of type StatusBar and contains the options as defined by the StatusBar class specification. The _menuBar property is of type MenuBar and displays common flight parameters to the instructor as defined by the MenuBar class specification.

Class Name: PageBase	**Description:** The IOS primary windows form that the instructor will interact with.
Instance Constructors	**Description**
PageBase	Requires no input parameters and instantiates a new instance of the PageBase class
Member Properties	**Description**
_verticalSplitterPanel	SplitterPanel component used to separate the StatusBar control from the _tabPageContainer
_horizontalSplitterPanel	SplitterPanel component used to separate the _verticalSplitterPanel from the MenuBar control
_tabPageContainer	TabControl component which allows the user to switch between which page is visible
_scenarioTabPage	TabPage component containing the ScenarioPage
_healthTabPage	TabPage component containing the HealthPage
_flightParametersTabPage	TabPage component containing the FlightParametersPage
_statusBar	StatusBar control that displays basic flight parameter values to the instructor during an active simulation session
_menuBar	MenuBar control that contains information regarding the simulation session (i.e. simulation session, current time, etc...)

Table 4.5.30 PageBase Specifications

4.6 Throttle Design

Electrical

The electrical design of the throttle device is very straight forward. After all the research was done it was understood that there were not many options available in this area: the throttle's signal is to be modulated by a potentiometer. **Figure 4.6.1** shows the configuration, the standard pot has 3 input terminals: voltage supply, ground, and output. The recommended value for the potentiometer is 100kOhm (see research sections). However, different resistance values can be experimented with and should give similar results. Using different pot values will yield different max byte values on the software end of the project once the control loader converts the signal to a digital form. Calibration must be done on this side to define the meaning of the different range of byte values of the throttle's output.

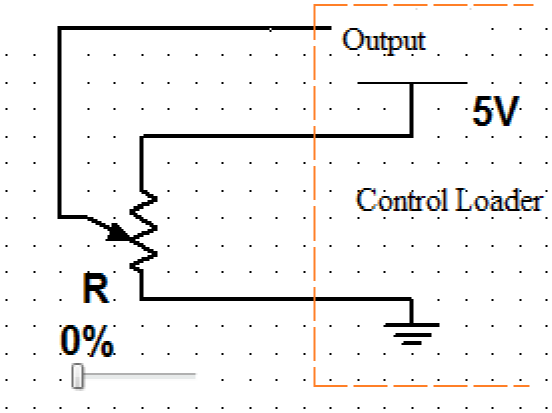


Figure 4.6.1 Throttle Circuitry

The throttle does not need any other source of power other than the digital Vcc signal which will be provided by the control loader. There are many options for the potentiometer: single-turn, multi-turn, logarithmic, linear, slide or rotary. Logarithmic potentiometers offer a response most useable in audio applications, so a linear potentiometer will be used. Because of the mechanical structure within a throttle, a rotary potentiometer is best for this application. A multi turn shaft is not necessary because the movement of the handle by the player will not cause the potentiometer to rotate a whole revolution. Therefore, a single-turn rotary potentiometer with a linear response is the best option for this project. The shaft of the potentiometer is represented in the circuit by the percentage value and slider below the resistor. When the resistance value of the potentiometer is set to max, the throttle is off. The plan is that as you gradually push the lever it will rotate the pot causing the resistance to lower. As the resistance lowers, the output will get closer and closer to 5V. Using Multisim, a simulation was made to show the analog throttle signal. With a virtual oscilloscope the results were recorded in a graph showing output voltage versus time as seen in **Figure 4.6.2.**

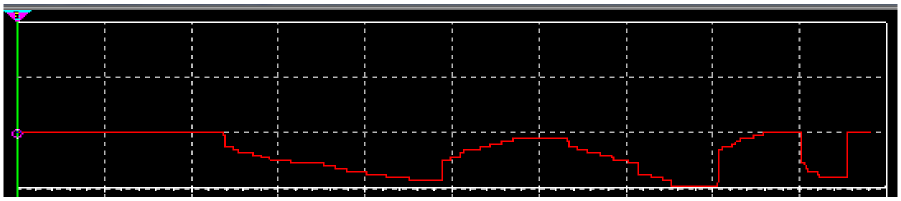


Figure 4.6.2 Throttle's Analog signal

Mechanical

There will be a plastic rod which will serve as the lever for the device. It can be pushed or pulled and it rotates around the axis of a gear it's attached to. This lever will be plastic and will be allowed to rotate back 45 degrees with respect to its flat surface base and 135 degrees forward. In order to avoid future accidents, this attachment will be both glued and tied until considered strong enough to resist gameplay. A wooden rectangular structure will encase the throttle system, with a 3 cm wide slit in the center to allow the lever to move. The length of this slit will define the rotational constraints of the lever's movement. This value might be adjusted during the testing phase but it is estimated to be 6 cm. The physical dimensions of this wooden encasing are 18x6x8cm. The system will consist of two gears: one is attached to the lever, with a support rod at the center opening. This rod will rest on 2 metal plates (with a height of 7 cm) on each side of the device for stabilization. The metal plates will be bolted to the base of this platform. The bottom of the large gear will make contact with a smaller gear whose axial opening will be attached to the rotary shaft of the potentiometer. In order for the pot's shaft to spin, the rest of the potentiometer's body must be constrained so it will be attached to the same metal plate. Also, the other side of this small gear will be supported by the metal plate on the other side. A second gear with a smaller diameter is included in the design so that a wider range of the pot's resistance variation can be taken advantage of. While the bigger gear will only rotate about 90 degrees, this second gear will make almost a complete revolution. This gear ratio between them will allow for the full turning of the rotary potentiometer. This interaction is shown in **Figure 4.6.3** below.

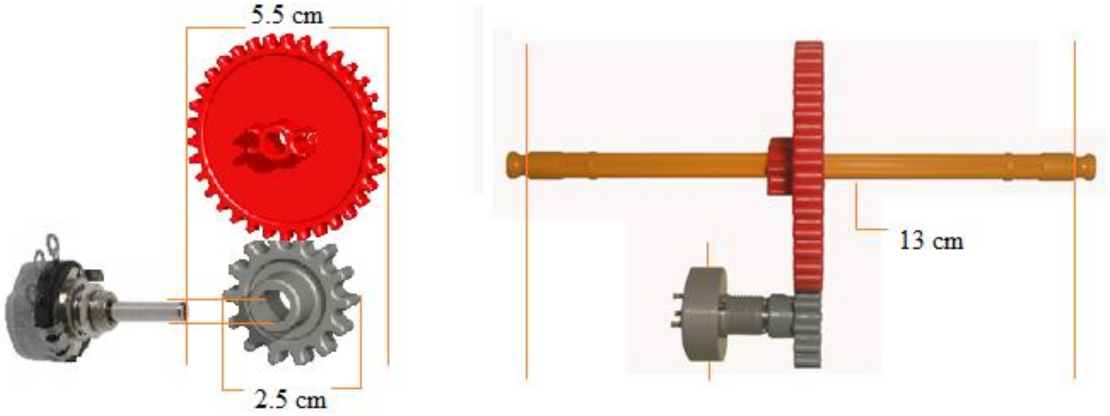


Figure 4.6.3 Gears and pot. Interaction

The opposite side of the potentiometer will then be secured to the metal plate on that side. the length of the support rod is 13 cm and it will provide balance to the contraption. This parameter was taken into consideration when deciding on a length for the protective cover. The 5 cm difference between the cover and the end of the rod was intentionally kept to provide spacing for the metal support plates. Another option could have been to attach this support rod to the walls of the surrounding wooden box. However, it was decided not to implement it in this fashion so that the contraption on the inside was independent from the protective cover. More over, on the left side of the cover there will be a circular opening from which the pot's voltage, ground, and output cables will come out. These cables can be combined into one shielded cable for safety

4.7 Joystick Design

Electrical

The electrical design of the joystick itself will be composed of two subsystems: directional, and force feedback. The system's electrical design is shown in **Figure 4.7.1.**

The mechanical structure of the device is discussed in 4.7.2. In order to allow for 2 degrees of freedom, the device must track signals for each axis. To accomplish this task, each axis of the moving shaft will be attached to a rotary potentiometer as seen in the researched examples and in the left side of the diagram below. Because of the movement constraints that will be applied to the joystick, the complete 270 degrees of the potentiometers will not be used, therefore the value of resistance used will be 470KOhm. The reasoning behind this decision is as follows: if the rotation of the handle is constrained to 60 degrees then (60/270) 20% of the original range is used. 20% of 470Kohm is 104K which is a typical value for the joysticks in the researched examples. Each axis is basically similar in design to the throttle's design. The voltage and ground signals will come from the control loader and the output terminal will be sent out to it. The control loader will take care of these two analog signals ultimately sending them to the IOS. As the player rotates the joystick's handle a load cell within the system measures the force applied on it by this said rotation. The force feedback system shown on the right side of this diagram only applies to one axis of movement for simplicity. In the complete design there is a duplicate load cell, amp, and motor system corresponding to the other axis. The second load cell will measure the movement in this said axis and trigger the second motor which is dedicated to. Having a motor apply force to the system on each axis will yield a more robust and realistic effect: which ever way the user moves the joystick, they will feel a force fighting against their movement.

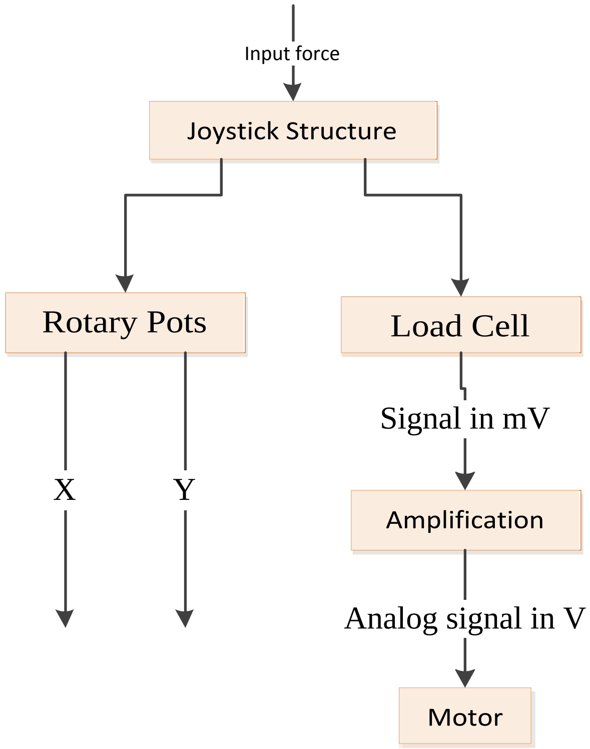


Figure 4.7.1 Overall Joystick design

The motor to be used in the force feedback model is the PITTMAN Gearhead motor GM9236C460. According to its specifications, it is usable from 6V to 24V DC. The goal is to take full advantage of this range of motor response, so the input range of the load cell circuit must make it possible to do so. The load cell to be used in the circuit is a Moment Compensated Load Cell Model: ESP4 **[4.7.1]** designed to sense low capacity forces. It is ideal for our project because of its size, sensitivity, and price. Most importantly, it provides both tension and compression readings which are necessary for our design to work properly. Let's say the player moves the joystick to the left, and the load cell is compressed making the motor turn counter clockwise. On the other hand, when the player pulls the handle in the opposite direction, the tension felt by the load cell will output a voltage in the opposite polarity which will trigger the gear motor to rotate in the opposite direction. That is why it is necessary to use a bipolar load cell. The ESP4 will provide a weight measurement proportional to the reference voltage provided, which is recommended to be 10V. According to its specs, for every voltage unit provided to power the device, the range of signal output increases by 1mV.This means that if this device is powered at 10V, its output would range from 1-10mV depending on the weight applied to its sensors.

Rin	395 +/- 10 Ohms
Rout	350 +/- 30 Ohms
Rated Output	1 +/- .1mV/V
Recommended Vref	10V
Weight range	.6-3kg

Table 4.7.1 ESP4 Specifications

It was decided to use 10V as a reference input because it was the recommended value by the manufacturers, for optimal operation. One needs to keep in mind, that in order to amplify the signal, power must also be supplied to the circuit which will amplify the load cell's output. The maximum voltage for the operation of the motor is 24 volts, so the circuit must not surpass this value for any reason. Therefore, a 15VDC power supply is acceptable. This will assure that the maximum value for the motor will never be reached, and it is enough to supply the load cell with its 10VDC reference input. This component will be a COTS item, a generic 15VDC regulated power supply purchased **[4.7.2].** It is possible to use the complete 15v to power the load cell, eliminating the need of implementing a voltage divider circuit to bring the magnitude down. However, because this extra step is simple to do, the design will stick to the recommended 10v reference input. The amplification consists of a simple Non-inverting amplifier Op amp where the input signal is attached to its positive terminal. The load cell excitation signal is in millivolts range and must be amplified to the voltage range supported by the motor. The gain chosen for this process is 1500, which was decided after careful examination. Let's say the load cell is excited by the movement of the joystick handle and creates its minimal response of 1millivolt. For the motor to respond to this signal, it must be amplified by a factor of 6V/ (1mV) = 6000. However, a signal of this magnitude would reach the 15 volt ceiling by a load cell response of just 5 millivolts. This setting would not take full advantage of the range available. Therefore, several other gain values were studied and a line chart was assembled to compare (see **Figure 4.7.2).** The X axis corresponds to the range of possible load cell outputs, and the Y axis is the output of the amplifier going into the motor. It is easy to see that even though a gain of 6000 will reach the 6V at 1mV of excitation, it does reaches the maximum output at only 5 millivolts of excitation. Even a gain of 3000 reaches the 15V limit too soon, notice that the green line (corresponding to a gain of 1500) takes the most advantage of the available range. The only downside of using this gain value is that the 6V required to operate the motors will not be reached until the load cell is excited to about 3.5 mV. However, this is something the group is willing to allow in order to have the widest operating range for the motors. Also, in contrary to its specifications, 5 volts was found to be enough to power the motors after testing the motors at low voltage levels.

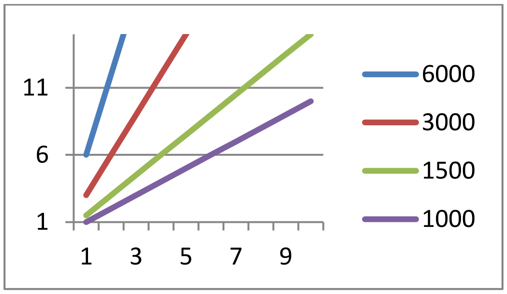


Figure 4.7.2 Effect of gain values in motor input vs. possible load cell signals

The gain of a Non-inverting amplifier circuit is found by the formula Av=1+(Rf/Rin). From this equation the relationship between the two resistors is derived: (1499)Rin=Rf. Therefore, if the feedback resistor is 1500Ohms, Rin has to be 1 Ohm in order to create the desired gain. The Op amp used is a generic amplifier type AP358 as seen in the schematic in **Figure 4.7.3.** Its datasheet has an example for amplification in which the positive terminal is connected to the input signal in a non-inverting set up similar to the one described above. This IC actually comes with a second amplifier which will be used for the amplifier for the force feedback system corresponding to the other axis of movement. One great quality about the AP358 is that it can be operated at 5V as well, so the second amp can also be used in any future digital applications or additions to the system. The amp and load cell are powered with a R800 9407 15VDC power supply. In order to obtain the 10V power supply for the load cell, a voltage divider circuit is used to extract 2/3 or the original supply. The voltage divider equation 15v=10v (R2/R1+R2) is solved to get the resistor relationship: R2=2*R1.

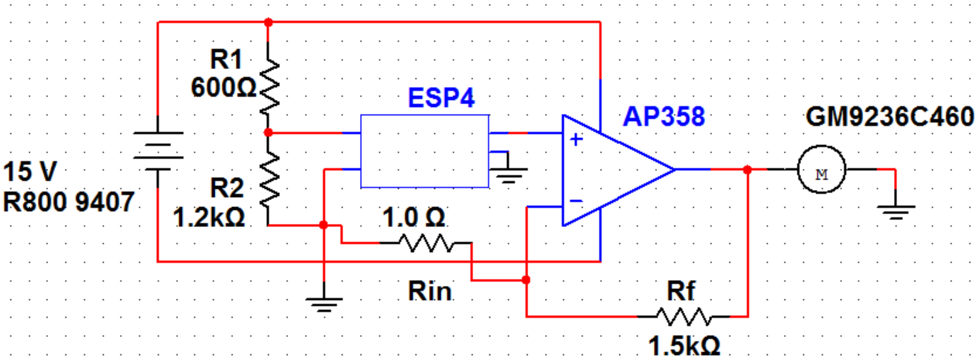


Figure 4.7.3 Load Cell, Amplifier, and Motor

From this equation the resistor values 600 and 1200 are obtained. These values need to be checked in the testing phase of the project to ensure that the right amount of current is being generated. The same goes for the amplifier resistors, since 1ohm is a low value, different resistor combinations with a ratio of 1500 should be tested to choose the best option. It is notable from this circuit that the motor is not fed any voltage when the load cell's response is zero. This is because the motor is dependant on the force applied on the load cell. Keep in mind that a duplicate circuit is implemented for the other axis of the joystick controller, so that the force effect can be felt in either direction of movement. The same power supply line and ground will be used for the second axis; however it will need 4 new resistors. The values of these resistors will be the same in magnitude because the electrical design for each axis is the same.

Mechanical

The mechanical structure of the joystick will resemble an arcade style shaft. A universal joint will be tightly bolted to a base which will also have the motor and circuitry placed on top of. Because our focus of the project is the electrical design, the group will not worry about aesthetics and just go with the safest, cheapest option. **Figure 4.7.4(a)** shows the universal joint used as the handle for the joystick. The bottom of this handle has a shaft for each axis which makes it easy to attach a belt pulley system to each. Each axis shaft rotates as the handle is moved. The belt will also move as the player rotates the handle. By default, this universal joint allows for 90 degree rotation in each axis. However, joysticks do not usually have this much freedom of movement. Therefore, a physical constraint will be implemented around the handle to prevent the player from passing more than 45-60 degrees of rotation. This constraint will consist of a circular opening in the container in which the joystick will be housed. The other end of each belt will be attached to a crank which is itself attached to the shaft of the motor for that axis. This crank is will be an extension of shaft#1 in **Figure 4.7.4(b)**. Whenever the belt moves, it causes the crank and shaft of the motor to rotate. The crank will have an additional link which will push or pull on the load cell's platform. The load cell will lay flat between the link and the base of the container.

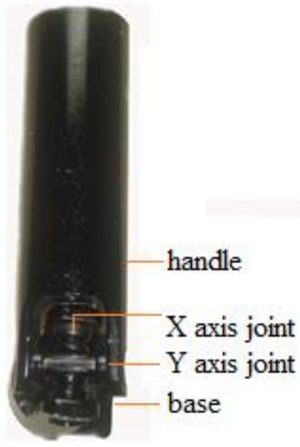


Figure 4.7.4 (a) Universal Joint

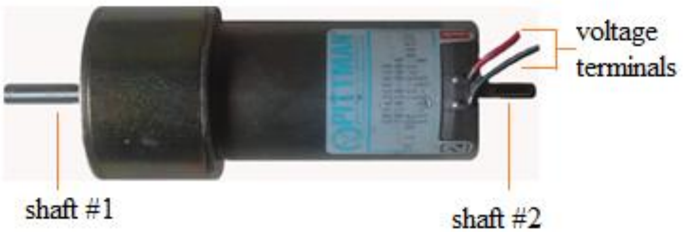


Figure 4.7.4 (b) Motor

As you can see, shaft #2 is smaller in diameter; it has a higher amount of rotations because of the gear ratio within the motor. The potentiometers will be placed on this shaft. A securing contraption will be needed to attach the shaft of the rotary potentiometer to the shaft of the motor. The main portion (circular in shape) of the potentiometer will be attached to a metal plate which will be bolted to the base. This is done so that when the shaft rotation takes place, only the shaft will spin rather than the whole device. The base of the joystick container will be wooden, with a 30x30 cm base and with a height of approximately 6cm. This was decided after laying out both motors in the estimated positions with respect to the universal joint which will be at the center. The joysticks' handle has a base length of 4x4 cm and each motor has a length of 15 cm. **Figure 4.7.5** shows the current layout for the different physical parts of the joystick as they would look from a top view perspective.

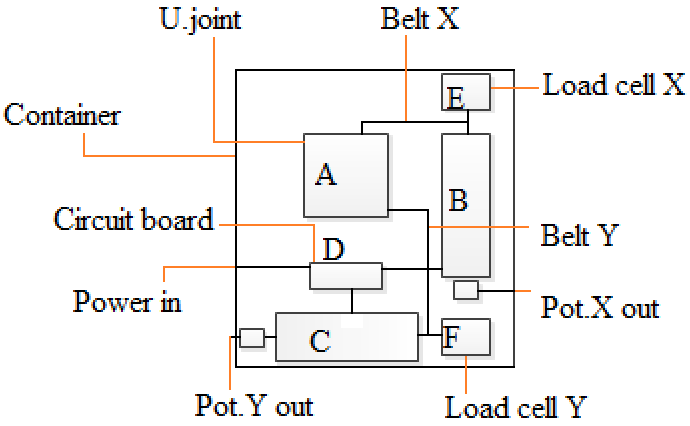


Figure 4.7.5 Physical Placements

There will be 3 openings into the box or container. One small hole on the side with a diameter of 2 centimeters will be used to let the two Pot wires out to the control loader. These will be shielded together for safety and protection of the wires. The other opening will be the same in magnitude and it will serve as an entrance for the 15V power supply's cable. The third opening will be at the top of the box, and it will be circular. The diameter of this opening is essential to control the moving constraint of the handle. The wider it is, the more freedom of rotation will be given to the handle. It is estimated, that this diameter will be 7cm. However; please note that when the testing phase of this device takes place, this number might change depending on the effect it has the game experience.

The belts used to connect the cranks to the U.Joint have to be very strong, and must provide enough tension so that the rotation can be translated effectively. See the reference section to see where these parts will be purchased **[4.7.3]**. The link that will push on the load cell will be perpendicular to the crank and will be securely attached to it.

5. Design Summary of Hardware & Software

5.1 Control Loader

Figure 5.1 shows a complete schematic of the Control Loader Unit and the overall interaction between the other components.

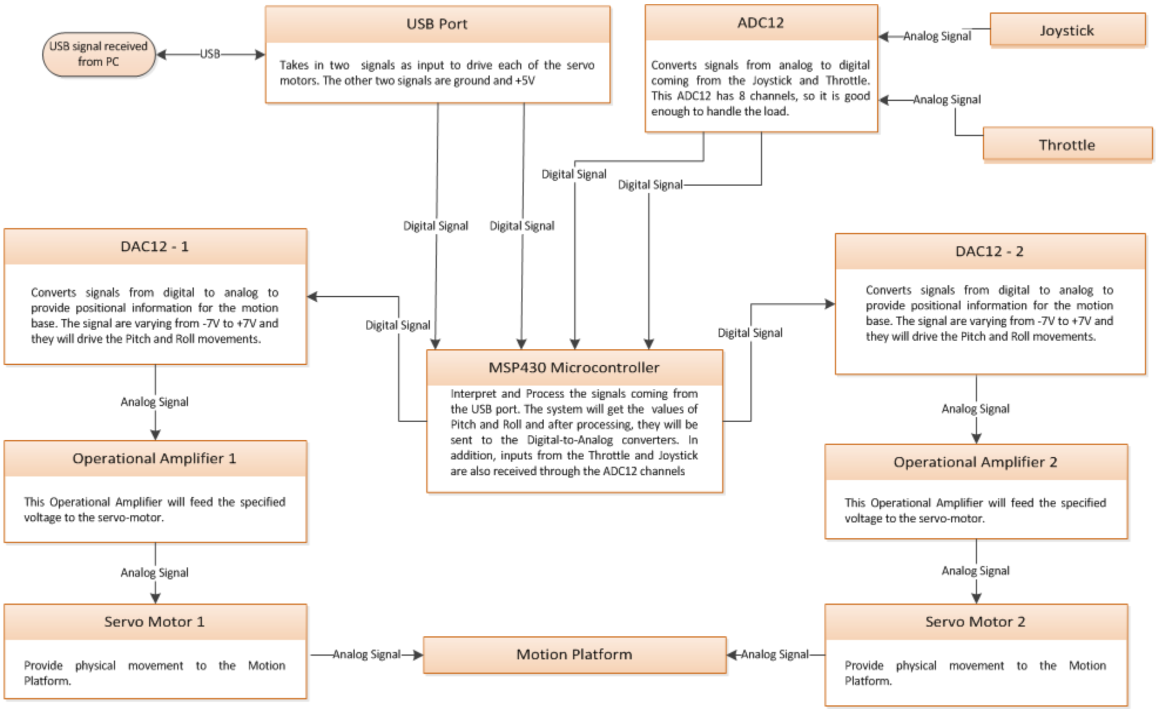

USB signal received from PC
USB
USB Port
Takes in two signals as input to drive each of the servo motors. The other two signals are ground and +5V
ADC12
Converts signals from analog to digital coming from the Joystick and Throttle. This ADC12 has 8 channels, so it is good enough to handle the load.
Analog Signal
Joystick
Analog Signal
Throttle
Digital Signal
Digital Signal
Digital Signal
Digital Signal
DAC12 - 1
Converts signals from digital to analog to provide positional information for the motion base. The signal are varying from -7V to +7V and they will drive the Pitch and Roll movements.
Digital Signal
MSP430 Microcontroller
Interpret and Process the signals coming from the USB port. The system will get the values of Pitch and Roll and after processing, they will be sent to the Digital-to-Analog converters. In addition, inputs from the Throttle and Joystick are also received through the ADC12 channels
Digital Signal
DAC12 - 2
Converts signals from digital to analog to provide positional information for the motion base. The signal are varying from -7V to +7V and they will drive the Pitch and Roll movements.
Analog Signal
Operational Amplifier 1
This Operational Amplifier will feed the specified voltage to the servo-motor.
Analog Signal
Operational Amplifier 2
This Operational Amplifier will feed the specified voltage to the servo-motor.
Analog Signal
Analog Signal
Servo Motor 1
Provide physical movement to the Motion Platform.
Analog Signal
Motion Platform
Analog Signal
Servo Motor 2
Provide physical movement to the Motion Platform.

Figure 5.1 Control Loader Layout.

Figure 5.1.2 shows the external circuits required to drive the servo motors. They will be connected to two output pins of the microprocessor. The cascading circuit allows for the servomotors to receive negative voltage values (for clockwise rotation) and it amplifies them to an acceptable range.

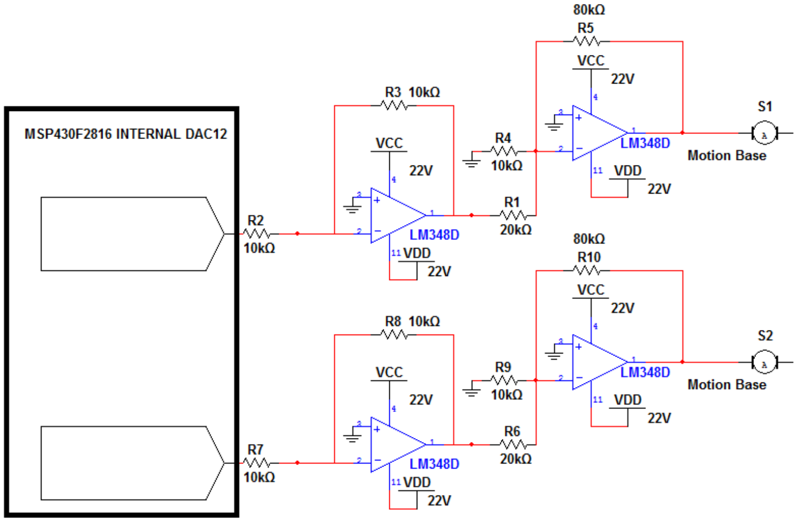


Figure 5.1.2 CL to Servo Motors Schematic

Figure 5.1.3 shows the overall layout for the firmware that is going to reside within the microcontroller. The Control Loader will act basically as a signal processing unit. First, the internal ADC and DAC devices must be enabled and their corresponding pins must be configured and defined. The pins that will output the packetted data and receive from the USB module will also have to be configured. After the inputs and outputs are set up, the software goes into an infinite loop that will send the voltage values in bytes for the servo motor through its corresponding pins. Before starting the process over, it will listen on each pin, pack the information, and send it to the USB interface.

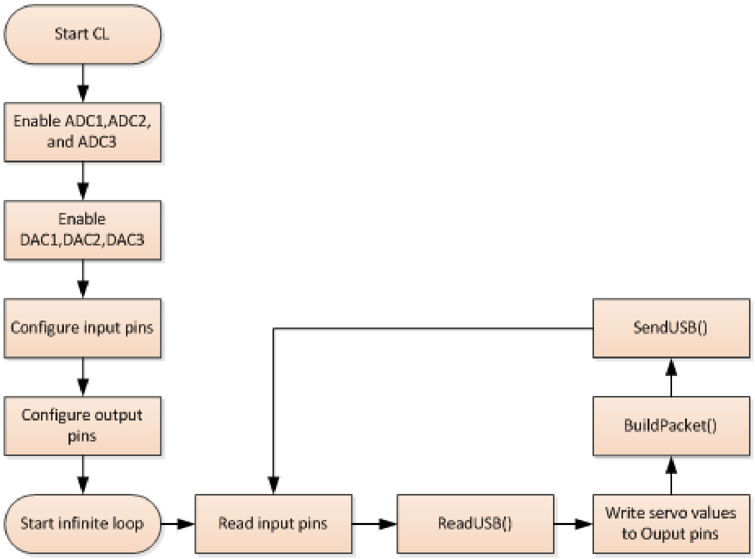
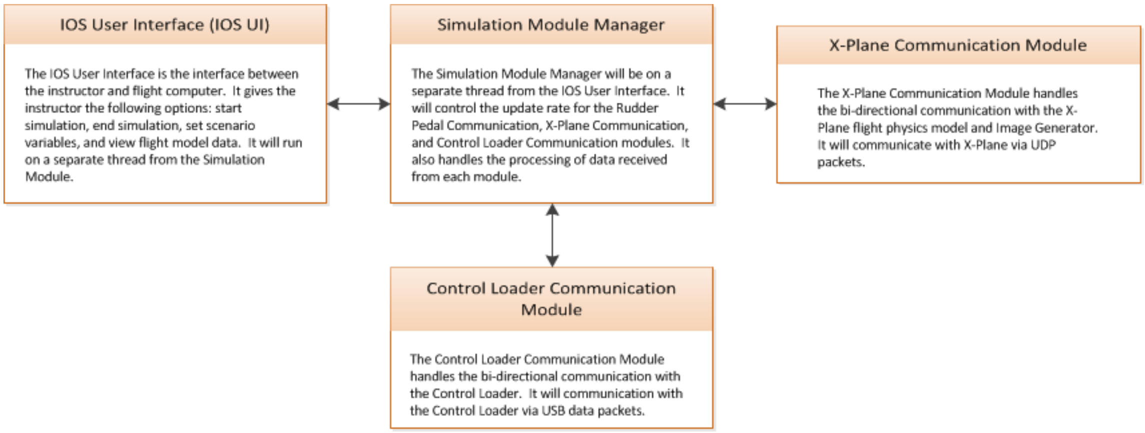


Figure 5.1.3 Embedded Software Flow

5.2 IOS

The IOS software was designed utilizing two programming paradigms. The front end (user interface portion) was designed with a heavily object oriented approach while the back end (simulation data processing) was designed in a non-object oriented fashion. **Figure 5.5.1** shows the interactions between the different systems and the IOS.

.



IOS User Interface (IOS UI)

The IOS User Interface is the interface between the instructor and flight computer. It gives the instructor the following options: start simulation, end simulation, set scenario variables, and view flight model data. It will run on a separate thread from the Simulation Module.

Simulation Module Manager

The Simulation Module Manager will be on a separate thread from the IOS User Interface. It will control the update rate for the Rudder Pedal Communication, X-Plane Communication, and Control Loader Communication modules. It also handles the processing of data received from each module.

X-Plane Communication Module

The X-Plane Communication Module handles the bi-directional communication with the X-Plane flight physics model and Image Generator. It will communicate with X-Plane via UDP packets.

Control Loader Communication Module

The Control Loader Communication Module handles the bi-directional communication with the Control Loader. It will communication with the Control Loader via USB data packets.

Figure 5.5.1 IOS Overview

The front end of the IOS is represented as the IOS User Interface and is responsible for providing a clean interface to the instructor in order to set up scenario values for a new simulation session as well to view the aircraft parameters in nearly real time during an active simulation session. The back end of the IOS software deals with handling data send to and from X-Plane and the Signal Processing Unit. The primary class that is responsible for this is the SimulationModuleManager. Due to the fact that the data processing must happen at a rate of between 50Hz-100Hz, the design was done to allow this with the least amount of overhead. To accomplish this goal, all methods are static and objects are used as little as possible. In some cases where objects are used, rather than instantiate a new instance with each iteration, a swap in place method was applied, which will just swap data in the existing object, this will reduce overhead and allow for the IOS to run more efficiently. Along with this approach, all values that are needed by multiple objects in the IOS are made available via a Memory Mapped File. This has two purposes; first it allows each user control to utilize reflection in order to get/set variables using their String representation. Second it allows for the future option to break the IOS into two separate applications while still using the same Memory Mapped File. This was done to give more flexibility when encountering issues during the development phase of the project. Much of the IOS design deals specifically with giving the flexibility to deal with issues that may arise, this outlook may seem a bit overzealous in the design phase, but will prove to be usefule during the development and testing phases.

5.3 Input Devices

The throttle's electrical design is composed of a rotary potentiometer with a digital reference. **Figure 5.1** shows the schematic for this device. **Figure 5.2** displays the schematic for the joystick's X and Y axis electrical circuits which are similar in nature to the throttle. The gear used for the throttle's pot allow for using most of the shaft rotation. However, the constraints of movement for the joystick potentiometers cause the rotation range to be much less than the full 270. When the shafts are rotated in the final design, the varying resistance will only reach about 100KOhms.

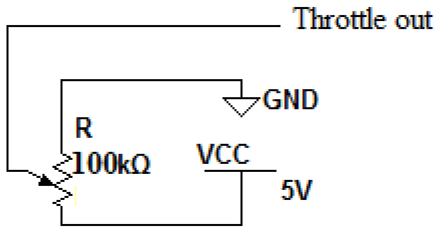


Figure 5.1 Throttle Design

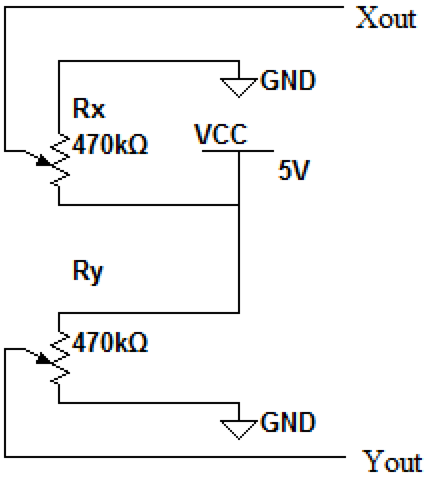


Figure 5.2 Joystick position tracking design

Figure 5.3 displays the schematic diagram for the joystick's force feedback system which is analog in nature. It needs a power supply to provide reference voltage to the load cells and also to power the amplifier IC.

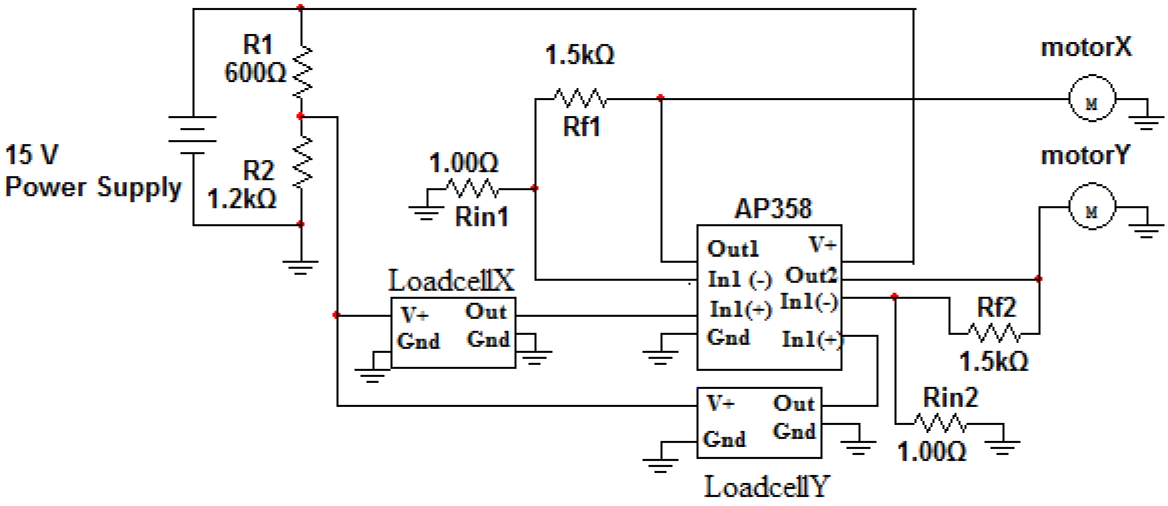


Figure 5.3 Joystick Force Feedback design

5.4 Power Distribution

The CL and Joystick force feedback systems need to be supplied with voltage. The control loader needs 22V to power the cascade amplifiers and the force feedback system needs 15V to power the load cells and amplifiers. Using voltage divider circuits the desired values were achieved. **Figure 5.4** demonstrates how this can be done; **Vc** corresponds to the voltage output going to the control loader, while **Vj** corresponds to the joystick voltage. The embedded electronics in the control loader will be given 5V by the flight computer's USB port.

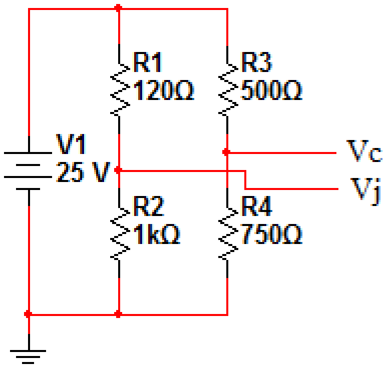


Figure 5.4 Power Distribution for Joystick and CL

6 Voltes Fly Assembly

The first part to be assembled will be the mechanical one. Since the pilot seat (part number **Adelfina High-Back Executive Chair**) is a regular office chair, the support legs have to be removed in order to join its base with the Motion Base. After the pilot seat is attached, the Joystick and the Throttle are placed using metal bars in front of the pilot. The Joystick will be placed at the center and the Throttle at the right hand side since its assumed that the mayority of the pilots are right handed. Since these input devices have to move with the pilot, the metal bars have to be attached (using bolts) to the Motion Base.

The second part of the assembly is the electronics. Every interface will be connected using their specific ports and the cable lengths need to be long enough to allow the system to move freely and avoiding accidents while performing. in addition, the projector needs to be placed at the right height to allow the pilot not to lose the line of sight. The audio system (**Corsair SP2200 Gaming Audio Series 2.1 PC Speaker System**) have two speakers and a subwoofer; each speaker is going to be attached to each side of the pilot seat and the subwoofer will be below it.

7 Prototype Testing

To ensure the quality and functionality of our prototype, the project must be fully tested. This phase of the project could take a lot of time since every testing need to be done thoroughly. The group needs to identify what would be the extreme or worst case scenarios for the project and it needs to be tested under these circumstances. For example, the group must see how the system's hardware responds to quick turns and unexpected moves within a wide angular range. In addition, the group has to test how much momentum is generated by some specific weights, so it can be determined which appropriate weight to apply in order to achieve a better system performance. When the software is tested, the group has to be looking for variables overflows scenarios and input's validations. After testing, the team should get the expected results based on the initial requirements and technical specifications.

7.1 Code Testing

After some research, the team decided to implement a well-known testing evolutionary approach, a Test Driven Development (TDD), which combines agile requirements and agile designs techniques. In addition, this approach is seen as a programming technique as well by some peers like Ron Jeffries who stated that "…the goal of TDD is to write clean code that works. I think that there is merit in both arguments, although I lean towards the specification view, but I leave it for you to decide" **[7.1].**

The TDD steps are as follow:

1. Quickly add a test.
2. Run the test.
3. Update functional code to make it pass the new test.
4. Run your test again.
5. If step 4 fails, update your functional code and retest.
6. If the test passed, retest.

The previous steps are represented and can be seen on **Figure 7.2.**

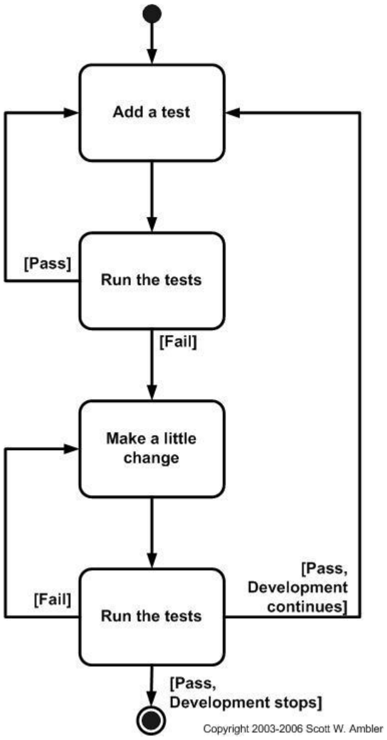

Add a test
[Pass]
Run the tests
[Fail]
Make a little change
[Fail]
Run the tests
[Pass, Development continues]
[Pass, Development stops]
Copyright 2003-2006 Scott W. Ambler

Figure 7.1. Testing Driven Development

There are many TDD helper applications on the web, all of them can be found as plug-ins for the several IDE's that currently exist on the market. For instance, **NUnit 2.x, MbUnit 2.4, MSTest, xUnit.net** are few examples of TDDs for the .Net platform. At the end of this stage, the prototype should come out of every test with a successful outcome.

7.2 Testing Procedures for Control Loader

The process of testing is really important for the application since will keep the team without last minute surprises. Even though the Control Loader is a subsystem of the entire system, the team needs to test individual subsystems inside the CL, so critical subparts like the ADC12, DAC12 modules, the JTAG interface, the USB converter and the CPU itself will be tested thoroughly.

7.2.1 DAC12 Testing

The first test will be performed on Multisim 11.0 where the team will run a simulation. The simulation will consist on plug in digital values on one end of the DAC12 and measure the output voltage value and check that they are in the correct range. Once the prototype is ready, the team conducted a test by following the steps below:

- Turn on the Control Loader.
- Put a digital value on the specific channel to be tested.
- Confirm that a correct voltage value is acquired at the output node.
- Verify that the voltage is in range.

7.2.2 ADC12 Testing

In addition, the group has to check the values received by the microcontroller, so tests on the ADC12 and JTAG modules will be done. Once the prototype is ready, the team will conduct a test by following the steps below:

- Turn on the Control Loader.
- Put an analog voltage value on the specific channel to be tested.
- Confirm that a correct digital value is acquired by the output node.
- Verify that the digital value is in range.

7.3 Testing Input Devices

When constructing the throttle and joystick components, testing will take place at every stage of the development process. These testing procedures will serve as “gates” which will open only once the current state of system has achieved the desired outcome. Once this occurs, the complexity of the system can be increased by adding the next component. It will start at the electrical component level, then move on to the circuit level. Once the circuits are analyzed and considered correct, their interaction with the control loader will addressed. After this level is completed, the circuits are integrated with the mechanical parts and tested once more. This however, does not mean that the different levels of testing will not have to be revisited. For example: it is a possibility that once the interface between the joystick and control loader is developed, it is found that the circuit of the joystick is missing a strategic electrical component. At this point, the whole sequence is started once again: the circuit is redesigned, tested independently, and integrated once more.

Let’s take a look at testing procedures at the circuit level, independently of the control loader. This can be done by making a breadboard circuit representation of the device. **Figure 7.3.1** is a snapshot of the throttle circuitry.

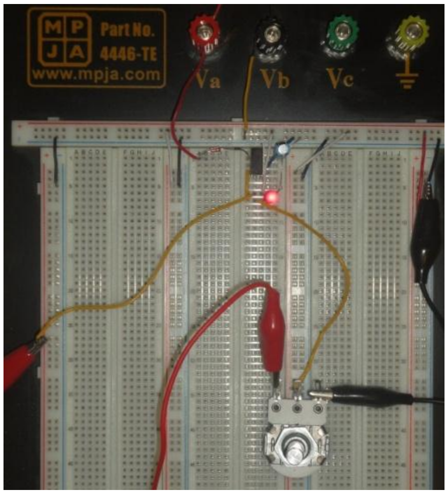


Figure 7.3.1 Breadboard arrangement

The red wire going into the leftmost potentiometer terminal is the 5V source, which comes from a simple regulator circuit above it. This regulator circuit takes the voltage from a cellphone charger and runs it through a voltage regulator chip, (LM7805C) and a diode which eliminate any AC remnants in the signal. The output voltage of the rotary potentiometer is applied to a light emitting diode so the effect of the rotation of the potentiometer's shaft can be observed. Because the joystick's directional tracking circuit is similar in design, the same setup can be used to test it. It is just a matter of adding a second potentiometer hooked up in series to another LED. To get actual data rather than just visual, a digital multi meter will be used to analyze such circuits. There are plenty of questions to keep in mind during the circuit level testing phase: is it necessary to add any diodes or resistor circuits to protect against possible random AC spikes? How stable is the current prototype? How does it respond to high frequency inputs? What would be the durability of the current prototype with regards to all of its electrical components?

In order to test the integration with the microcontroller, several steps need to be watched. In the hardware level, the device signals must be connected to the correct input pin. The analog to digital conversion is a process which can be tested separately from the control loader as well. In order to follow a good testing methodology, an additional MSP430 Launchpad board can be used to test these components independently from the main board within the control loader. A positive effect of this decision is that each group member can focus on separate functions of the microcontroller firmware independently. The MSP430 Launchpad development board will be used along with the breadboard circuits to test the interfacing. The board will be connected to a test computer for both power and data transmission. Once it is proved that the input devices achieve their initial goals and specifications, they can be integrated with the rest of the project. This will make it easier to implement the overall system once the testing is completed. The final code structure within the control loader will then become a combination of the separate functions finalized during testing. Once it is proved that the input devices achieve their initial goals and specifications, they can be integrated with the rest of the project. This modulating facilitates the group to find errors within the subparts of the firmware. The following guidelines will be used when testing the hardware peripherals:

- Manually change the potentiometers' resistance values by rotating the shafts.
- Check that the values received by the flight or test computer were changed by the correct amount, if this is not the case, then the issue must lie in the following areas:
    1. Hardware interfacing between devices and control loader.
    2. Analog to digital conversion.
    3. Firmware within MSP430 is not reading signals and sending them accordingly.
    4. USB driver issue, interfacing application issue.
- Attempt to solve any discrepancies in these areas and proceed to change resistance values again.

A typical scenario would be that the flight computer gets x and y values which make no sense. They could be of high magnitude, negative, etc. What can be done at this point is to separate the joystick and Analog to digital components and examine the output. Using a logic analyzer, it is possible to see the digital output of these parts. If the readings do not show the expected digital values, then the issue or mistake lies in the pin connections of the ADC or the component itself. On the other hand, if the correct values are shown then the problem is expected to lie within the launch pad. At this point, it is logical to check if the joystick output signals are being connected to the correct pins in the board or if the firmware being used for testing has the correct structure. Because embedded programming can be very fickle, a lot of errors can arise in this area. For example: pins must be set up in the beginning of the firmware code, identifying them as outputs or inputs. This means that one line of code can greatly affect the functionality. An effective way of dealing with this kind of issue is to use sample code available on the public domain. With this sample code, the tester can make sure that the language is being followed and every parameter the microcontroller needs is being provided.

When assembling the throttle mechanism, it is important to note that there is a constraint on the rotation range of the pot. This will lower the max resistance value available. When the constraint on the throttle rotation is being set, this limitation factor must be tested to decide what range of motion to allow. The same case goes with the Joystick's handle. Its rotations will not go the whole 90 degrees so the constraints must be tested and adjusted for optimal results. As explained in previous sections, wooden encasings will be used to protect these devices. They also have the function of providing the physical constraint needed. The top cover of this contraption will take part in the testing for the rotational constraints. Several of these lids might have to be made in order to find the optimal slit length and width (for the throttle's lever) and circular opening's diameter (for the joystick's handle).

In regards to the joystick's force feedback design, there is also plenty of testing to be done. The key components: load cells and motors each have their own testing phase before integration. A function generator will provide a great testing environment for the gear motor. First, the motor's functionality and effectiveness must be assured. This will be done as follows: a varying signal of 6-24V will be applied, attaching a crank to one of its shafts and physically trying to stop the rotation will allow the tester to see how strong the force feedback effect will be using this motor. The load cell will initially be attached to an oscilloscope to measure its excitation levels. The tester will physically push and pull on it to observe the output signal. After this is done, the amplifying circuit can be added to it. Once again, the tester measures the output through the oscilloscope. If the output range does not meet the initial expectations, resistor values might have to be changed. Worst case, additional circuitry might have to be implemented to get the desired output. At this point, integration with the motor can take place. Following the initial testing model, tests will continue to be conducted after every single mechanical component is added to the system. Eventually, the link from the crank to the load cell will be added; thus replacing the need to manually apply force to the load cell.

Once all the subsystems are integrated; preferably one month before completion, the input devices can be tested with regards to the X-plane environment. Sessions will be conducted to ensure that the feel of the input devices meet the initial expectations, and most of the adjustments will lie in the software side of the design.

7.4 IOS Testing

The IOS will be tested for functionality during the development and final sytem phase of the project life cycle. Testing that takes place during the development phase will be exclusive to the individual parts that make up the IOS software while the final system phase will be exclusive to testing the IOS interaction with each of the systems. The methology behind this is to test and integegrate each piece of the IOS software utilizing a bottom up approach which will ensure each sub part of the software works correctly, therefore reducing the amount of testing during the final system phase. This will decrease the chance of bugs in the code and help reduce the time involved to test the overall system.

Development Testing Phase

Development phase testing will take place as the IOS software is being written. The methology used will be to build each sub-piece of the IOS, test it, and then integrate it. The design of the IOS is what allows for this testing to take place and was done in part to reduce the amount of time needed to ensure each sub piece works correctly. The majority of testing will be related to the parts of the IOS that will communicate with other systems of the simulator. Note that testing of each individual class is not necessary since most classes are built from other classes, therefore once the basic functionality is tested, it is safe to move on.

In the Utilities project the first piece to be tested will be the UDPConnection class. In order to test the functionality of this, a “dummy” program will be written that will use the UDPConnection class to send and receive data packets from X-Plane and will display the packets count to the screen in order to notify that there is an active connection. It doesn’t matter what data is sent or received, but rather that data was sent and received. The purpose is to prove that the class is able to send and receive data via UDP, it is up the implementing class to decide the purpose. If this communication is possible, the UDPConnection class will be deemed complete. Next the USBConnection class will be developed and tested. In order to do this, a “dummy” program will be written that will display the position values of a COTS Joystick to the screen. If the values are able to be displayed, then it will be deemed that the USBConnection class is able to communicate via USB. Note that the USBConnection class is for generic USB communication and it is up to the implemented class to designate the device to communicate with. Once these two pieces have been tested and confirmed, they will be both added into the Utilities project.

The next major piece of the IOS softare to be developed, testing and integrated is the Datapool class. This class is a very important piece of software, because it provides the interface into mapped memory. After it has been developed, it will be tested by using a "dummy" program that will consist of a produce and consumer. The producer and consumer will be two different threads and the producer will set the variable in memory, while the consumer will read the variable and display it to screen. The producer will iterate from 1-100 and set those values to memory at an interval of one per second. So for time equals 0, the memory value will be 0 and at time equals 50, the memory value will be 50. If this is reflected by the consumer, then it will be deemed valid and the Datapool class will be integrated into the IOS software.

Once the Datapool has been verified next the UserInterfaceControls will be developed and tested. The same methology used to test the Datapool class along with the same "dummy" application. There will be an output window which will display the value at a specified memory location. The user controls within this project deal with either displaying data from memory or setting data in memory. As each control is developed, they will be first implemented in the "dummy" application to verify the functionalit. This will ensure they are all able to read and write to memory accordingly. After all the user controls have been tested individually a test will be run to test them running in parallel with each other. This is specific to the ParameterDisplay and the user controls that implement the class. Each ParameterDisplay user control has it's own Timer object that will run at a defined rate. In ordre to test the performance in an absolute worse case scenario, 200 ParameterDisplay objects will be instantiated in the "dummy" application with a defined rate of 120Hz. Then CPU usage will be recorded and if it is deemed within an acceptable range, finally all the user controls will be integrated into the IOS software.

Final System Testing Phase

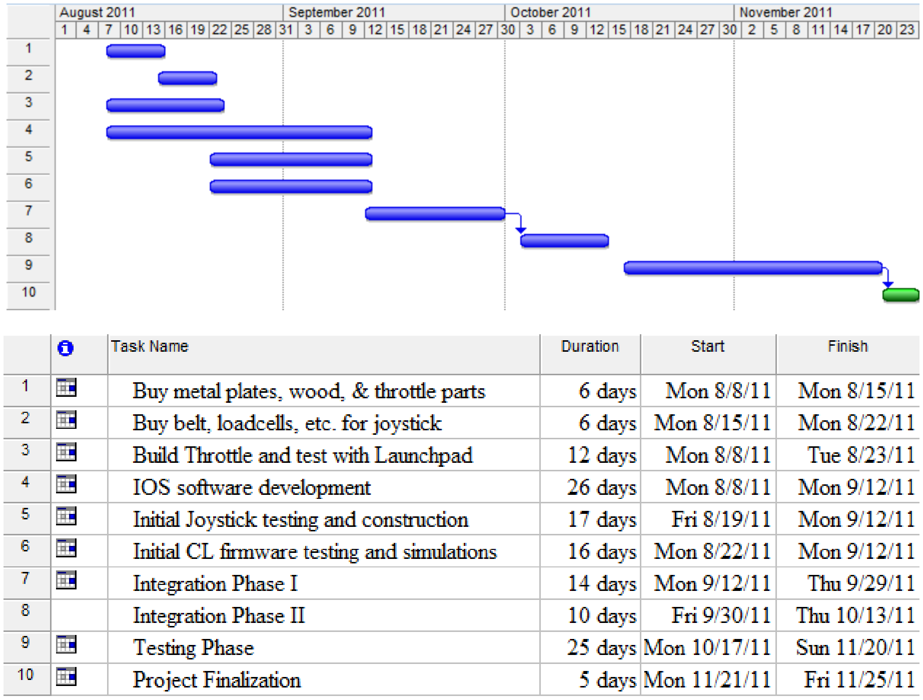
Once the IOS and all other systems have been developed, the Final System Testing Phase will begin. This will test each component connected to the IOS before integrating it into the system. The first system to be tested will be the Rudder Pedals. Since they are a COTS item, the only testing needed is to verify they are able to work with X-Plane. This only requires X-Plane to be launched manually, the Rudder Pedals plugged into a USB port, then view the output page of X-Plane to verify the Rudder Pedals position is moving. Once this has been verified, it will be accepted to the Final System. The next piece to test is the communication between the IOS and X-Plane. To do this, the instructor will attempt to set various scenario values and start the simulation. If X-Plane responds to the scenario settings, it will be deemed passing. Next, communication from X-Plane will be verified. X-Plane outputs data via UDP at a user defined frame rate, the IOS will attempt to read the input data, then it will be displayed in the FlightParametersPage, if the values for Roll and Pitch are being updated, then it will be deemed passing. Finally communication with the Signal Processing Unit will be tested.

First receive communication will be tested, by using the FlightParametersPage to verify the values sent from the Signal Processing Unit are valid. If they are valid, then send communication will be tested. This is a bit trickier since the value are received by the Signal Processing Unit. This will be tested as a joint effort between the IOS and Signal Processing Units debugger. If the debugger displays the values sent from the IOS correctly, it is deemed passing.

8 Administrative Content

8.1 Milestones

In the construction and development phase of the project, the process will take a test, build, and integrate methodology as can be seen by the Milestone Gantt chart in **Figure 8.1**. The beggining consists of a lot a parallel work, and as the group moves closer to the due date of the project all the systems are put together . The testing phase was given almost a month because of the nature of this project. The group has decided to finish the project at least a week before the presentations in december so that there is more time for personal occupations (final exams, graduation preparations, etc.) The goal is to finish the testing phase as soon as possible so that there is additional time to design the website of the dedicated to this project, make a second edition of this document, and prepare the powerpoint and oral presentation.



	ℹ	Task Name	Duration	Start	Finish
1	▦	Buy metal plates, wood, & throttle parts	6 days	Mon 8/8/11	Mon 8/15/11
2	▦	Buy belt, loadcells, etc. for joystick	6 days	Mon 8/15/11	Mon 8/22/11
3	▦	Build Throttle and test with Launchpad	12 days	Mon 8/8/11	Tue 8/23/11
4	▦	IOS software development	26 days	Mon 8/8/11	Mon 9/12/11
5	▦	Initial Joystick testing and construction	17 days	Fri 8/19/11	Mon 9/12/11
6	▦	Initial CL firmware testing and simulations	16 days	Mon 8/22/11	Mon 9/12/11
7	▦	Integration Phase I	14 days	Mon 9/12/11	Thu 9/29/11
8		Integration Phase II	10 days	Fri 9/30/11	Thu 10/13/11
9	▦	Testing Phase	25 days	Mon 10/17/11	Sun 11/20/11
10	▦	Project Finalization	5 days	Mon 11/21/11	Fri 11/25/11

Figure 8.1 Milestone for Second Phase

9 Future Improvements

Flight simulation is an area with lots of possible features to be implemented. The scalability and flexibility of this project was clear to the group once existing similar projects were studied. Every project researched had something unique to it: the technology used for motion, hardware components used, aircraft supported for simulation, the number of hardware components implemented. Obviously, the more features available to the user, the more realistic the experience becomes. Because of the scope and timing given for the completion of this project, many of the initial ideas were dropped or postponed until the main components were developed.

One of the first ideas the group thought about was G-Force. This system would use more than 2 degrees of freedom to create a “rollercoaster feel”. 2 Diagonal belts would be needed in the pilot seat to protect the pilot as he/she is launched forward by this effect. With more degrees of freedom, Sway, Yaw, and Heave can also be implemented. There were several reasons why this system was not taken seriously; first off, a motion base with more than 2 degrees of freedom was needed to simulate the G-Force algorithms. The motion base to create the effect would have been outside our initial budget proposal and the algorithms to control the platform with these additional features seemed too difficult. However; with more time and money in our hands, it makes for a great upgrade. Other similar ideas included seat vibrations with multiple levels of intensity, and a cockpit design which would encapsulate the player. The seat vibrations could be triggered after a certain speed has been achieved, or when the airplane is moving on the ground. The cockpit design could be modeled after the Cessna which the group has focused the simulation on.

After having learned how to read values like pitch and roll from X-plane, one can easily expand the control loader to take in more interesting parameters. Height, speed, acceleration, motor information, amount of gas available, and light indicators from the virtual cockpit can be read in by our software interface and dispatched to new hardware components. Gauges, flip switches, LEDs, displays and knobs can be assigned to each of these parameters. At this point, the system would probably need more hardware components to handle the quantity of bidirectional information being transferred. These extensions seem relatively easy to implement once the main joystick, throttle, and motion base design has been completed. Another interesting addition would be the support for a wide range of aircraft models. Commercial planes would be great for training purposes. Also, fighter jets could be added and possibly a simulation software which supports combat would be researched. The algorithms to move the motion base would change depending on the type of airplane being flown. The Throttle could be redesigned to have 4 levers so that the support for more than one engine can be achieved. This will give the pilot more flexibility as to what kind of experience or training he/she would like to receive by Voltes fly. The G-force and seat vibration systems would become more complex as a cause of this change.

A more original component which is not used in many simulators is the incorporation of head tracking. It would be based on a Wii remote or a hacked Kinect device which would rotate the pilot’s view as the pilot turns his/her head around in any direction. With so many different options for expansion, the group is excited with the possibility of finishing early so one of these components can be added before the project deadline.

10 Summary

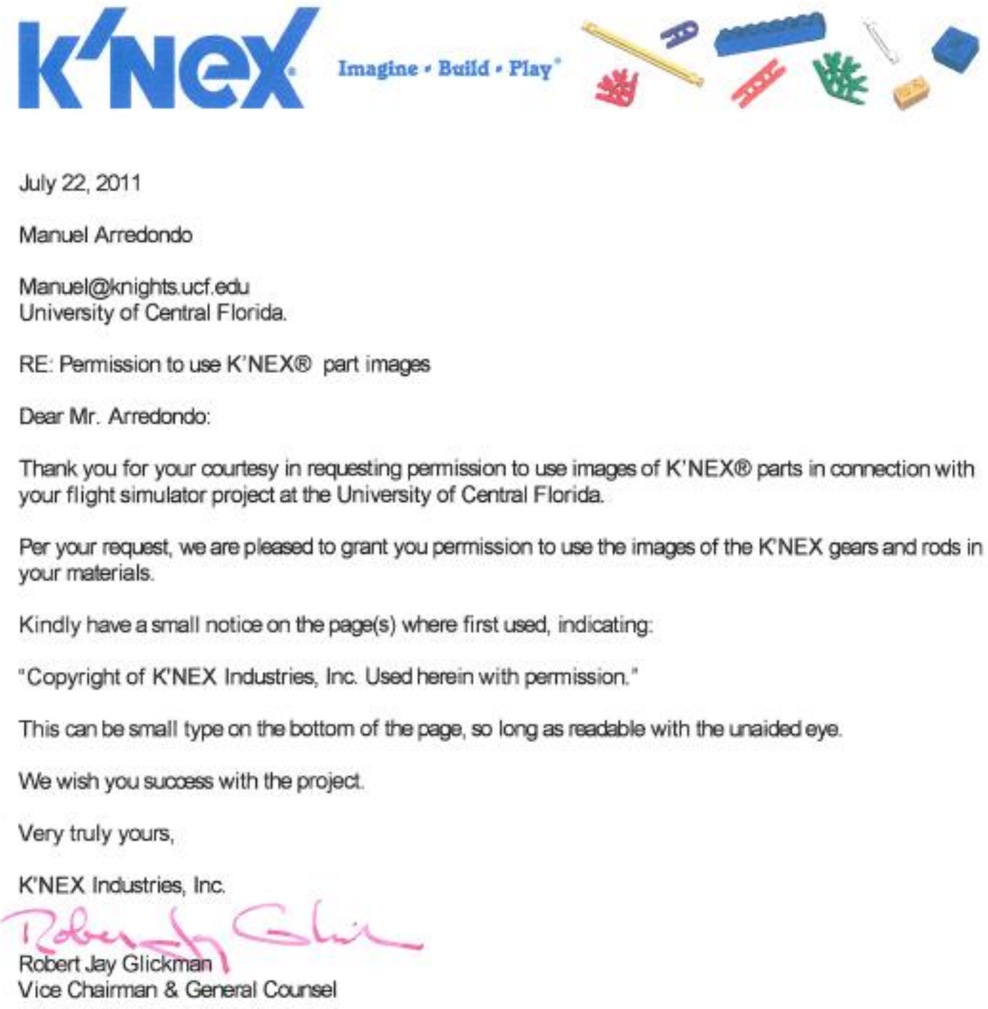
Voltes fly is an interactive flight simulator which communicates with the Pilot in several ways. The team is responsible for the design and construction of a joystick, throttle, and all the software and hardware required for these and a motion platform to communicate with the main CPU or flight computer. The motion platform was leased to the group by Servos Inc. Rudder Pedals will also be included in the final prototype, but these are COTS and not designed by any team members. The flight simulation software to be used is X-plane, which will communicate with an instructor operating station. The main goal of this project is to have a seating platform that tilts in synchronization with the airplane within the X-plane simulation.

11 Appendixes

11.1 Table of Figures.

11.2 Table of Tables

13 Copyright Permissions



K'NEX Imagine • Build • Play

July 22, 2011

Manuel Arredondo

Manuel@knights.ucf.edu
University of Central Florida.

RE: Permission to use K'NEX® part images

Dear Mr. Arredondo:

Thank you for your courtesy in requesting permission to use images of K'NEX® parts in connection with your flight simulator project at the University of Central Florida.

Per your request, we are pleased to grant you permission to use the images of the K'NEX gears and rods in your materials.

Kindly have a small notice on the page(s) where first used, indicating:

"Copyright of K'NEX Industries, Inc. Used herein with permission."

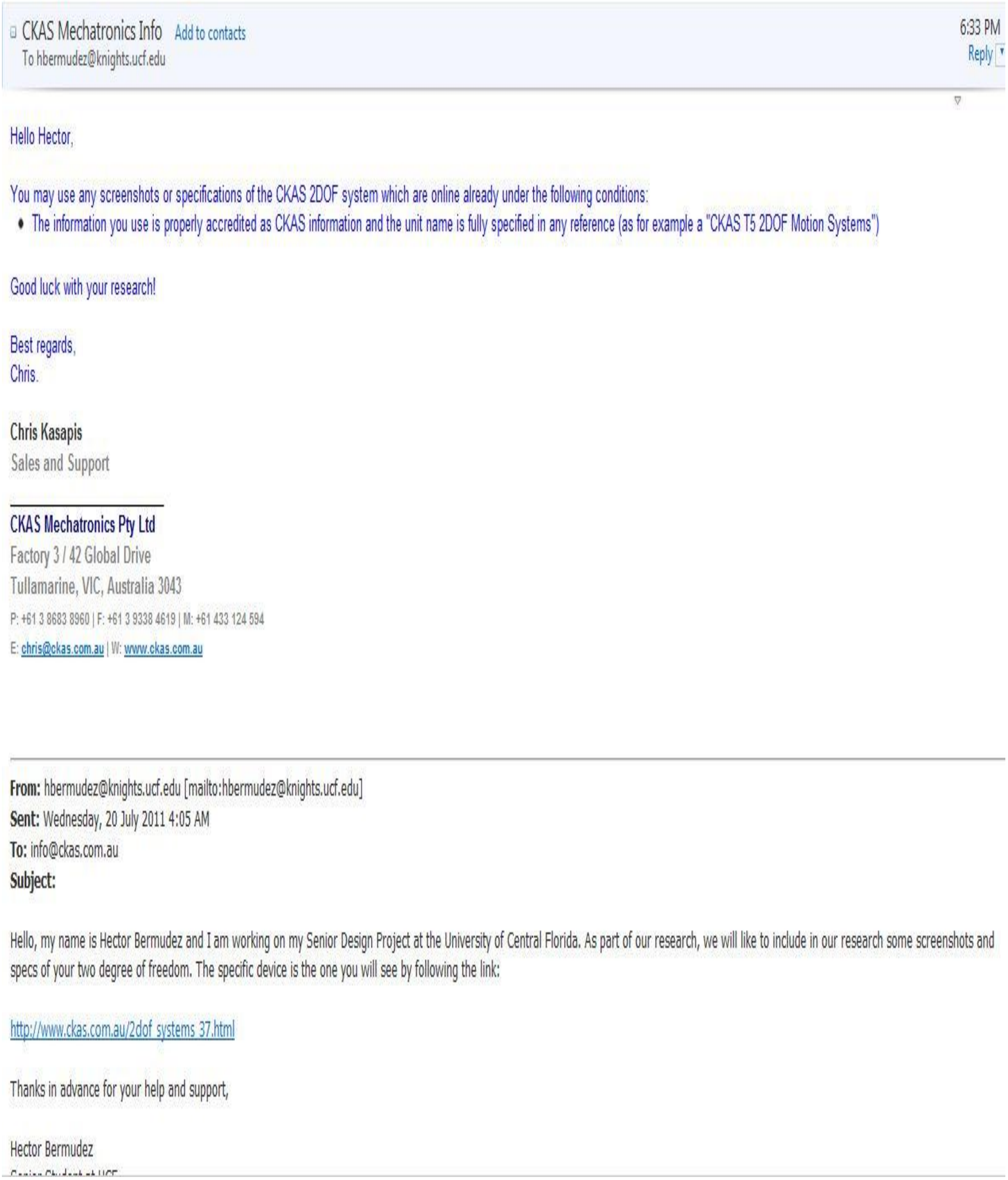
This can be small type on the bottom of the page, so long as readable with the unaided eye.

We wish you success with the project.

Very truly yours,

K'NEX Industries, Inc.

Robert Jay Glickman
Vice Chairman & General Counsel



□ CKAS Mechatronics Info Add to contacts
To hbermudez@knights.ucf.edu
6:33 PM
Reply ▾

Hello Hector,

You may use any screenshots or specifications of the CKAS 2DOF system which are online already under the following conditions:

- The information you use is properly accredited as CKAS information and the unit name is fully specified in any reference (as for example a "CKAS T5 2DOF Motion Systems")

Good luck with your research!

Best regards,
Chris.

Chris Kasapis
Sales and Support

CKAS Mechatronics Pty Ltd
Factory 3 / 42 Global Drive
Tullamarine, VIC, Australia 3043
P: +61 3 8683 8960 | F: +61 3 9338 4619 | M: +61 433 124 594
E: chris@ckas.com.au | W: www.ckas.com.au

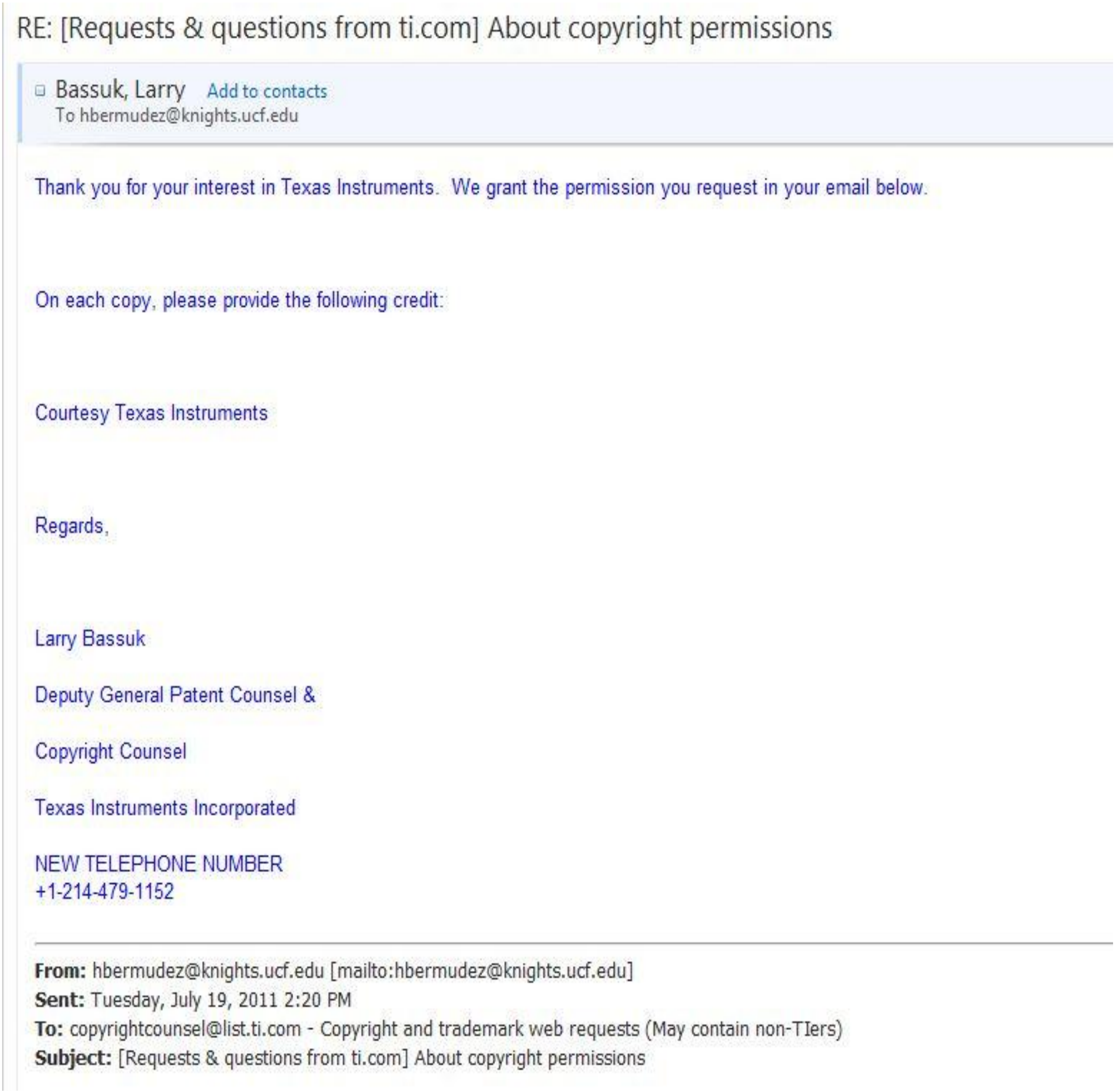
From: hbermudez@knights.ucf.edu [mailto:hbermudez@knights.ucf.edu]
Sent: Wednesday, 20 July 2011 4:05 AM
To: info@ckas.com.au
Subject:

Hello, my name is Hector Bermudez and I am working on my Senior Design Project at the University of Central Florida. As part of our research, we will like to include in our research some screenshots and specs of your two degree of freedom. The specific device is the one you will see by following the link:

http://www.ckas.com.au/2dof_systems_37.html

Thanks in advance for your help and support,

Hector Bermudez
Senior Student at UCF



RE: [Requests & questions from ti.com] About copyright permissions

Bassuk, Larry
To hbermudez@knights.ucf.edu

Thank you for your interest in Texas Instruments. We grant the permission you request in your email below.

On each copy, please provide the following credit:

Courtesy Texas Instruments

Regards,

Larry Bassuk

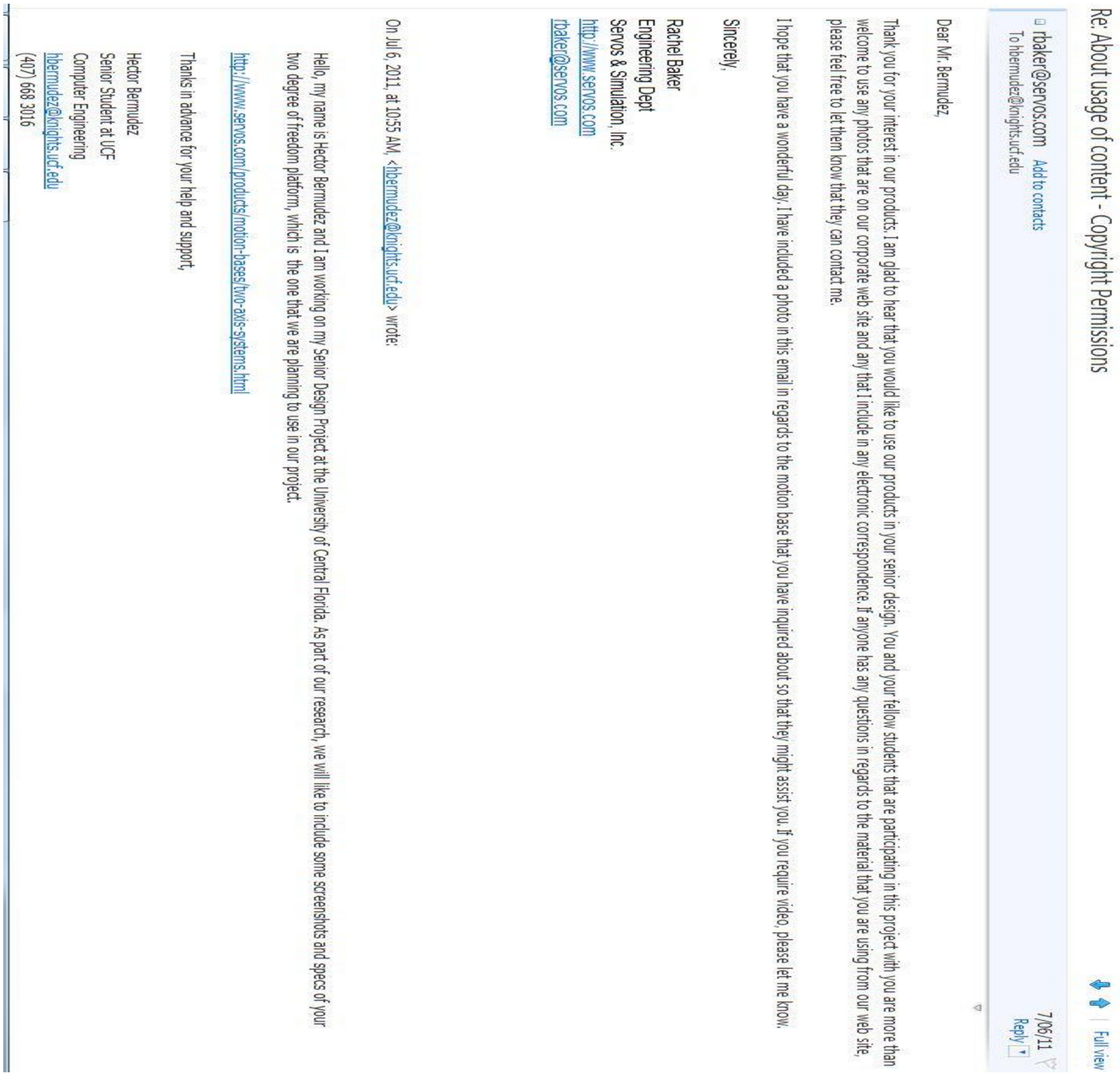
Deputy General Patent Counsel &

Copyright Counsel

Texas Instruments Incorporated

NEW TELEPHONE NUMBER
+1-214-479-1152

From: hbermudez@knights.ucf.edu [mailto:hbermudez@knights.ucf.edu]
Sent: Tuesday, July 19, 2011 2:20 PM
To: copyrightcounsel@list.ti.com - Copyright and trademark web requests (May contain non-TIers)
Subject: [Requests & questions from ti.com] About copyright permissions



Re: About usage of content - Copyright Permissions

rbaker@servos.com 7/06/11

To hbermudez@knights.ucf.edu

Dear Mr. Bermudez,

Thank you for your interest in our products. I am glad to hear that you would like to use our products in your senior design. You and your fellow students that are participating in this project with you are more than welcome to use any photos that are on our corporate web site and any that I include in any electronic correspondence. If anyone has any questions in regards to the material that you are using from our web site, please feel free to let them know that they can contact me.

I hope that you have a wonderful day. I have included a photo in this email in regards to the motion base that you have inquired about so that they might assist you. If you require video, please let me know.

Sincerely,

Rachel Baker
Engineering Dept
Servos & Simulation, Inc.
http://www.servos.com
rbaker@servos.com

On Jul 6, 2011, at 10:55 AM, <hbermudez@knights.ucf.edu> wrote:

Hello, my name is Hector Bermudez and I am working on my Senior Design Project at the University of Central Florida. As part of our research, we will like to include some screenshots and specs of your two degree of freedom platform, which is the one that we are planing to use in our project.

http://www.servos.com/products/motion-bases/two-axis-systems.html

Thanks in advance for your help and support,

Hector Bermudez
Senior Student at UCF
Computer Engineering
hbermudez@knights.ucf.edu
(407) 668 3016

14 References

[2.6.1] http://www.servos.com/products/motion-bases/two-axis-systems.html

[3.5.1] http://focus.ti.com/docs/toolsw/folders/print/iar-kickstart.html

[3.5.2] http://www.ti.com/CCStudio

[3.5.3] http://focus.ti.com/docs/toolsw/folders/print/grace.html

[3.9.1] http://www.ti.com/launchpad

[3.10.1] http://www.flightsim.com/main/howto/throt.htm

[3.10.2] http://www.cesko.host.sk/IgorPlugUSB/IgorPlug-USB%20(AVR)_eng.htm

[3.10.3] http://www.howstuffworks.com/joystick.htm

[3.10.4] Peter J.Mikan. “Joystick Control having Optical Sensors”. March 15, 1988. Patent number: 4,731,530.

[3.10.5] .PDF edition of Carl David Todd (ed), "The Potentiometer Handbook",McGraw Hill, New York 1975 ISBN 0-07-006690-6

[3.10.6] http://sound.westhost.com/pots.htm#taper

[3.10.7] http://pinouts.ru/Inputs/GameportPC_pinout.shtml

[3.10.8] Elaine Y.Chen, Bine an, Timothy R. Osbourne, Paul Dilascia, Matthew Coill. “Force Feedback Joystick with Digital Signal Processor controlled by Host Processor”. April 21, 1998. Patent number: 5,742,278.

[3.10.9] Louis B Rosenberg, Adam C. Braun, Mike D. Levin. “Method and Apparatus for Controlling Force Feedback Interface Systems Utilizing a Host Computer”. March 31, 1998. Patent number: 5,734,373.

[3.11.1] http://www.usb.org/developers/docs

[3.11.2] http://msdn.microsoft.com/en-us/windows/hardware/gg487428

[3.11.3] http://www.beyondlogic.org/serial/serial.htm

[3.13.1] Kernighan; Dennis M. Ritchie (March 1988). The C Programming Language 2nd ed.Englewood Cliffs, NJ: Prentice Hall. ISBN 0-13-110362-8. http://cm.bell-labs.com/cm/cs/cbook/.

[3.14.1] http://www.youtube.com/watch?v=TBD9Juqx7PI

[3.14.2] http://www.eecs.ucf.edu/seniordesign/fa2009sp2010/g11/documents.php

[4.7.1] http://www.800loadcell.com

[4.7.2] http://www.performanceaudio.com/buy/Radial_Engineering/R15DC_US/10450

[4.7.3] http://www.timingbeltpulley.com/