Development of a Fault Tolerant Flight Control System

By

Cary Benjamin Feldstein
B.Sc., University of Victoria, 2002

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Supervisor: Dr. J. C. Muzio

ABSTRACT

This Thesis discusses the design and development of a fault tolerant flight control system. The requirements of safety critical systems, reliable systems, fault tolerant systems, Avionics and Embedded systems are considered for this project. Byzantine Resilience and Common Mode Faults are discussed but not considered for this work. The fault tolerant system designed for this work was set up as a triple modular redundant system to tolerate the existence of one fault within the system. The system was implemented with the PC/104 embedded platform. Microsoft Flight Simulator was used to generate input data and to demonstrate successful operation by showing a flight under the control by the flight control system. The end results show that a fault tolerant system can be developed and it can successfully tolerate one fault while the system is in operation.

Table of Contents

List of Figures

Acknowledgements

I would first like to thank Dr. Jon Muzio for supervising me through this Master's program and for all his help and support over the last three years. Without his support this work would not have been completed.

Next, I would like to thank a group of people who were able to help me complete certain steps of this work when I was stuck and unable to make progress. These people are Brian Douglas, Paul Stead, David Manning, Andrew Wyeth and Anthony Howe. Brian Douglas was a huge help with setting up the PC/104 equipment. He worked with me for a few hours when I needed specific wiring made up for my experiment and he offered advise on how to best setup the hardware to run. I could not have set up the fault tolerant flight controller without Brian's help and the resources available in his workshop. Paul Stead and David Manning were both able to help me with specific software problems that had halted my progress. David was able to help me reconfigure the Linux operating system to work properly on the PC/104 boards. Paul was able to help me when I had some programming problems with C and Linux Sockets that stopped my work from proceeding. Anthony Howe and Andrew Wyeth are both good friends who were interested in my work. Anthony helped me configure the routing tables on the PC/104 boards. Andrew helped me make design implementation decisions by talking through the options with me and we would weigh the advantages and disadvantages of each.

I would also like to thank the members of the Digital Systems Design group for their help and support over the last two years. In particular, Jason Hannula who progressed through his degree at about the same pace as me. By trying to keep up with Jason I was able to stay on schedule and complete this research and degree in under two years.

I would also like to thank my friends and family for all their help and support through the last two years. My future wife, Kelly Gardin, who has sacrificed two years of her teaching career to move to Victoria to be with me and help me stay focused. The main

reason I was so motivated to finish this degree in under two years was because of the promise I made to Kelly that we would not be in Victoria over the expected two years so that she could return to her teaching position in Vancouver. This also motivated me to stay as healthy as I could over these two years so I did not fall behind schedule. I would like to thank my parents for all their all their support with my never ending education. Thanks to Steven Shelford for his help and guidance through each step of the way. Finally, thanks to all of the Viper's UVic Intramural Ice Hockey Team for helping me keep sane each week with a good-humoured game of hockey.

Finally, thanks to anyone else who I might have inadvertently missed!

In Memory of

my Saba,

Morris Feldstein
(1913-2001)

Chapter 1

Introduction

This thesis explores the design and implementation of a fault tolerant flight control system. The motive for doing this was to combine the author's passions in the fields of computer science and aviation. It was triggered when the author experienced endless frustration while researching the topics of modern flight control systems. The literature [1-10] lacked sufficient implementation details to allow even a scaled-down version of the same systems to be implemented. Obviously this lack of information is intended to keep intellectual property proprietary and gain a strategic advantage for each company over its competition.

The concept of having a computer based system controlling aircraft has been in practice for over forty years [1, 6]. This type of system is considered a safety critical real time system. It is a system that will not endanger human life or the environment [11]. These systems are usually developed by large engineering companies with apparently endless resources because of the complexity and safety issues inherent in this type of computer application. The mission requirements of a safety critical system bring with them a development requirement that the system must be built right the first time no matter what the cost; therefore if a system crashes, any catastrophic system failure would have a much higher cost (eg. people may be killed) than a delay in development. We can not afford to send customers beta-releases or do-it-yourself versions of these systems.

One common application for safety critical systems is to commercial aircraft. The practice of taking responsibilities away from the pilot and trusting a safety critical system is, for better or for worse, becoming more common in industry. Little has been published in this area of research. These systems are proprietary systems, thus only limited details are available to the public. On the one hand, it can be argued this is done solely to keep the

proprietary technology away from competition but it is the opinion of the author that this is also done so that the general public cannot get sufficient information about the systems to verify that the systems actually do what they promise to do. Although we do not question whether or not these systems are safe, there is always the possibility that they could be analyzed by independent researchers to find ways to make them even safer.

Since insufficient details are available to employ a copy of one of the systems in aviation use we have chosen to design and implement our own system in order to make an attempt to see if it is even possible to do some of the things the industry claims it can do. To do this we decided to use component off-the-shelf technology, hardware available in the existing market instead of a custom product designed for this purpose.

This work shows that it is possible to design and implement a scaled-down fault-tolerant flight control system within academia as a research project. The value of having this system built is significant as further designs can be tried and tested by reconfiguring this system instead of beginning from scratch.

The nature of flight control brings with it a requirement that the system must work quickly enough to react to events which happen in the real world. This requirement must be met for the system to succeed in its task. This type of control system is referred to as a real time system. If a flight control system cannot make its reactions frequently and quickly then the system would not successfully meet this real time requirement and it would have great difficulties controlling an aircraft in flight. The focus of this work is not to meet formal real time system requirements as we are concentrating on making such a system fault tolerant. It is possible that the end result of this work will be fault tolerant but the system will not function quickly enough to meet the real time requirements.

The next chapter discusses the background material and context for this work. The terms: safety critical, fault-tolerant and reliability are defined. As well, there is a discussion on current challenges in designing these systems. Then, the term avionics is defined followed by a discussion on the particular requirements of the aviation field as it relates to digital systems and this work.

Chapter three presents an architectural design for a fault tolerant flight control system followed by a discussion of the design. Decisions about how to achieve a high level of reliability and how to define the scope and limitations of this work are discussed. Criteria for selecting the hardware platform to be used are also discussed. A test environment needed to be defined in order to operate the flight control system without purchasing an aircraft and installing it on board. Chapter three also discusses the decisions made for the testing environment. Microsoft Flight Simulator was used to supply the fault tolerant flight control system with pseudo real time flight data and to visually demonstrate the correct operation of the system.

Chapter Four discusses the implementation and results of the fault tolerant flight control system. During the implementation process, detailed and low level designs had to be defined along with many implementation decisions. For example, selecting the hardware to use for this work seemed simple enough, yet, as we committed ourselves to one product, we discovered several additional and unanticipated tasks in order to move on. Next, the chapter discusses the implementation of a non-redundant flight controller and the addition of redundancy to achieve the final desired system. The chapter concludes by discussing the results of the fully operating fault tolerant flight control system.

Finally, Chapter five discusses the conclusions that were reached from this work and some future directions this work can take.

Chapter 2

Background

This Chapter is an introduction to the background topics of this thesis. These areas are Safety Critical Systems and Avionics.

2.1 Safety Critical Systems

When considering the safety of a system we are concerned that "... it will not endanger human life or the environment" [11 pg. 2]. In order to design a safety critical system, one must incorporate design aspects of reliable systems and include application-specific decisions to cause the system to fail in a safe manner [3]. This section discusses:

- Reliable Systems;
- Fault Tolerance;
- Byzantine Resilience;
- and Common Mode Faults.

These topics are discussed because they are all relevant when considering the design of a real time safety critical system.

2.1.1 Reliable Systems and Fault Tolerance

Webster's New World Dictionary defines the word 'reliable' as "that can be relied on; dependable" [12]. Given that reliability has a broad definition in the English language, it is a good starting point for considering the term in the context of fault tolerant computing. This is because the term can mean many different things and can have a different meaning in

each field or context. When saying a safety critical system is reliable, we must be specific in understanding what this means. For example, consider a student who always sleeps too late and misses their morning class every day. To this student's professor and peers they appear to be an unreliable person. But this one personality trait does not mean they are or are not a reliable person in other aspects of life. Although they exhibit unreliability in school attendance they can always be found at home, asleep and reachable by phone at the time they should be in their morning class. Although this behaviour is unexpected to most, for those who know this person, the behaviour is very reliable as it is well-defined and consistent.

In fault-tolerant computing it is imperative that the system or a component behave in a defined and expected manner, regardless of what the behaviour is, as the design of the system can then be made in such as way as to handle the expected behaviour. It is more difficult to handle unexpected behaviours. In fault tolerant computing we need a better definition for reliability which would include what a system is capable of being reliable for as it is unrealistic to expect anything or anyone to be universally reliable. As shown in the example with the student, from another point of view, something considered reliable can appear unreliable.

Most definitions for the reliability of a system, or a component, state, that it must operate as expected over a given period of time [3, 9-11, 13]. Some definitions include a specific way to quantify reliability such as Daniel Siewiorek and Robert Swarz's definition: "The reliability of a system as a function of time, R(t), is the conditional probability that the system has survived the interval [0,t], given that the system was operational at time t=0" [9 pg. 4]. Other definitions are clearer to understand such as Nancy Leveson's definition: "Reliability is the probability that a piece of equipment or component will perform its intended function satisfactorily for a prescribed time and under stipulated environmental conditions" [10 pg. 172].

For the purpose of this thesis we have chosen to define reliability as the probability that a system will operate correctly and continuously for a defined period of time while operating under a defined set of conditions. It should be noted that this definition is very

similar to Neil Storey's definition: "Reliability is the probability of a component, or system, functioning correctly over a given period of time under a given set of operating conditions" [11 pg. 20]. These two definitions differ as our definition only addresses systems. To keep this work manageable we have not addressed the reliability of each component.

A reliable system is one which has a high level of reliability. Such a system must be clearly defined as to how it is expected to operate under defined conditions. There are two basic approaches to achieving reliability in a system. One approach is to prevent faults from occurring in the first place. This is called fault-avoidance and although it is a logical approach, it is not commonly used due to its high costs and incomplete solution to the problem [9]. The second approach is to design the system to include redundancy so that if a component within the system should create a fault, the system would detect or tolerate that fault. Detecting a fault would allow the system to be aware of the fact that something has gone wrong and then a defined reaction to the fault could take place. For example, when the system controlling the traffic lights at a street intersection suffers a fault it would be best for the lights to flash red instead of turning off completely, or worse, all turning green at the same time. Tolerating a fault goes a step further and allows the system to still operate correctly even though a component is faulty. Both of these systems are called redundant systems and they are commonly used to achieve system reliability.

The methods used to achieve system reliability are summarized in figure 2-1. This taxonomy of failure response strategies addresses the methods that can be used to achieve reliability in a system. With an overall goal of achieving system reliability, the figure shows that this can be accomplished using non-redundant systems or redundant systems. The available options when the system is non-redundant are limited to fault avoidance. When the system does include redundancy, reliability can be achieved by using the strategies of fault detection or fault tolerant systems. This last strategy has various different ways to tolerate faults which fall under the categories of masking redundancy and dynamic redundancy.

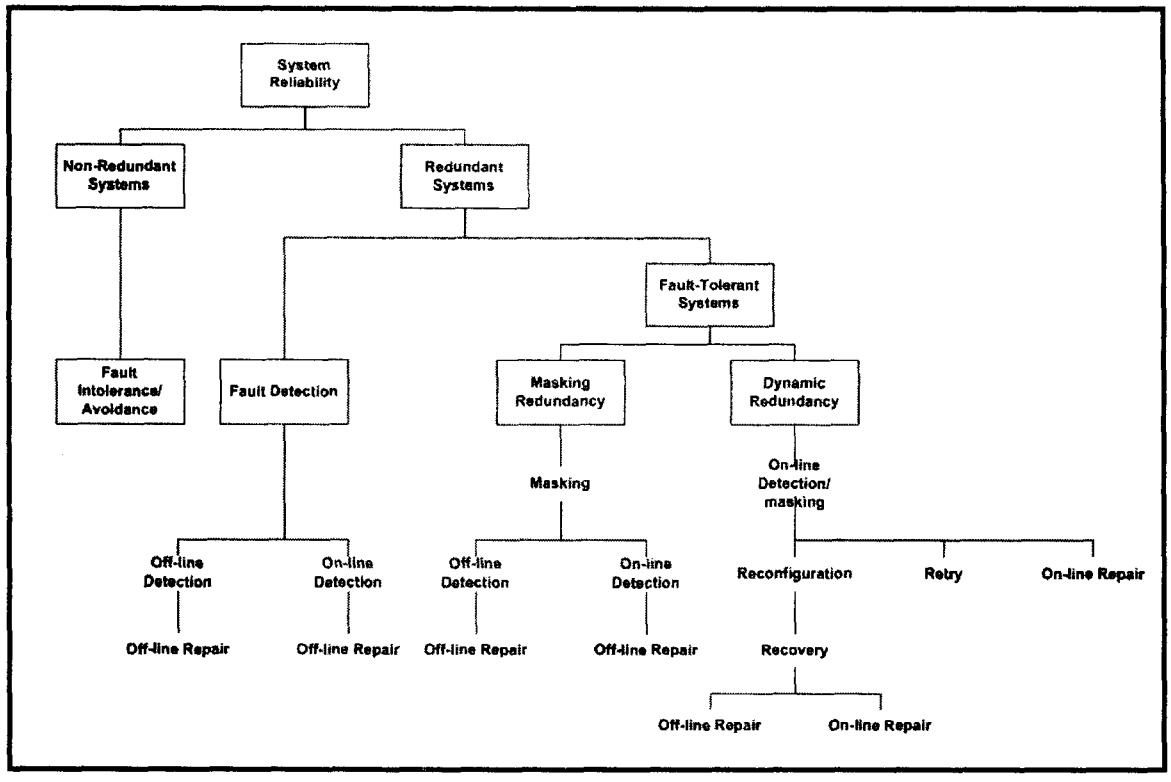


Figure 2-1 Taxonomy of system-failure response strategies [9]

Before discussing the details of these system reliability strategies it is important to discuss why they would need to be used. Introducing any of these techniques into a system would increase the cost of developing and building the system. In the world's economy, businesses and consumers usually choose to purchase the cheapest product that meets their needs. In general there are four system applications which require a high level of reliability, these applications being: long life, critical computation, maintenance postponement and high availability [14].

Figure 2-2 shows that there is a balance between including a number of reliability and maintainability features in a system in order to meet the requirements of the application and to minimize the overall cost of the system. In order to make a decision on how many reliability features to include in a system, an organization must weigh between high acquisition costs and high service costs to reach a minimum cost of ownership. A system with very few reliability features will have to be serviced more frequently in order to meet its safety requirements, while a system with many of these features would be expensive to

design and build. A balance is usually found with the goal being to minimize the cost of owning a system that meets its reliability requirements.

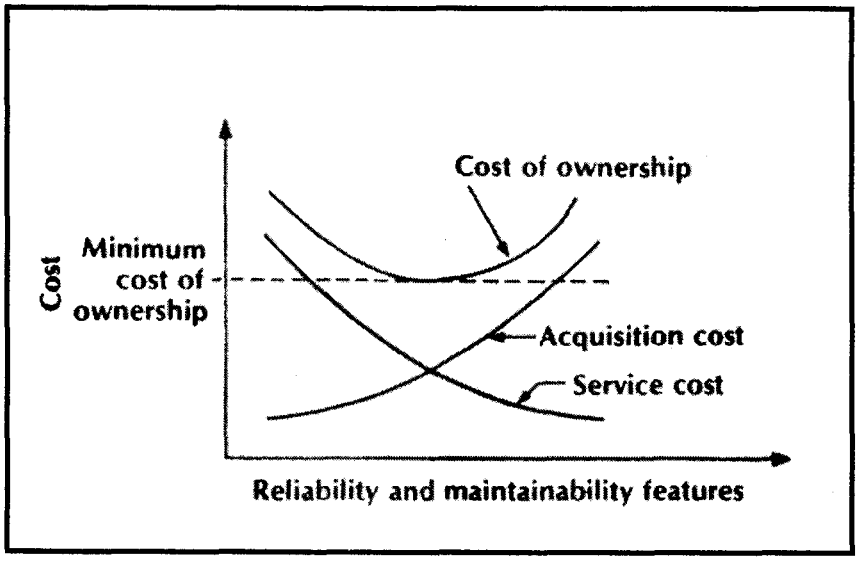


Figure 2-2 Cost of ownership as a function of reliability and maintainability [9]

The strategies of fault avoidance, fault detection and fault tolerance will now be discussed. Within the discussion on fault tolerance, masking redundancy and dynamic redundancy will be discussed.

The first strategy to be discussed is fault avoidance. The premise behind this strategy is very straight-forward. The goal is to achieve reliability by preventing faults from occurring at all. These systems are considered to be fault intolerant because they rely on the prevention of faults but they still fail once a fault does occur.

"Fault avoidance is any technique that is used to prevent faults in the first place" [3 pg. 38]. "Fault avoidance can be obtained by manipulating factors that affect the failure rate" [9 pg. 85]. These failure rate factors include the environment, quality control and the complexity of the system.

A good historical example of fault avoidance can be found when looking at NASA's early space programs. The Apollo missions to the moon involved sending three astronauts to

the moon. The spacecraft was critically dependant on the Apollo Guidance, Navigation and Control computer (AGN&C.) If this computer were to fail at any point in the flight, the astronauts would not have been able to return to earth. Because of severe weight limitations, the spacecraft could only carry one of these computers. "The AGN&C computer relied on simplicity and quality control to achieve high reliability" [1]. The digital logic within this computer was implemented using only three input NOR gates. These were simple and testable to achieve the desired quality levels and thus the required reliability levels. The gates could be manufactured and tested so that if they past the rigorous testing there were verified to be sufficiently reliable for the purpose. The AGN&C computer operated for over 100,000 hours without one recordable fault [1].

With the exception of minimizing the complexity of the system, all of the fault avoidance techniques can be used to increase the reliability offered by any of the other reliability strategies discussed below.

After fault avoidance, the next reliability strategy to discuss is fault detection. With this strategy the system designers realize that a fault is inevitable within a system and they take the approach of finding ways to detect when a fault has occurred. To do this, additional complexity must be added to the system to monitor how the system is performing and to react if any faults are detected.

Fault detection is always accomplished through the use of redundancy. In this context, redundancy is defined as "extra information or resources beyond those needed during normal system operation" [9 pg. 96]. The extra information is referred to as information redundancy. The extra resources are redundant components of the system and additional components which compare the behaviour of the redundant ones.

Hardware redundancy is used to detect faults by techniques such as duplication and monitoring. The simplest fault-detection technique is duplication. This involves duplicating the system with an identical copy and comparing the results. If a failure occurs in one of the two copies, the comparison will detect it and report that the system has failed. Duplication can be used to detect any one fault in the system except for any faults that might exist in the components doing the comparison [9] as shown in figure 2-3.

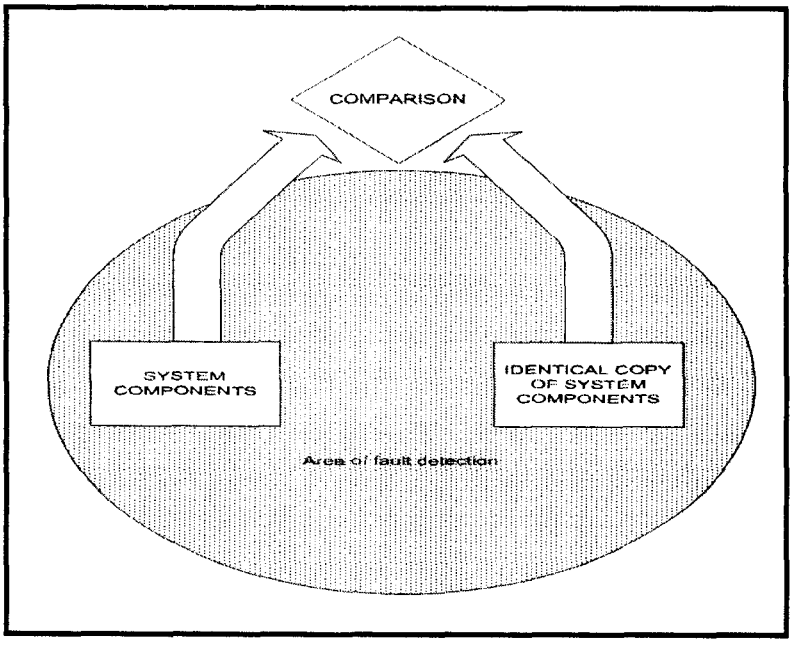


Figure 2-3 Hardware Duplication

Monitoring is another technique used to detect faults. It involves a specific piece of hardware called a monitor which is not a duplicate of the system but instead is a basic subset which can determine if the system is behaving correctly or not. Monitoring has less overhead than duplication [9].

Introducing redundancy in to a system reduces the reliability of the system [1, 9]. A system that uses fault detection techniques will therefore be less reliable than the same system without the redundancy. With twice the components, there are twice the potential failures and with any failure the system can no longer operate correctly. For some systems this is acceptable, as a system can be implemented to shut down once a fault is detected or it can inform a human operator. For other systems that cannot shut down immediately, other techniques are needed to use redundancy in a coordinated manner that will increase the reliability.

These techniques fall into the category of fault-tolerant systems. "The objective of the use of fault tolerance is to design a system in such a way that faults do not result in

system failure" [11 pg. 113]. This is accomplished through the use of redundancy [9]. The problem as discussed above is that introducing redundancy into a system reduces the reliability of the system [1, 9]. "For a redundant system to continue correct operation in the presence of a fault, the redundancy must be managed properly" [1]. Redundancy management techniques have an impact on the performance of the entire system [1]. "A fault tolerant computer can spend up to 50% of its throughput managing redundancy" [1].

The fault-tolerant systems discussed above are sufficient when faced with one failed component in the system but they do not do as well in other circumstances. For example, one component in the system may fail at a particularly critical time for just one moment and then return to normal. Also, it is very difficult to verify that a system will be reliable under the specified conditions. Another example is if a system design fails to address a situation that it will face in operation. Although it could be argued that this system can not deal with the unknown, it would still be beneficial to have the system deal with these problems in an elegant and safe manner. The next section will discuss some solutions to these problems.

2.1.2 Byzantine Resilience and Common Mode Faults

Safety Critical Real Time Systems need to tolerate intermittent and transient faults. Leslie Lamport et al. [15] proposed a way of defining these unexpected faults as well as a solution for dealing with them in their paper titled "The Byzantine General's Problem" published in 1982. The Byzantine failure model is a fault model at the system or component level which includes the possibility of intermittent and transient faults. It is defined as a fault which "may include stopping and then restarting execution at a future time, sending conflicting information to different destinations, and, in short, anything within a failed component's power to attempt to corrupt the system" [1]. It is possible to design a system to be Byzantine Resilient and in fact "designing a Byzantine resilient system is, however, surprisingly simple. Such a system need contain only a pre-specified minimum number of processors and interconnections, provide for their synchronization, and utilize certain simple information exchange protocols" [14 pg. 145]. For safety critical systems, designing a system which is Byzantine resilient is much easier to implement, verify and certify than other ad-hoc

methods of fault tolerance because it "does not require foreknowledge of component misbehaviour and can tolerate faulty components with even the most malevolent behaviour" [14 pg. 145].

The Byzantine fault model adequately solves testing for the intermittent and transient faults [1]. A Byzantine resilient system is able to tolerate a specified number of permanent, intermittent and transient faults. But there is still a class of faults that it does not handle sufficiently, these being common-mode faults. "In general, a common-mode failure occurs when two or more identical modules are affected by faults in exactly the same way at exactly the same time" [3 pg. 387]. In general these faults may be caused by design faults or operational faults; they may be the result of the environment such as electromagnetic interference, or the result of an internal hardware or software error [1].

There is no easy way to model common-mode faults as the very nature of a common-mode failure is something which is unexpected. Lala and Harper [1] discuss a three-pronged approach to reduce the likelihood and impact of a common-mode failure from occurring. Their approach is: "1) Fault-avoidance techniques applied primarily during the specification, design and implementation phases. 2) Fault-removal techniques applied primarily during the test and validation phases. 3) Fault-tolerance techniques applied during the operational phases". They note that all three steps must be taken and only if all three are taken sufficiently, then the risk of a common-mode failure is significantly reduced.

2.2 Avionics

Avionics is a contraction of the words aviation and electronics. It refers to all electronics as applied to aeronautics and astronautics [12]. This section discusses the following aspects of avionics that are relevant to this work:

- Embedded Systems;
- Component off the Shelf Technology;
- Basics of Flight Control Systems;

- History of Digital Flight Control Systems;
- Example of a Current Digital Flight Control System.

2.2.1 Embedded Systems

Due to the nature of Avionics --that is electronics used on aircraft-- these systems can be classified as a particular type of embedded system. An embedded system is a device designed for a specific purpose. Embedded systems typically have tight constraints on both their functionality and implementation in that they must guarantee real time operation reactive to external events Furthermore, they must conform to strict size and weight limitations, budget power and cooling consumption, satisfy safety and reliability requirements, be designed to operate in harsh environments and meet tight cost requirements [16].

The guarantee of real time operations is accomplished by designing for worst case performance. The system must react within specified time constraints. For example, a system used for signal processing would have to meet certain external input and output constraints. Another example could be with a mission critical system where there are control stability requirements, such as on a supersonic aircraft [16].

The weight and size limitations are due to the nature of embedding the system inside a larger artifact, or in many cases a very small artifact. With avionics systems this is especially relevant as the systems are embedded inside an aircraft where size and weight have a significant impact on the rest of the design of the aircraft.

Power consumption and cooling also need to be considered in the overall design of the aircraft. As most avionics systems are safety critical, there is the stringent requirement that the critical systems function correctly even under deteriorating operating conditions. For example, some avionic systems may need to operate while in flight even when the main power has failed (such as if the engines shut down) and the backup power is supplied by an auxiliary power unit or ram air turbine. However, some uninterruptible power source battery

backup may be needed to keep the system operational until the backup power source can be brought online.

The safety and reliability requirements are similar to those discussed in the previous section of this chapter. With avionics systems, care needs to be taken to guarantee that if a system does fail, it will fail safely, meaning that it will fail in a predictable manner [11]. In some cases, for critical components if a failure occurs, it must be a fail operationally, meaning that sufficient redundancy exists in the system to guarantee it will continue to function correctly for a long enough period of time before the system is shut down. For example the system must operate long enough so that the aircraft can get safely on the ground [2, 4].

Most embedded systems and particularly avionics systems have requirements for operating in harsh environments. Most computers are designed to operate in a dry environment with temperatures between 15°C and 30°C. Aircraft typically fly in much harsher environments with temperatures ranging between at least -40°C and 50°C and varying air moisture and pressure [2]. These conditions need to be taken into account when designing and maintaining these systems.

Finally, the economic requirements for embedded systems usually have some bearing on design decisions. This is less significant in avionics systems and the safety requirements dictate the need for further spending to design the system right.

These embedded systems design criteria and considerations each play a significant role in designing a fault tolerant flight control system.

2.2.2 Component off the Shelf Technology

Use of component-off-the shelf technology is gaining continued acceptance in safety critical systems. The reason for this is, that by using existing developed components such as formally verified processors or Real Time Operating Systems, the system designers gain the

industry's large investment in testing and verification with little of their own costs. This significantly reduces the overall cost of development [1]. Even for processors with known faults, such as the Intel 486 processor (division error), extensive investment and deployment allows users to confidently adopt the processor in a safety critical application because its behavior is well understood [4]. The use of this technology can significantly aid in reducing the costs of developing, testing and verifying a safety critical system [1]. Also by using existing industry knowledge, it is easier to design and verify the reliability of the system.

2.2.3 Basics of Flight Control Systems

Since this thesis addresses flight control systems, it is important to define these systems and set the context for the relevance of this work. A flight control system is the system on an aircraft responsible for controlling the flight control surfaces. Depending on the size and type of aircraft, the number of control surfaces and complexity of controlling them can vary greatly.

Figure 2-4 shows the various control surfaces of a modern wide-bodied commercial aircraft, the Boeing 777.

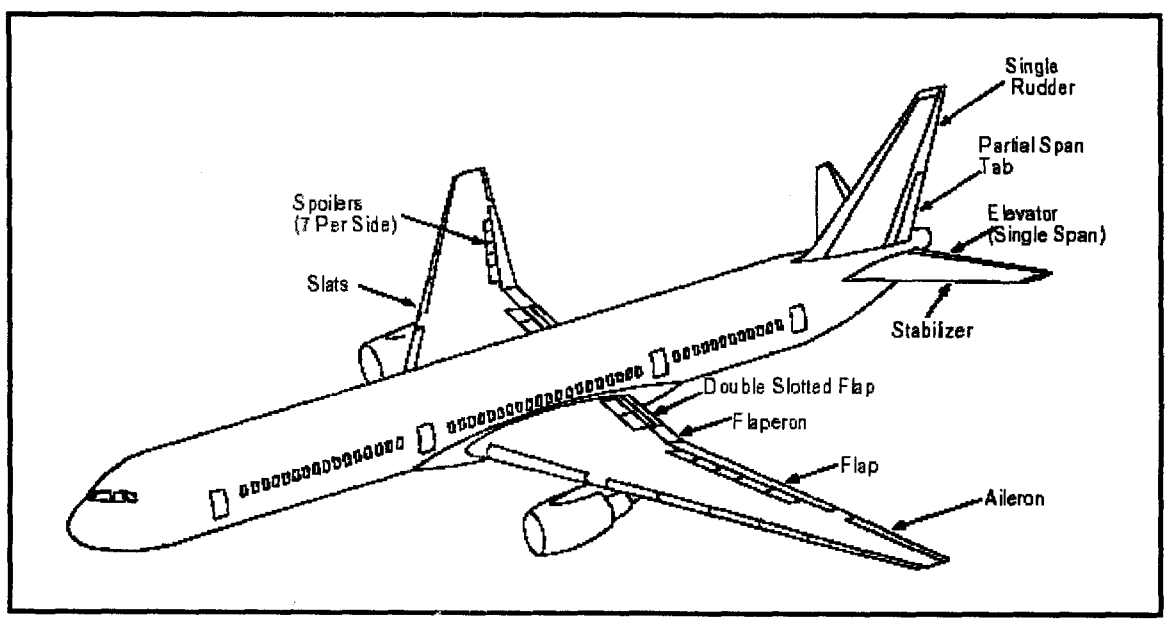


Figure 2-4 Control Surfaces of Boeing 777 [4]

The flight control system takes its inputs from the pilot or autopilot and moves the control surfaces. Traditionally the surfaces were moved by cables and pulleys which were directly connected to the controls in the cockpit. Most small aircraft still use this method but larger aircraft can not, because the large forces required to move the control surfaces exceed the amount of force a pilot can exert. To generate sufficient forces to move these surfaces, hydraulic systems have been typically used. These systems are large, heavy, difficult to maintain and need to be redundant. Large amounts of area and weight are used up by these systems. Commercial aircraft operators need to generate revenue and the presence of these systems requires frequent maintenance and lost passengers and cargo. The Boeing 777 shown in figure 2-4 uses a digital fly by wire system as a modern alternative to a hydraulic system.

2.2.4 History of Digital Flight Control Systems

As an alternative to hydraulic systems, industry has been adopting digital flight control systems since the mid 1980's. These systems are usually referred to as digital fly-by-wire systems because control signals are sent to the surfaces by electronic signals on wires instead of the traditional mechanical linkages [11]. The first example of a digital fly-by-wire system was developed by NASA in the 1970's in experiments for military aircraft and, eventually, the space shuttle [1].

A slightly earlier example is the Concorde aircraft which began flying in 1969 with a three-axis full-authority analogue fly-by-wire system. But this system was not a digital system. It led to additional generations of fly-by-wire systems and eventually to the development of a fully digital fly by wire system for the Airbus A320 in the mid 1980's. In the early 1990's Airbus continued its commitment to digital fly-by-wire with its A330/340 program.

There is controversy as to the way Airbus's fly-by-wire system operates as the computer is able to override the pilot in cases where the computer considers the pilot's actions beyond the envelop of safety [10]. Boeing responded to these concerns with their

first digital fly-by-wire system with a different architecture in 1995 [2, 4]. The fundamental difference between the two systems lies with a decision over how the system deals with potentially unsafe situations. The Airbus system will override a pilot to prevent the aircraft from operating in what the computer considers to be an unsafe manner. Under normal operating conditions there is no reason why a pilot should operate the aircraft in an unsafe manner. It is possible that a situation would arise in which the pilot fights the aircraft and loses even though the pilot might know better than the flight control system. It is important to note that it is even more likely that the Airbus system has prevented pilots from inadvertently operating their aircraft in an unsafe manner.

Boeing has designed a fly-by-wire system with a different paradigm. This system allows pilots to override the computer if they feel it necessary. The system still offers similar unsafe operation preventions but the pilot is able to flip a switch and disengage the safe guards if necessary. This minor difference between these two systems has led to significant safety discussions and a public perception that the Boeing system is safer than the Airbus system. The next section will discuss the Boeing fly-by-wire system.

2.2.5 Example of a Current Digital Flight Control System

The Boeing digital fly-by-wire system contains an extra level of redundancy over the Airbus system. In the Boeing system the pilot's signals are sent to the Actuator Control Electronics (ACEs) which are quadruply redundant components that were designed and built to be as simple and reliable as possible. Similar to NASA's AGN&C computers used in the Apollo missions, these ACEs are designed with fault avoidance in mind and then the system still has four redundant units in case some of them should fail. The ACEs are capable of sending the pilot's signals directly to the aircraft's control surfaces in a fail-safe mode if the primary flight control computers were to completely fail [4].

When the primary flight control computers are working correctly, the ACEs send the pilot's signals onto the flight control data bus where they are received by the primary flight

control computers. These computers process the signals and calculate the appropriate control surface movements and send the modified signals back to the ACEs where they are then sent to the control surfaces. Figure 2-5 shows an overview of the complexity of the Boeing 777 Flight Control System.

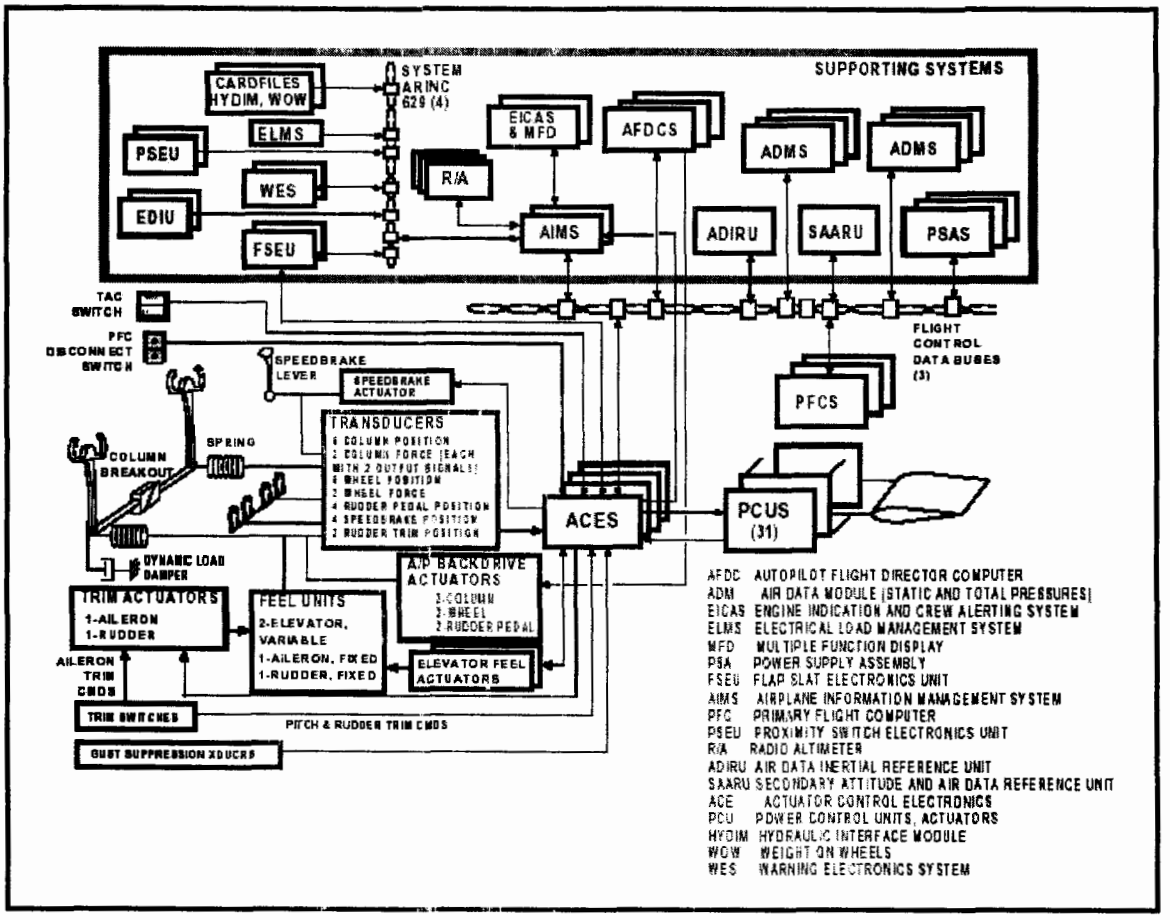


Figure 2-5 System Overview of the Boeing 777 Flight Control System [4]

The Boeing 777 actually has three redundant data channels defined as the left, center and right data buses. These are installed at different locations in the aircraft so that in the event of minor structural damage it is highly unlikely that all three channels will be damaged. The primary flight control computers are discussed in more detail below. The rest of the components in the flight control system will not be discussed as the overall system is too complex and beyond the scope of this work.

The Boeing 777 has three primary flight control computers (PFCs). Each of them is installed in a different physical location within the aircraft and they are referred to as the left, center and right PFC's. Each primary flight control computer is attached to all three flight control data buses in order to receive data. Only one primary flight control computer can transmit data on each flight control data bus. For example, the left primary flight control computer is the only primary flight control computer that can transmit onto the left data bus. This is done to support Byzantine resilience in the system. Each computer receives data from all the same sources and it will compare the results it calculates with the results of the other primary flight control computers in order to identify Byzantine faults and prevent the system from failing.

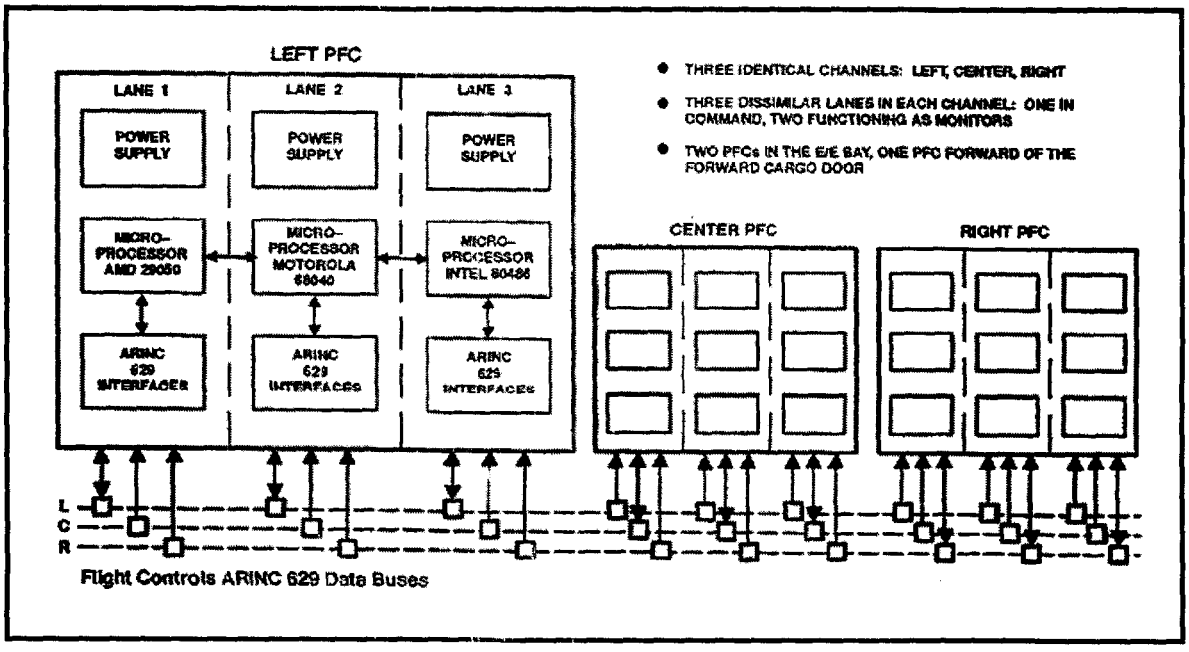


Figure 2-6 The Three Primary Flight Control Computers on the Boeing 777 [8]

Each primary flight control computer is an identical line replaceable unit. Within each primary flight control computer are three lanes of dissimilar processors. The lanes are functionally equivalent. Each lane is one component of the triple redundant primary flight control computer. The system contains three of these computers making the Boeing fly by wire system a "triple-triple" redundant system. One lane is always in command while the other two lanes are in monitor or standby modes. If one lane is identified as faulty, the monitor will force that lane to shut down and one of the remaining lanes will assume the

command role. At power up, each lane is dynamically assigned a role and each of the three primary flight control computers must select a different lane for the processor which is assigned the command role [17].

Each processor lane is basically identical in functional capability but dissimilar in exact implementation. The three processors used in the primary flight control computers are the AMD 29050, the Motorola 68040 and the Intel 80486. The processors are running the same Ada code but are running dissimilar compiled code because different compilers were used for each processor. This is done to reduce the possibility of a common mode fault from crippling the system.

This chapter has discussed topics related to safety critical systems and avionics. This material was presented to give context and background for the rest of the thesis. In the next chapter the design of the fault tolerant flight control system will be discussed.

Chapter 3

Design of System

This chapter takes the reader though the process of creating the requirement specifications and system design. The requirement specification for this work was necessary in order to define the realm in which the system would be required to function and more importantly define a boundary of requirements and functions to be considered beyond the scope of this work. With the requirements complete, decisions had to be made on hardware to be used and how to implement the system.

3.1 System Requirements

The purpose of this work was to take a much closer look at the design and implementation of fault tolerant systems while paying particular attention to the unique requirements of the avionics field. It was decided that the end product of this work would be a functioning fault tolerant flight control system. We determined that our goal would be to show that a fault tolerant system can be designed and implemented using fairly inexpensive component off-the-shelf (COTS) technology.

Developing a fully functioning fault tolerant flight control system would require a team of engineers from many different backgrounds and a very large number of resources, even without considering the design and construction of the aircraft in which the system is to be embedded. In order to reduce the work into a reasonable amount for a Master's thesis, care was taken to define the realm in which the system would be required to function and more importantly define a boundary of requirements and functions to be considered beyond the scope of this work.

Extensive study of some publications on the Boeing 777 DFBW system [2, 4, 8] allowed us to evaluate how such a system is designed and implemented. The publications thoroughly addressed the reliability achievements and comprehensive testing of the aircraft but it became apparent that insufficient details were available to directly implement even a scaled down version of the design. Obviously Boeing chose to keep many implementation details within its realm of intellectual property.

At the other end of the spectrum, the implementation details of the flight controller used in the Stingray UAV [7] were complete enough that implementing the design could be attempted. Unfortunately the Stingray design did not attempt to meet any of the requirements for safety critical systems.

It was decided to explore a system design that would control a simple and basic aircraft system (like the Stingray UAV.) This system would incorporate a form of fault tolerance to increase the reliability. It is important to note that the intention of this work was to go as far as to show the feasibility of the concept. As this work was done in the context of computer science it was decided that we would not go as far as flight testing and actually flying the system. Focus was to remain on the computer science problems of designing and implementing a reliable system instead of the avionics problems of installing and testing such a system into a operating aircraft.

A feedback control system contains a controller that must:

- be a real time system which takes input signals from various sensors;
- makes decisions based on these inputs;
- sends output signals to control components of the system;
- receives feedback on the results of its actions.

In order to add fault tolerance to a feedback control system one must add redundancy to the components. For this work, only the controller itself is to be fault tolerant. The reliability of the input sensors and output control components is assumed to be

handled by other means. Figure 3-1 shows the portion of the system that is to be fault tolerant in white. The shaded portions contain the components of the system in which reliability is necessary, but considered beyond the scope of this work.

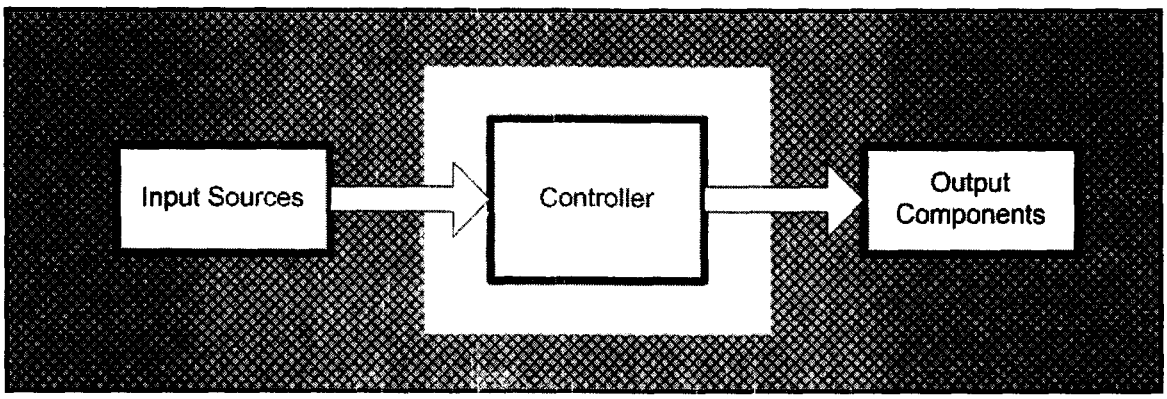


Figure 3-1 Scope of Reliability

In order to add reliability to this system a form of fault tolerance needed to be added. In the previous chapter various methods for achieving system reliability are discussed. It was decided that, for this work, a form of masking redundancy called Triple Modular Redundancy (TMR) would be used and it is shown in Figure 3-2.

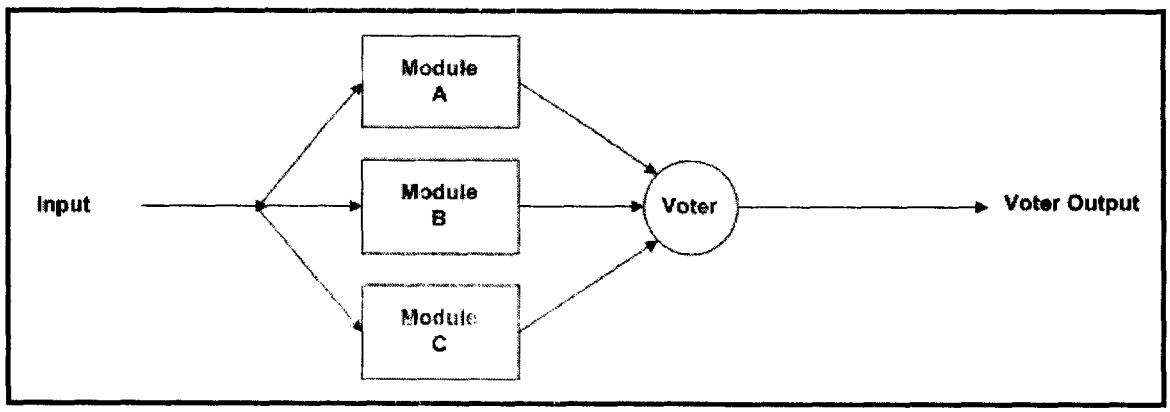


Figure 3-2 A Fault Masking TMR System [9]

The advantages of a TMR system over its comparable non-redundant system is that there are three identical modules instead of one. The three modules can use the same input sources and they each operate independently. Each module will individually decide on an

output which is sent to the voter where a majority vote takes place. If one of the three modules is faulty, its incorrect vote would be masked out by the other two modules. This makes the TMR system more reliable than the comparable non-redundant system as the TMR can tolerate one faulty module while the non-redundant system can not. A TMR system allows for a fairly simple addition of fault tolerance as the redundancy only needs to be managed in a voter module.

A TMR system has some disadvantages. It is only able to mask faults. Thus, no online repair or reconfiguration can take place. The system as a whole can only tolerate the one faulty module. If a second module were to fail, the system would no longer function correctly even though it might still have a correctly working module within it.

It is clear from the literature on the Stingray UAV [7] that a PC/104 single board computer can meet the real time requirements needed to control a small aircraft. With the addition of redundancy and voting to the system we were unsure if we would still be able to meet this requirement. It may have been the case that these additions would take too long to meet any reasonable real time requirement. For the sake of this experiment it was decided to attempt to complete ten votes per second. This would require the system to receive ten inputs per second and to transmit ten outputs per second. Without implementing the system with voting we did not know if this was possible to achieve.

Tolerating Byzantine faults is out of the question with TMR as it does not meet the required level of redundancy [15]. In order to tolerate Byzantine faults the system must contain 3m + 1 redundant components where m is the number of Byzantine faults to be tolerated. It was decided that tolerating Byzantine faults would be considered beyond the scope of this work. We would require at least four identical components to tolerate one Byzantine fault. Although tolerating these faults is a very important consideration in developing safety critical systems, it is not the main focus of this work.

With the TMR architecture in mind, we then needed to further define the capabilities of the system. In order to meet the high level goal of controlling an aircraft in real time, we explored the computer requirements for this task. The controller would need to accept

multiple data values at a frequency of many readings per second from various sensors. The controller would then have to evaluate those readings, consider its current goal (what the system should be doing) and then make a decision on appropriate control outputs to send to various output components. Figure 3-3 shows a high level architecture of the system that was to be implemented.

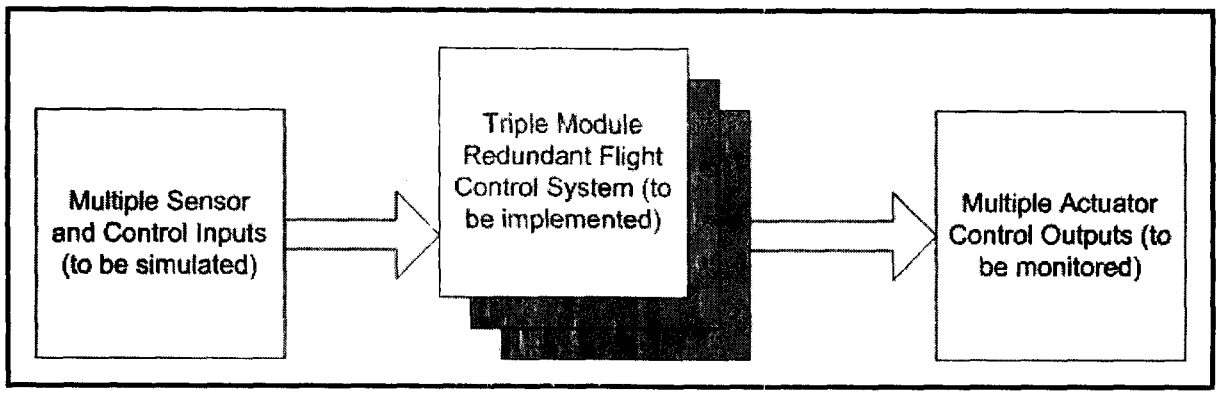


Figure 3-3 High Level Architecture of System to be Implemented

Figure 3-3 also indicates the type of work for each portion of the system. The inputs could come from actual sensors but it was decided to simulate the input data to save on costs, development time and testing complexity. The outputs could also be sent to actual servos or actuators on an aircraft or test bench but it was decided to just monitor the outputs in a manner that would visually make it obvious the system was working, without expending much time or money.

The TMR control system is the main focus of this work, but some work had to be done to correctly simulate input data and to monitor output data. The remaining part of this section discusses the criteria we used to make our implementation decisions .

With the decision that the TMR controller would be implemented, the first step was to select a hardware platform to be used. The hardware platform had to meet the following criteria:

- Embeddable;
- Programmable;

- Affordable;
- Reproducible;
- Usable.

The decision to chose an embeddable platform was made to give this work some potential. It was hoped that this project might be carried on by another student wishing to take on the challenges of installing the system on an aircraft, or similar application, to test it.

The platform needed to be easily programmable in order to aid in efficient development of the work. The basic criterion used to determine if a platform was sufficiently programmable for this work's purposes was that the platform had to have a C compiler available. This way the work could be programmed at a high level language instead of low level machine language.

The affordability criterion was defined as a level of cost which Dr. Jon Muzio was willing to spend on this work. No initial amount of funds was named so various options were explored and negotiated until we arrived at a reasonable solution.

The platform was required to be reproducible. The purpose of this criterion was to allow others to continue the work by purchasing the same hardware and running the same software. This is a requirement often considered vital by industrial suppliers and it is often referred to as "component off-the-shelf technology."

The usable criterion was necessary as it was in the best interest of all involved to minimize the time required to set up and operate the hardware. Thus a familiar platform would be preferable over one that required more time to learn how to set up and operate.

This chapter has discussed the design of the fault tolerant flight control system. In the next chapter the implementation and results of this work are discussed.

Chapter 4

Implementation of System and Results

This chapter discusses the implementation of the system and the final results achieved when the work was completed. The implementation took place between August 2003 and March 2004. Once implemented, the results were reached quickly.

4.1 Implementation

Implementing the system described in the previous chapter took nearly a year and many detailed design decisions had to be made. These included:

- Hardware platform selection;
- Hardware platform ordering and purchasing;
- Hardware platform development environment setup;
- Test environment;
- Communications protocol setup;
- Implementation of a non-redundant flight controller;
- Addition of redundancy to flight controller.

4.1.1 Hardware platform selection

Five hardware platform options were identified and then the criteria discussed in the previous chapter were used to make the final decision. These five options were:

- Option 1 – PC/104 Embedded PC Architecture;
- Option 2 - Standard Desktop x86 Systems;
- Option 3 - PIC Microcontrollers;
- Option 4 - Motorola HC-11 Microcontroller;
- Option 5 – Continue Search.

The PC/104 Embedded PC Architecture was the hardware platform used in the Stingray UAV [7]. The platform uses the standard PC architecture including x86 based processors, PCI and ISA buses but in a compact and embeddable format. The PC/104 boards measure 9cm by 9.5 cm in size. They are stackable and are designed to have low power consumption.

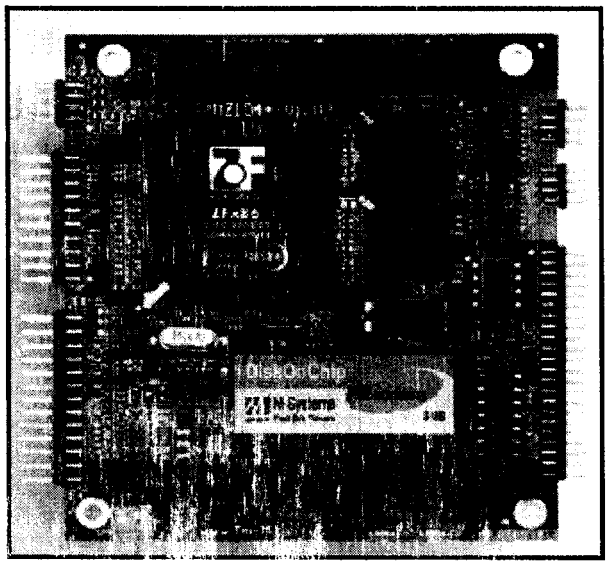


Figure 4-1 A PC/104 CPU Board [18]

The standard desktop x86 system does not need much description as desktop PC's are common knowledge. This platform was considered because it is so easy to use and a sufficient number were available for use without any cost. We determined that if there was

no advantage to using any of the other platforms, the standard desktops would be sufficient to complete the desired work.

The PIC Microcontrollers were suggested as an option by Dr. Kin Li during a conversation related to his Software Engineering 540 – Embedded Systems course [19]. These microcontrollers are very small and easily embeddable in any application. Figure 4-2 shows a enlarged image of the PIC Microcontroller.

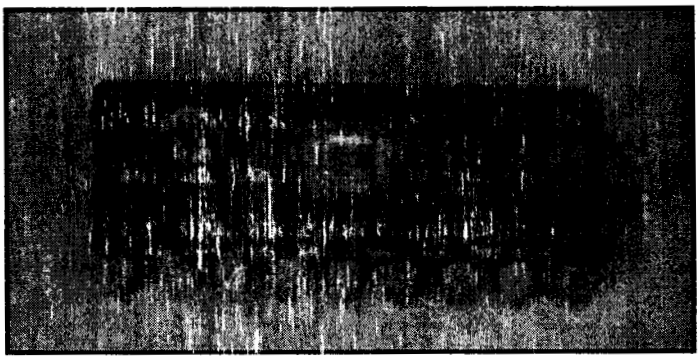


Figure 4-2 PIC Microcontroller [20]

The Motorola 68HC11 processor has been used in courses offered by the department of Computer Science at the University of Victoria so was is a platform which we were familiar with. A version of this processor is shown in Figure 4-3.

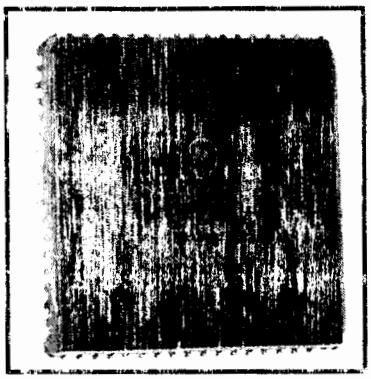


Figure 4-3 Motorola 68HC11 Microprocessor [21]

The final option was to consider using one other candidate. We felt that this option should only be explored if we failed to find a suitable platform amongst the already

identified candidates. It is important to note that a tradeoff usually has to be made between endlessly searching for the perfect product and finding one that is good enough for the required purpose. There may be a better hardware platform available that was not considered for this work but in the interest of moving on and completing the work on schedule, only the previously mentioned options were considered in detail.

Figure 4-4 shows a feature-by-feature comparison of the candidate platforms when considered, using the criteria discussed in chapter three. Each feature was rated as yes, no, or possibly. The 'yes' value was given to features that we knew for certain to be present in the candidate platform and the 'no' value was given if we knew for certain the feature was not present in the platform. The 'possibly' value was given if we felt that the presence of the feature was unclear. For example, the PC/104 platform was advertised to be affordable and usable, yet, without first hand experience of using these computers, it was unclear how much one would have to pay to acquire a fully working PC/104 system. Likewise, the amount of additional work necessary to use the system compared to more common and familiar platforms was unclear.

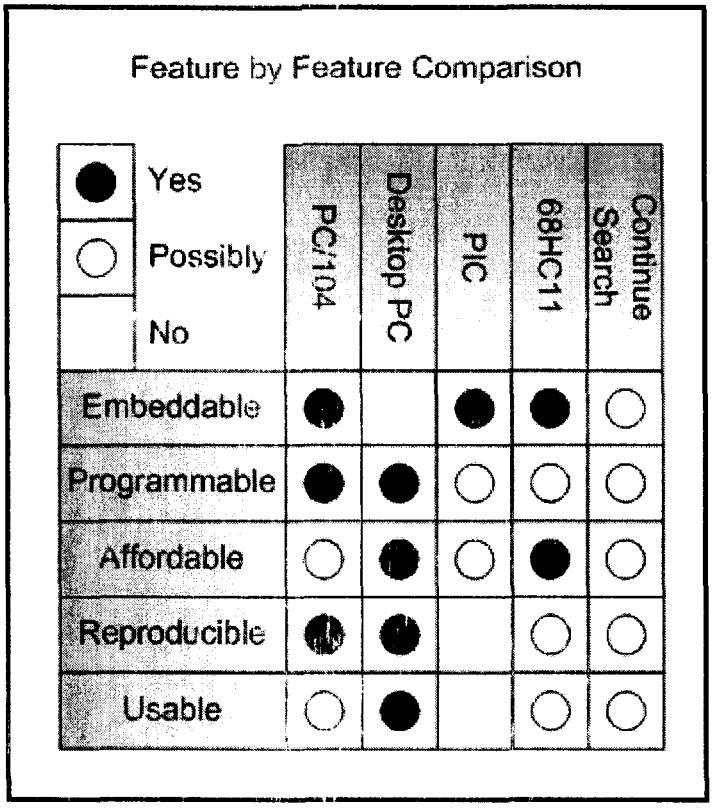


Figure 4-4 Feature by Feature Comparison

After a detailed evaluation, we chose the PC/104 platform. The PIC Microprocessor would have required too much time to learn how to work with the platform and custom circuitry to set up the TMR flight controller. The Motorola 68HC11 option would have used the BLT boards and would have required extensive work to set up the TMR flight controller. Also, for both the PIC and HC11, there was a concern that they might not offer sufficient memory and processing power to accomplish the task. It was considered risky to do the work with either of these platforms with the possibility that insufficient resources might halt the work before it was completed. The desktop PC option was not embeddable so the PC/104 platform seemed more appropriate.

4.1.2 Hardware platform ordering and purchasing

With the decision made to use the PC/104 platform for this work, we began researching different PC/104 single board computer (SBC) products. Again, criteria were set

to narrow down the search. We contacted different suppliers who made products that fit within our required criteria and went with the best option we could find.

The criteria used to narrow down the selection were:

- x86 based processor;
- at least three communication ports per SBC;
- development platform available;
- runs common Operating Systems;
- product donation or academic pricing available.

An organization called the PC/104 Consortium (www.pc104.com) publishes a list of companies that produce products that conform with the PC/104 standard [22]. By using this list to locate and browse many company web sites, we were able to narrow our choices down to products made by three companies. These companies were VersaLogic Corporation, Ampro Computers Inc. and Tri-M Systems & Engineering.

We contacted each of the three companies to inquire about pricing and the possibility of getting the equipment donated or reduced in price for this academic work. Each of the companies responded to our requests in a professional manner. VersaLogic Corporation politely declined offering any special pricing for this work and they even suggested we would be able to find cheaper products with other companies. Ampro Computers Inc. and Tri-M Systems each responded in a positive manner and asked for more information.

Ampro Computers Inc. sells their products in Canada through a company called Integrys, which is a subsidiary of Allan Crawford Associates Ltd. We spoke to the regional sales representative about this work and our inquiries were warmly received. We were invited to a upcoming sales seminar in Vancouver for the purpose of meeting directly with some representatives of Ampro directly after the seminar. The meeting was very successful and at the end we were told that they would take our proposal back to their head office and see

how much support they could offer. Regrettably, their final offer was a price reduction of just a few percent off of the price of some of the equipment we would need. We would still have to purchase the rest of the equipment at full price.

Tri-M Systems was also receptive to our inquiries. They offered a significant reduction of prices for their PC/104 SBC's plus if we did purchase the computers they would give us many needed peripherals at no charge. The decision was made to order three SBC's from Tri-M Systems, along with a development kit, three power supplies and various connection wires and hardware. Figure 4-1 shows the Tri-M Systems MZ104+ SBC and figure 4-5 shows the contents of the development kit.

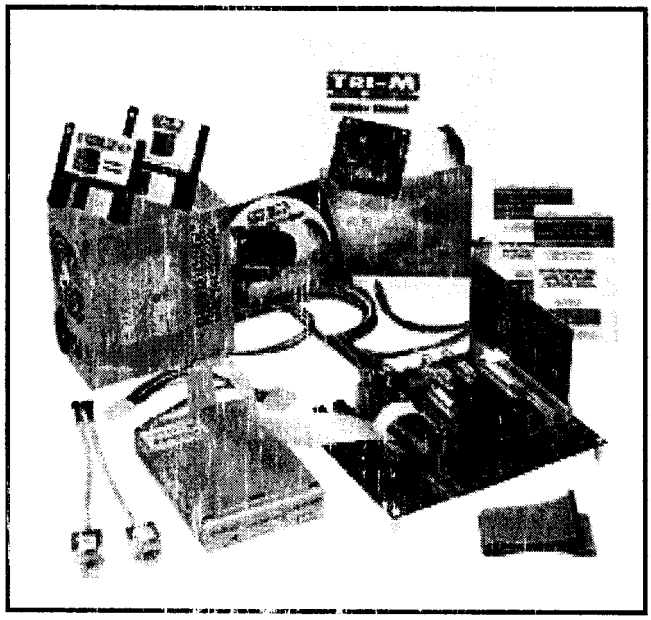


Figure 4-5 Tri-M Systems MZ104+ Development Kit [18]

The specification sheet for the MZ104+ SBC is included as Appendix A. This SBC included 32MB of RAM, a 100 MHZ x86 based processor, IDE port, floppy drive, keyboard, mouse, parallel port and two ports each of serial, USB and Ethernet. The development board converted the PC/104 buses into standard desktop PCI and ISA buses so that up to two PCI cards and two ISA cards could be connected to the SBC for development purposes.

4.1.3 Hardware platform development environment setup

When the hardware arrived it took very little time to set up the system and get it running. The development kit allowed for a graphics card to be attached to the system so that we could work directly on the SBC instead of accessing it remotely. This proved useful at times when the SBC locked up and needed to be reconfigured. We also attached an Ethernet card to the development kit giving the SBC access to three different Ethernet ports.

One of the first setup tasks was to replace the included 8MB disk-on-chip (DOC) flash drive on the SBC with a 48MB DOC. Compressing an operating system and our application in just 8MB of memory might have been difficult so we chose to avoid this problem by using larger disk storage. The process of preparing a bootable DOC involved installing the chip onto the board, booting from a floppy, and then flashing the DOC with an image file. Obviously a 48MB image file would not fit on a floppy.

We intended to use an IDE hard drive for this purpose but unfortunately we discovered that this was not possible because the supplied IDE cable for the SBC did not fit into a standard IDE hard drive as the cable and connectors were much too small. It turns out that Tri-M Systems had decided to use 44 pin laptop IDE connectors and cables on the SBC. The standard IDE connector is only 40 pins and the additional pins on a laptop IDE connector are to supply the drive with power. The 44 pin connectors are also more compact than the usual 40 pin connectors. We were unable to proceed without acquiring additional equipment.

Fortunately, we were able to find or borrow the additional equipment we would need. We learned that 1.3 GB Laptop hard drives were available at used computer shops for around thirty dollars and a 40/44 pin IDE converter cable was available for use within the department. We were able to load the image file onto the laptop hard drive by using the

converter cable and connecting the drive to a desktop computer. Then we were able to connect the drive to the SBC and load the image file

With the image loaded and the SBC fully functioning, it was time to test the code development and build process that would be used to compile the software to run on the boards. The plan was to develop the source code on another machine, transfer it to the SBC where it would then be compiled into an executable program. Unfortunately, we quickly discovered that the SBC's operating system image did not include any C-Compiler. Again it turned out that the Tri-M Systems MZ104+ was missing a critical component that was not clear in the sales literature.

A telephone call was placed to Tri-M systems to inquire why they would sell their product as a development kit if you could not develop any software on in. It was politely explained that we needed to develop our executable files on a desktop machine running the same operating system as the SBC's and then we would find that the developed program would run on the boards without any problems. We had to set up a desk top computer with Linux Slackware version 7.1. It is important to note that, at the time of writing this thesis, version 9.1 was the current release, so much time and effort was spent locating the obsolete version. We could have set up a DOC image with another operating system or a more updated version of the same operating system but we felt we did not have the expertise to carry out this additional work and one reason we had chosen the Tri-M Systems SBC in the first place was because it offered ready to boot images of Linux.

With the new desktop set up and running Linux Slackware 7.1, we were able to create executables that would run on the SBC's. With this step we had completed the setup of our development environment and could move on to work on implementing the flight control system.

4.1.4 Test environment

Up to this point in the thesis, little has been mentioned as to how the input/output of the flight control system would be set up. In Chapter Three it was stated that the inputs would be simulated and that the outputs would be monitored in such a way as to make it visible that the system was working correctly. What was needed was a test environment that could supply multiple sensor data at a fairly fast rate and could receive control outputs from the flight control system in a manner that would show that the system was working.

It was determined that Microsoft Flight Simulator (MSFS) offered all of the features we needed for our testing environment. The flight data from the simulation could be extracted using an utility called FSUIPC developed by Peter Dowson [23]. This utility allows software to gain read and write access to flight simulator's variables while the simulation is in progress. It can be used to update the simulation's weather, customize the pilots controls (for people building home cockpits) or for developing any other variety of applications to work with flight simulator.

The FSUIPC utility made it possible to simulate an aircraft's flight data, within the MSFS program, with sufficient accuracy for the purpose of this work. It also allowed us to send flight control signals from the flight control system to the simulated aircraft with immediate visual results. Figure 4-6 shows the TMR flight control system connected to a PC running Microsoft Windows, Microsoft Flight Simulator and FSUIPC.

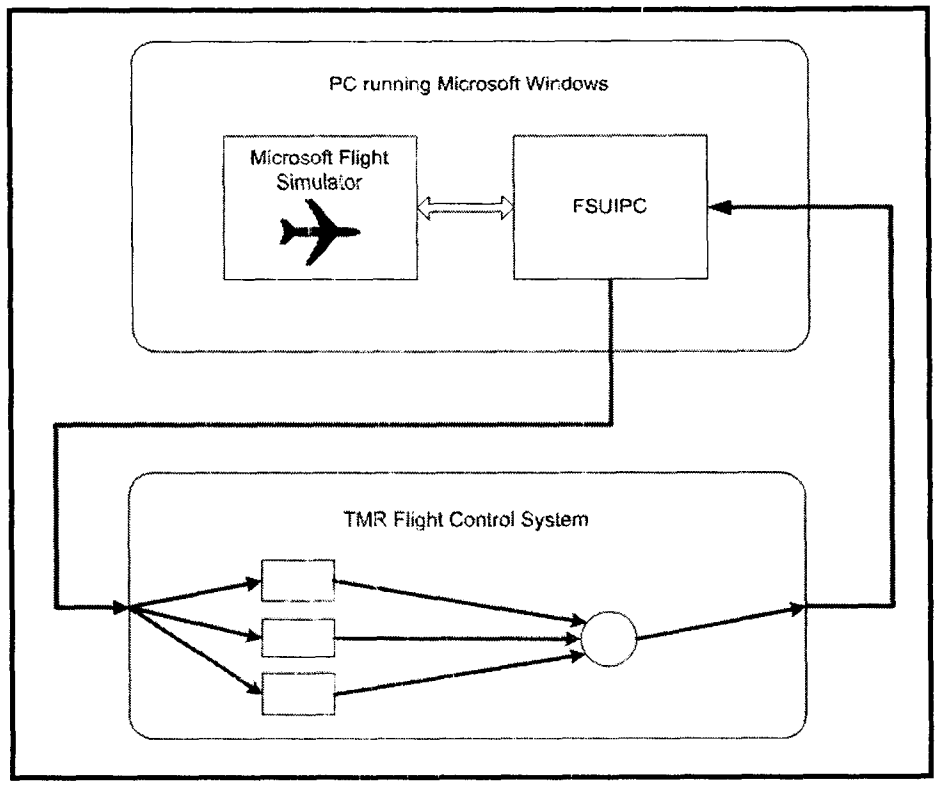


Figure 4-6 Proposed System setup using MSFS and FSUIPC for input/output

FSUIPC does not directly communicate with any application. It offers an application programmers interface (API) to access flight data and carry out various functions in MSFS. We needed to write an application which would use FSUIPC to extract the necessary flight data from MSFS and then send that data to the TMR Flight control system. The same application would also have to receive data and send the control signals to MSFS. We reviewed the FSUIPC software development kit (SDK) but found the documentation difficult to understand. Before investing much time working on this windows application, we decided to post a question on the Pete Dowson Forum on www.simflight.com, which is a common forum for third party developers using FSUIPC. Our posted inquiry explained what we were trying to do and asked if it was possible and if anyone could offer any tips or help.

Within a few days, we received an enthusiastic response from a member of the site, named Sean McLeod, indicating that we definitely could use FSUIPC as we were planning. He also offered to develop the application for us. Seeing as this was periphery work to the

main focus of the thesis and potentially involved a steep learning curve and unforeseen difficulties if we were to develop the application on our own, we decided to accept Sean's offer.

Through multiple emails back and forth we were able to describe the requirements of the windows application to Sean and he took about two weeks to implement it. The end result was an application called PFCBus (primary flight control bus) which would run in the background of the computer running Flight Simulator. This program would extract flight data from flight simulator at specified frequency (default is 10 Hz) and send this data by UDP network packets to a receiver. More details regarding the communication protocols are discussed in the next section. The program would also listen for incoming UDP packets, which it would receive and communicate to flight simulator. Sean also developed a test program called PFCBusTester which would receive and send the UDP packets so that the PFCBus program could be tested.

With a fully working and tested PFCBus application, we were confident we had a sufficient test environment in which to run the TMR flight control system.

4.1.5 Communications protocol setup

As mentioned in the previous section we chose to use User Datagram Protocol (UDP) packets to communicate between modules. We needed a protocol that would allow for many packets to be sent per second and would not experience bottlenecks due to a faulty component. By always hooking up two computers together by dedicated Ethernet we could avoid some of the common problems of a computer network such as packet loss and delay. Each sender would only have one receiver on the same network so if the packet was not received, location of the problem would be identifiable. UDP has the advantage of being a connectionless communication protocol as packets are just sent to an address without the need of opening and closing a connection. Also UDP packets are not acknowledged by the receiver. There was no need for packets to be retransmitted if they were not received correctly.

Two types of UDP packets needed to be created for this work. The first of these packets was to contain the flight data that was extracted from flight simulator. The second was to contain the flight control surfaces data to be sent back into flight simulator.

The flight data packet was called the Aircraft Attitude packet. It contained a sequence number and values for each of the Pitch, Roll, Heading, Altitude and Airspeed of the aircraft as these values are all needed to determine the state of the aircraft while in flight. The FSUIPC documentation supplied the details as to which data types to use for these values and also the valid ranges to expect for each value.

The flight control surfaces packet was called the Aircraft Flight Surfaces packet. It contained a sequence number and values for each of the Elevator, Aileron, Rudder, Throttle, and Elevator Trim positions.

The header file PFCBus.h was used throughout the various applications to create the data structures for these packets. This header file is included as Appendix B. Originally the sequence numbers were defined as unsigned integers but we found that compiling programs with this header file produced different data structures in windows than in Linux. Apparently the Windows compiler was generating unsigned integers to be 64 bits wide when they should have only been 32 bits. We had to declare the sequence numbers a long long, which produced a 64 bit wide variable in both Windows and Linux.

4.1.6 Implementation of a non-redundant flight controller

With the communication protocol defined and the test environment set up we were ready to begin implementing our flight control system. It was decided that we would develop the system sequentially by first getting the system to work with no redundancy included and then introducing redundancy to the already working flight control system.

Care was taken to design the system in a way to simplify the introduction of redundancy when needed. To do this we decided to implement the flight control system into three software programs, PFCBus, PFCNode and PFCVoter. For the non-redundant version of the system these modules were set up to do the following:

PFCBus was responsible for listening for new Aircraft Attitude packets being sent from the PFCBus program running on the computer running flight simulator. When PFCBus received a packet, it forwarded it on to the PFCNode. This was done so that eventually the PFCBus program would send the same packet to three redundant nodes.

PFCNode is the program that preformed the flight controlling. This program acts on transactions. It listens for an Aircraft Attitude packet from the PFCBus program. When the packet is received the PFCNode program makes aircraft control surface position decisions based on the attitude date with the goal of flying the aircraft "straight and level". These decisions were then formed into a flight control surface packet and sent to the PFCVoter program. Once sent, the PFCNode program would wait for the next packet to be received. It should be noted that with regard to performance for this program, it is necessary that it run fast enough so that it can complete its loop in the time period between incoming packets.

The PFCVoter program would receive the flight control surface packets and send them back to the windows PFCBus program. The program would gain complexity and significance with the addition of the redundancy later on.

The three modules were written in a manner to allow their physical location within the system to be portable. The modules could be easily moved from one SBC to another. This was done to support incremental development of the system and to allow ease of expansion and reconfiguration. For example, at first all three programs were run on just one SBC but the next step was to move the PFCNode program onto a different SBC.

4.1.7 Addition of redundancy to flight controller

Upon the completion of the non-redundant flight controller, all that remained to be added was the triple modular redundancy to complete the project. Adding the redundancy required that the input date be sent to three computers. Each of those computers ran a similar version of PFCNode and the output from each computer was sent to the PFCVoter for majority voting.

Due to a few limitations on hardware, we had to make some compromises in our design. The MZ104+ SBC modules each contained two Ethernet ports so that they could each connect to the other two modules on dedicated lines. One of the three SBC's, called CYYZ, was attached to the development board and had a third Ethernet port. The other two SBC's were named CYEG and CYYJ. We only had three SBC's so there was going to be some difficulty with dividing the input data in to three SBC's and having a fourth SBC to serve as the voter. A laptop computer called CYVR was used to run MSFS.

Figure 4-7 shows the arrangement of the computers and what software programs ran on each computer. CYYZ serves as both the input and output point for the TMR flight controller. PFCNode runs on all three SBC's and this is indicated in the figure by the programs PFCCYYZ, PFCCYEG, and PFCCYYJ. Although all three SBC's run the same program, minor changes had to be made to the code so that the program would send and receive data packets to the correct location.

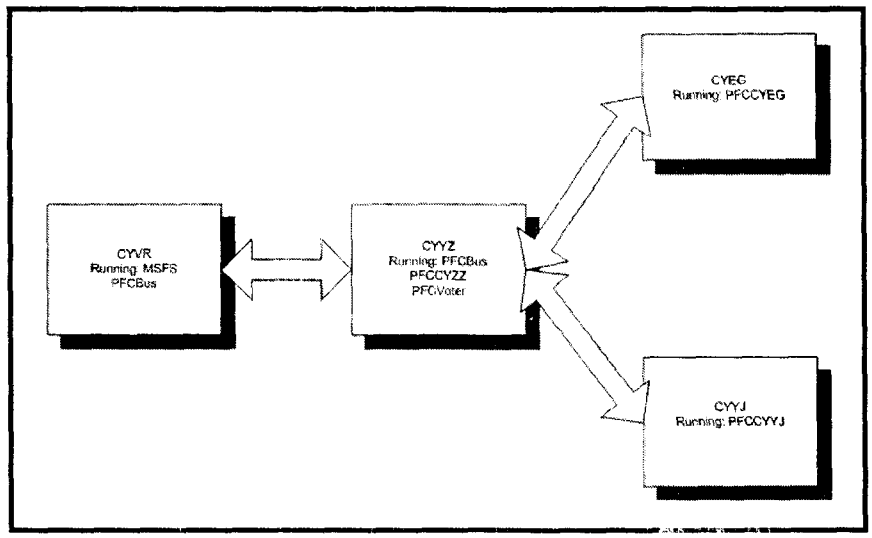
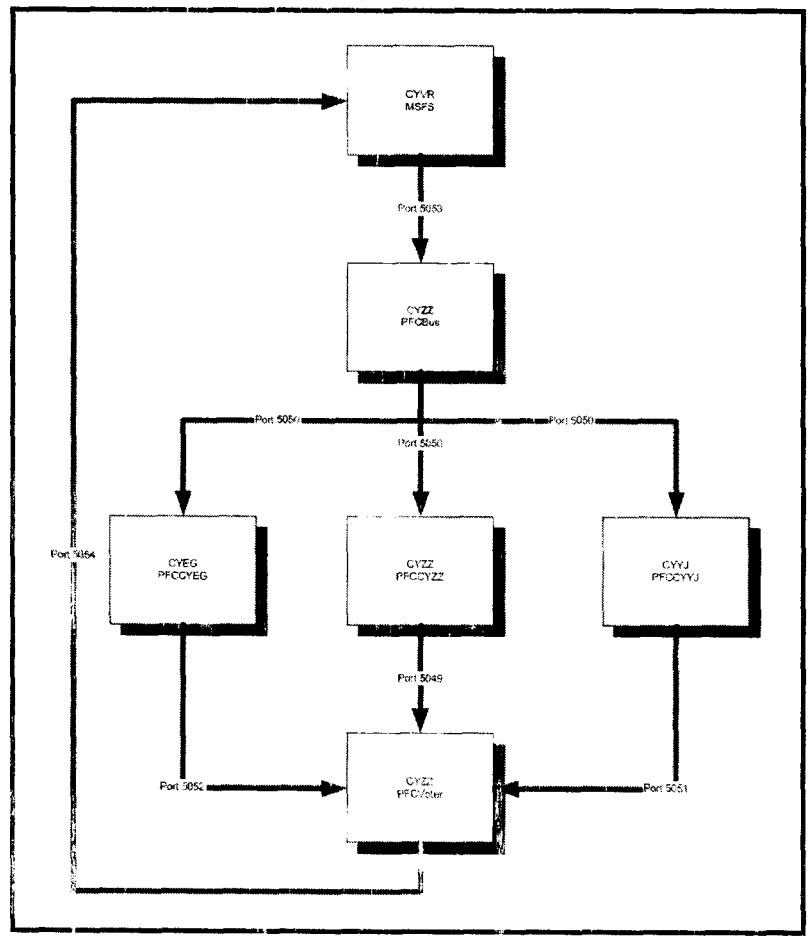


Figure 4-7 Arrangement of Computers and Programs

Figure 4-8 shows the data flow through the system for each incoming packet. Data is taken out of MSFS on CYVR and sent to PFCBus on CYZZ. PFCBus then forwards the data directly to PFCNode running on CYZZ, CYEG and CYYJ at virtually the same time. PFCNode would then create control surface positions to keep the aircraft flying straight and level. These positions would be sent to the PFCVoter running on CYZZ and the positions would be compared. As long as two of the position packets were received and identical, PFCVoter would send the agreed upon result back to CYVR and into MSFS. If, for some reason, the two packets did not agree, PFCVoter would wait for the third packet to come in. If the third packet does not arrive and newer packets begin to arrive the current vote is abandoned and a new one is begun. Under no circumstance will PFCVoter forward on a packet without another packet agreeing with it.

When developing the PFCVoter, care was taken to ensure that the voting process would complete as quickly as possible. A low level memory compare was used to test if one packet was identical to another. This was the most efficient way to compare two complex memory structures. The memory compare was used as little as possible. For example, if the result of the first comparison found two packets were identical, there was no need for further tests and the PFCVoter could complete its current vote.



The source code for all of the programs used in this work is included as appendix C. PFCVoter is the most complex of all the programs. It uses three POSIX Threads to listen for incoming packets from each of the SBC's. A fourth thread checks for received and updated packets and compares them with a MEMCHK call. Mutual exclusion calls are used to keep data from being accessed while another thread is accessing it. Once two packets agree, one of them is sent on to the computer running MSFS.

The compromises we made to implement the system left a critical component. Even though the end result of this work was supposed to be a TMR system, it suffers from a critical dependence on CYYZ as it handles all input, voting and output for the system. Ways to correct this flaw will be discussed in the future work section of the next chapter.

This concludes the discussion on the implementation of the system and the results of the final completed system will now be discussed.

4.2 Results

With the system implemented and operating, our focus moved to testing. It was important to verify that:

- the system would correctly handle any one fault occurring within a PFCNode module;
- the system would operate quickly enough to meet the real time requirement of ten votes per second;
- and the system would handle intermitant faults and allow a failed node to rejoin the voting once it recovers.

Although the CYZZ computer played a critical role in the system, we could still test the correct operation of the TMR system by causing faults to occur in the other two SBC's or within the PFCNode program working on CYZZ.

In order to verify that the system would correctly handle any one fault we performed exhaustive tests on all the possible faults. Figure 4-8 indicates the tests that were performed. For the tests the PFCBus and PFCVoter programs were running on CYZZ. Each row of the figure represents a test, the value of 0 represents an incorrectly working module and a value of 1 represents a correctly working module. It can be seen that for all possibilities, when ever at least two modules are functioning correctly, the TMR System functions correctly because it receives two identical votes.

CYZZ	CYEG	CYYJ	TMR System
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figure 4-8 System Operational Results

Next we had to test wheter the fault tolerant flight control system was successfully able to meet the ten votes per second real time requirement. This test was conducted by configuring the PFCBus program to send ten packets a second from Flight Simulator. The system was setup to be in control of the simulated aircraft and allowed to run for a period of at least ten minutes. This test was conducted over five times. The results were very positive as, with each test, not a single input packet was missed. At the end of each test the number of input packets was equal to the number of output packets. Had the system not met this real time requirement, some input packets would have been dropped. This successful result shows us that a fault tolerant system can operate within the real time requirements.

The final tests we conducted were done to confirm that the system could handle intermittent faults. To run this test we randomly removed one or two PFCNodes from the operating system. This was done by terminating the PFCNode process to simulate the process failing for any number of reasons. As one PFCNode was removed from the system we were able to observe from the PFCVoter output that the system was still meeting the real time requirements and that it was only receiving two inputs (as only two PFCNodes were operating.) We would then restart the failed PFCNode and we observed its inputs being

received by the PFCVoter and its votes being considered for the final systems output. This successful result shows us that the system could handle these intermittent faults.

This chapter has discussed the implementation process and results for the fault tolerant flight control system. The next chapter will discuss the conclusions that can be made from this work and future directions this work can take.

Chapter 5

Conclusions and Future Work

The purpose of this work was to show that it was possible to implement a fault tolerant flight control system using component off the shelf technology. We researched available embedded hardware platforms and chose to use the PC/104 single board computer. We then designed a triple modularly redundant flight control system which controlled a simulated aircraft within the environment of Microsoft Flight Simulator. We tested the system to be sure it was tolerating the expected number of faults and meet the real time requirements. With successful results of the implemented system we have shown that it is possible to develop a fault tolerant flight control system.

5.1 Conclusions

As stated above, the first significant conclusion that was reached from this work is that a triple modular redundant flight control system could be implemented and developed using component off the shelf technology and published information on the topic. Although, at this point, the developed system is by no means ready to be controlling an actual aircraft, it does appear that it could, if proper adoption, testing and certification were carried out.

The PC/104 platform turned out to be an excellent choice for this work. The small embedded boards have more computing power than the other embedded options we considered. The platform also is fairly easy to work with for people who are already familiar with a standard PC x86 system. As discussed in Chapter 4, we encountered some unexpected

problems when working with this platform. We believe that these were attributable to the specific supplier rather than the platform itself.

We have also been able to conclude that Microsoft Flight Simulator ended up being a great platform for simulating an aircraft and testing our flight control system. For the purposes of this work we only touched on the available features within the simulation but still achieved the results we were looking for. This program could very well be used as a test platform for a variety of control system and real time system research or teaching.

Although we did not produce a Byzantine resilient system this work leads us to conclude that implementing such a system is achievable for work at this scale. In our particular case we did not have a sufficient number of single board computers but our implemented design could be adapted to tolerate Byzantine faults. To do this our minimum requirement would be four SBC systems, each having three network ports, and we would have to redesign and implement a new voter module.

Finally, we are able to conclude that designing a system to tolerate common mode faults is a very difficult task. The nature of this type of fault makes it very difficult to predict and to design a system to avoid them. Unlike the Byzantine resilience, we feel that this work could not be easily adapted to handle common mode faults.

5.2 Future Work

This final section of the thesis discusses possible directions for expansion and exploration of this work.

Without changing the architecture of the current TMR flight control system this work could be expanded to be actually embedded and tested inside of an application. There may not be much value in the context of computer science research to apply this design to an aircraft similar to the Stingray UAV discussed earlier but these may be of great value to a

researcher in another fields such as control systems, digital systems testing, real time systems testing and aeronautics.

As mentioned in Chapter 4, our design unfortunately left one of the three single board computers as a critical node in our system. If that one node were to fail then the system would fail. The reason this node was critical was that it contained the three Ethernet ports. One direction this work could take would be if it was redone using a different PC/104 single board computer that contained at least three Ethernet ports. This would allow the voter application to exist on all three modules and would remove the critical node, making the system a much truer triple modular redundant system.

This work has shown that a fault tolerant system can be implemented using fairly inexpensive technology. Therefore, we can conclude that it is possible to use fault tolerant architectures in many areas that traditionally do not consider this technology. As a future direction this work could be used to explore other realms that might benefit from a fault tolerant system such as the automobile industry.

Another direction would be to redesign the system to be Byzantine Resilient. In order to tolerate one Byzantine fault the system would need four modules instead of three. This could be done if other hardware was found that contained enough Ethernet ports.

The final direction this work could take would be to explore solutions to the common mode fault problem. Dissimilar, hardware and software configurations could be explored in the hope of finding a solution to the problem that would make a significant contribution to the field.

Bibliography

[1] J. H. Lala and R. E. Harper, "Architectural principles for safety-critical real-time applications," *Proceedings of the IEEE*, vol. 82, pp. 25-40, 1994.

[2] K. Sabbagh, *21st century jet : the making and marketing of the Boeing 777*. New York: Scribner, 1996.

[3] B. W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*. United States of America: Addison-Wesley Publishing Company Inc., 1989.

[4] Y. C. Yeh, "Design considerations in Boeing 777 fly-by-wire computers," presented at High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International, 1998.

[5] NASA, "Exploring aeronautics." Moffett Field, Calif.: Ames Research Center External Affairs Office, Multimedia
R&D Group,, 1998, pp. 1 computer laser optical disc.

[6] C. Favre, "Fly-by-wire for commercial aircraft: the Airbus experience," *International Journal of Control*, vol. 59, pp. 139-57, 1994.

[7] C. E. Hall, Jr., "A Real-Time Linux system for autonomous navigation and flight attitude control of an uninhabited aerial vehicle.," presented at Digital Avionics Systems Conference, 2001.

[8] Y. C. Yeh, "Safety critical avionics for the 777 primary flight controls system," presented at Digital Avionics Systems, 2001. DASC. The 20th Conference, 2001.

[9] D. P. Siewiorek and R. S. Swarz, *Reliable Computer Systems: Design and Evaluation*, 3 ed. United States of America: A K Peters, Ltd, 1998.

[10] N. G. Leveson, *Safeware: System Safety and Computers*. United States of America: Addison-Wesley Publishing Company, Inc., 1995.

[11] N. Storey, *Safety-Critical Computer Systems*. England: Addison Wesley Longman Limited, 1996.

[12] V. Neufeldt, "Webster's New World Dictionary." United States of America: Warner Books, Inc., 1990, pp. 694.

[13] P. G. Neumann, *Computer-Related Risks*. United States of America: The ACM Press, 1995.

[14] D. K. Pradhan, *Fault-Tolerant Computer System Design*. United States of America: Prentice Hall PTR, 1996.

[15] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, pp. 382--401, 1982.

[16] P. Koopman, "Embedded system design issues (the rest of the story)," presented at Computer Design: VLSI in Computers and Processors, 1996. ICCD '96. Proceedings., 1996 IEEE International Conference on, 1996.

[17] C. Spitzer, "Modern Avionics Architectures," Tutorial Session - 21st Digital Avionics Systems Conference, 2002.

[18] "Tri-M Systems and Engineering MZ104+ v3 Website," vol. 2004. Port Coquitlam: Tri-M Systems and Engineering http://www.tri-m.com/products/engineering/mz104+.html, 2003.

[19] K. Li, "Personal Communication - Course work discusions related to Software Engineering 540 - Embedded Systems," C. B. Feldstein, Ed. Victoria, 2003.

[20] M. Predko, "Your first PICMicro Project Website," vol. 2004: www.rentron.com, 1999.

[21] J.-L. Padiolleau, "Site génie électronique de l'Académie d'Orléans - Tours - Microconroleur Website," vol. 2004. France: http://www.ac-orleans-tours.fr/sti-gel/MICROCONTROLEUR/Accueil_Micro.htm, 2004.

[22] "PC/104 Embedded PC Modules Website," vol. 2002: The PC/104 Consortium www.pc104.org, 2002.

[23] P. Dowson, "Software by Peter Dowson," vol. 2004: http://www.schiratti.com/dowson.html, 2004.

[24] K. Yaghmour, Building Embedded Linux Systems. United States of America: O'Reilly & Associates Inc., 2003.

[25] M. R. Napolitano, G. Molinaro, M. Innocenti, B. Seanor, and D. Martinelli, "A complete hardware package for a fault tolerant flight control," vol. - 4, pp. - 2619 vol.4, 1999.

[26] M. Abramovici, M. A. Breuer, and A. D. Friedman, Digital Systems Testing & Testable Design, 1 ed. United States of America: W. H. Freeman and Company, 1990.

[27] "Integrys Company Website," vol. 2003: http://www.integrys.com/, 2003.

[28] S. Walton, Linux Socket Programming. United States of America: Sams Publishing, 2001.

[29] J. B. Dugan, S. A. Doyle, and F. A. Patterson-Hine, "Simple models of hardware and software fault tolerance," vol. -, pp. - 129, 1994.

[30] W. R. Stevens, UNIX Network Programming. United States of America: Prentice-Hall Inc., 1990

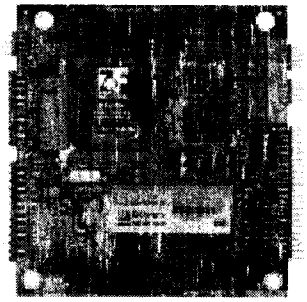
Appendix A

Tri-M Engineering's MZ104+ Spec Sheet

This appendix contains the specification sheet for the MZ104+ single board computer used for this work.

MZ104+

PC/104+ Computer with Dual 10/100 Ethernet Ports



Features

- **Dual Intel 82559ER 10/100 BaseT Ethernet PC/104+ interface**
- **Dual RS-232 serial, dual EIDE, floppy support, dual USB, parallel port**
- **DiskOnChip socket**
- **Dual watchdog timers, Phoenix BIOS and FailSafe Boot ROM**

Specifications

Z5x86, Embedded PC-on-a-Chip
- 32bit CPU core with 33, 66, 100 and 133*MHz operation
- Fully Desktop AT compatible
- BIOS: Full Phoenix PC BIOS
- SDRAM support 8 to 64 MB SO-DIMM
- Floppy disk controller

Fail-safe Boot ROM
- Total system recovery
- Recover easily from the corruption or loss of CPU boot data

Dual Watchdog Timer
- Programmable tickle sources

PC/104 Bus
- Compliant with standard PC/104 expansion bus with an IRQ and DMA channel subset

Serial Ports
- Two 16550-compatible serial ports, RS232

Parallel Ports
- One bi-directional AT-compatible printer port

USB Port
- One USB root hub interface

PC/104+ Bus
- Compliant with standard PC/104+ expansion bus

USB Port
- Two fully independent USB interfaces

* 133MHz is overclocking (additional cooling required)

EIDE Hard Drive Interface
- 44 pin standard pin header
- Supports up to two EIDE drives (master/slave)

Solid State Flash Memory Device
- Supports M-Systems DiskOnChip Millennium and DiskOnChip 2000 - 8MB to 1GB

Ethernet Controller
- 2x 10/100BaseT (twisted-pair)
- IEEE and Ethernet standards
- LINK LED verification
- Supports Microsoft's Plug and Play System
- Driver support includes popular OS including Novell Netware and Microsoft Windows 95

Software Compatibility
- Phoenix embedded PC BIOS - 100% X86 compatible
- MSDOS 3.X, 4.X, 5.X, 6.X, DRDOS, Win95/98/NT, Linux, most PC compatible RTOSes (Wind River VxWorks RTOS with browser)

Electrical Specifications
- Single 5V DC Operation

Mechanical/Environmental
- Size: 3.55" x 3.775" x 0.9"
- Operating temperature: 133*MHz @ -20°C to 70°C
- 33MHz, 66MHz, 100MHz @ -40°C to 85°C

Ordering Information

Part Number	Description
*MZ104+	*PC/104+ Computer with dual Ethernet
Options	
*MZ104+ KIT	*MZ104+ Starter Kit
*CABLESET2-MZ104+	*Cabling solution
*MEM-8MB-MZ	*8MB SDRAM SO-DIMM for MZ104 series
*MEM-16MB-MZ	*16MB SDRAM SO-DIMM for MZ104 series
*MEM-32MB-MZ	*32MB SDRAM SO-DIMM for MZ104 series
*MEM-64MB-MZ	*64MB SDRAM SO-DIMM for MZ104 series

Tri-M Systems and Engineering
1407 Kebet Way, Unit 108 • Port Coquitlam, BC V3C 6L3 • Canada
Tel: 604.945.9565 Fax: 604.945.9566 www.Tri-M.com

Appendix B

Source Code - PFCBus.h

The PFCBus.h file was used to define the communication data packets to be used for this work. This file was originally written by Sean Mcleod but was modified because the original SequenceNo was declared to be a unsigned int. When this code was compiled in both Visual Studio and Linux we found that the different platforms had a different bit size for the unsigned int data type. By using an unsigned long long data type we were able to avoid having to use different settings for each platform.

```
#ifndef _PFC_BUS_H_
#define _PFC_BUS_H_

typedef struct {
        unsigned long long      SequenceNo;
        double                  Pitch;          // degrees +pitch up
        double                  Roll;           // degrees +roll right
        double                  Heading;        // degrees
        double                  Altitude;       // m above sea level
        double                  Airspeed;       // IAS - m/s
} AircraftAttitude;

typedef struct {
        unsigned long long      SequenceNo;
        double                  Elevator;       // +- 16383
        double                  Aileron;        // +- 16383
        double                  Rudder;         // +- 16383
        double                  Throttle;       // -4096 to +16383 (-#'s= reverse thrust)
        double                  ElevatorTrim;   // +- 16383
} AircraftFlightSurfaces;

#endif
```

Appendix C

Source Code - PFCBus.cpp

The PFCBus.cpp program was written by Sean McLeod. It is to run on the same computer as Microsoft Flight Simulator. This program uses the FSUIPC utility to extract flight data from the running simulation. This program also listens on a particular UDP data port for incoming data packets, which it receives and sends back into the simulation.

```
// This program written by Sean McLeod

#include "stdafx.h"
#include "PFCBus.h"
#include "FSUIPC_User.h"

BOOL CtrlHandler(DWORD fdwCtrlType);
void Usage();

DWORD WINAPI FlightSurfacesListenerThread(LPVOID);
DWORD WINAPI AttitudeBroadcasterThread(LPVOID);

DWORD hz = 10;
CRITICAL_SECTION g_FSUIPC_CS;

int main(int argc, char* argv[])
{
	if(argc > 1)
	{
		if(!strcmpi("-h", argv[1]))
			return Usage();

		if(!strcmpi("-hz", argv[1]) && argc > 2)
		{
			hz = atoi(argv[2]);
		}
	}

	SetConsoleCtrlHandler((PHANDLER_ROUTINE) CtrlHandler, TRUE);

	cout << "PFCBus running at " << hz << "Hz" << endl;

	WORD wVersionRequested = MAKEWORD(2, 2);
	WSADATA wsaData;

	if(WSAStartup( wVersionRequested, &wsaData ) != 0)
	{
		cerr << "WSAStartup failed." << endl;
		WSACleanup();
		return -1;
	}

	// Connect to MSFS
	DWORD dwOpenResult = 0;
	if(!FSUIPC_Open(SIM_ANY, &dwOpenResult))
	{
		cerr << "FSUIPC_Open failed, error code: " << dwOpenResult << endl;
		return -2;
	}

	// Create critical section since we're not sure if FSUIPC is thread safe
	InitializeCriticalSection(&g_FSUIPC_CS);

	// Create threads
	CreateThread(NULL, 0, AttitudeBroadcasterThread, NULL, 0, NULL);
	CreateThread(NULL, 0, FlightSurfacesListenerThread, NULL, 0, NULL);

	// Will never return, the ctrl-c handler will terminate us
	Sleep(INFINITE);

	FSUIPC_Close();
	WSACleanup();

	return 0;
}

void Usage()
```

```
{
        cout << "PFCBus.exe -h | [-hz rate [10] ]" << endl;
}

BOOL CtrlHandler(DWORD fdwCtrlType)
{
    switch (fdwCtrlType)
    {
        case CTRL_C_EVENT:
        case CTRL_CLOSE_EVENT:
        case CTRL_BREAK_EVENT:
        case CTRL_LOGOFF_EVENT:
        case CTRL_SHUTDOWN_EVENT:
        default:
                        FSUIPC_Close();
                        WSACleanup();
            return FALSE;
    }
}

DWORD WINAPI FlightSurfacesListenerThread(LPVOID)
{
        SOCKET s = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);

        sockaddr_in ip_destination = { 0 };
        ip_destination.sin_family = AF_INET;
        ip_destination.sin_port = 5054;
        in_addr addr = { 0 };
        addr.S_un.S_addr = INADDR_ANY;
        ip_destination.sin_addr = addr;

        bind(s, (sockaddr*)&ip_destination, sizeof(ip_destination));

        AircraftFlightSurfaces data;
        DWORD dwResult;

        // TODO Discard old sequences if we get any out of order
        while(1)
        {
                memset(&data, 0, sizeof(data));
                recvfrom(s, (char*)&data, sizeof(data), 0, NULL, NULL);
                printf("got one! \n");
                EnterCriticalSection(&g_FSUIPC_CS);

                // Disconnect joystick and also sync all throttles
                char disconnect = 0x20;
                FSUIPC_Write(0x310A, 1, &disconnect, &dwResult);

                short elevatorPos = (short)data.Elevator;
                FSUIPC_Write(0x0BB2, 2, &elevatorPos, &dwResult);

                short aileronPos = (short)data.Aileron;
                FSUIPC_Write(0x0BB6, 2, &aileronPos, &dwResult);

                short rudderPos = (short)data.Rudder;
                FSUIPC_Write(0x0BBA, 2, &rudderPos, &dwResult);

                short elevatorTrimPos = (short)data.ElevatorTrim;
                FSUIPC_Write(0x0BC0, 2, &elevatorTrimPos, &dwResult);

                short throttlePos = (short)data.Throttle;
                FSUIPC_Write(0x088C, 2, &throttlePos, &dwResult);

                FSUIPC_Process(&dwResult);

                LeaveCriticalSection(&g_FSUIPC_CS);
        }

        return 0;
}
```

```
DWORD WINAPI AttitudeBroadcasterThread(LPVOID)
{
	SOCKET s = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);

	BOOL bAllowBroadcasts = TRUE;
	setsockopt(s, SOL_SOCKET, SO_BROADCAST, (const char*)&bAllowBroadcasts,
sizeof(bAllowBroadcasts));

	sockaddr_in ip_destination = { 0 };
	ip_destination.sin_family = AF_INET;
	ip_destination.sin_port = 5053;
	in_addr addr = { 0 };
	addr.S_un.S_addr = INADDR_BROADCAST;
	ip_destination.sin_addr = addr;

	DWORD dwSequenceNo = 0;
	while(1)
	{
		EnterCriticalSection(&g_FSUIPC_CS);

		DWORD dwResult = 0;

		DWORD dwTrueHeading = 0;
		FSUIPC_Read(0x580, 4, &dwTrueHeading, &dwResult);

		long dwPitch = 0;
		FSUIPC_Read(0x0578, 4, &dwPitch, &dwResult);

		long dwBank = 0;
		FSUIPC_Read(0x057C, 4, &dwBank, &dwResult);

		long qwAltitude[2] = {0};
		FSUIPC_Read(0x570, 8, &qwAltitude[0], &dwResult);

		long dwIAS = 0;
		FSUIPC_Read(0x2BC, 4, &dwIAS, &dwResult);

		FSUIPC_Process(&dwResult);

		LeaveCriticalSection(&g_FSUIPC_CS);

		AircraftAttitude data = { 0 };
		data.SequenceNo = dwSequenceNo++;

		data.Heading = (dwTrueHeading*360.0)/(65536.0*65536.0);
		data.Pitch = -(dwPitch*360.0)/(65536.0*65536.0);
		data.Roll = -(dwBank*360.0)/(65536.0*65536.0);
		data.Altitude = qwAltitude[1];
		data.Airspeed = (dwIAS / 128.0)*1852.0/(60.0*60.0);

		//printf("data size: %d \n", sizeof(data));

		sendto(s, (const char*)&data, sizeof(data), 0,
(sockaddr*)&ip_destination, sizeof(ip_destination));

		Sleep(1000/hz);
	}

	return 0;
}
```

Appendix D

Source Code - PFCBus.c

The PFCBus.c program was written by Cary Feldstein. This program is responsible for receiving data packets from the PFCBus.cpp program running on another computer. This program will then forward on the packets to the three nodes.

```
// PFCBus receive program
// This program is responsible for receiving network packets from the flight sim
program
// and then forwarding them along to all three PFCNodes.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

#include "PFCBus.h"

#define SOCK_PREF "/tmp/#"

int receive_socket, send_socket1, send_socket2, send_socket3, Seq_No, address_size,
bytes;
char rec[44];
AircraftAttitude data;

struct sockaddr_in receive_address, send_address1, send_address2, send_address3;

void Receive_data_Setup ()
//Setup to receive flight sim data
{

    int errno;

    receive_socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);
    receive_address.sin_family = AF_INET;
    receive_address.sin_port = 5053;
    receive_address.sin_addr.s_addr = INADDR_ANY;

    if (errno=bind(receive_socket, (struct sockaddr *) &receive_address,
sizeof(receive_address)) != 0 )
        printf("Bind Receive Error %d \n" , errno);
}


void Send_data_Setup_1 ()
// Send to CYEG
{
    int errno;
    send_socket1 = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);
    send_address1.sin_family = AF_INET;
    send_address1.sin_port = 5050;

    if ( inet_aton("192.168.1.1", &send_address1.sin_addr) == 0)
            printf("address error");

    if (errno=bind(send_socket1, (struct sockaddr *) &send_address1,
sizeof(send_address1)) != 0 )
        printf("Bind Send Error 1: %d\n", errno);

}

void Send_data_Setup_2 ()
// Send to CYYJ
{
    int errno;
    send_socket2 = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);
    send_address2.sin_family = AF_INET;
    send_address2.sin_port = 5050;

    if ( inet_aton("192.168.1.11", &send_address2.sin_addr) == 0)
            printf("address error");
```

```
    if (errno=bind(send_socket2, (struct sockaddr *) &send_address2,
sizeof(send_address2)) != 0 )
        printf("Bind Send Error 2: %d\n", errno);

}

void Send_data_Setup_3 ()
// Send to CYZZ
{
    send_socket3 = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);
    send_address3.sin_family = AF_INET;
    send_address3.sin_port = 5050;

    if ( inet_aton("127.0.0.1", &send_address3.sin_addr) == 0)
            printf("address error");

    if (bind(send_socket3, (struct sockaddr *) &send_address3, sizeof(send_address3))
!= 0 )
        printf("Bind Send Error 3\n");

}

void Print_AircraftAttitude (AircraftAttitude data)
{

    printf("Seq. Num = %u \n", data.SequenceNo);
    printf("Pitch = %f \n", data.Pitch);
    printf("Roll = %f \n", (double) data.Roll);
    printf("Heading = %f \n", (double) data.Heading);
    printf("Altitude = %f \n", (double) data.Altitude);
    printf("Airspeed = %f \n", (double) data.Airspeed);

}
AircraftAttitude Receive_data ()
// Receives a network UDP packet
{
    memset (&receive_address, 0, sizeof(receive_address));
    address_size = sizeof(receive_address);
    bytes = recvfrom(receive_socket, (char*)&data, sizeof(data), 0, &receive_address,
&address_size);
    //printf ("Received data packet size %d.\n", sizeof(data));
//    reply++;

    //Print_AircraftAttitude (data);

    return data;
}



void Send_Data (AircraftAttitude data)
// Sends the same data to three different sockets
{

    sendto(send_socket1, (const char*)&data, sizeof(data), 0, (struct sockaddr
*)&send_address1, sizeof(send_address1));

    sendto(send_socket2, (const char*)&data, sizeof(data), 0, (struct sockaddr
*)&send_address2, sizeof(send_address2));


    sendto(send_socket3, (const char*)&data, sizeof(data), 0, (struct sockaddr
*)&send_address3, sizeof(send_address3));


}


int main(int argc, char* argv[])
{
```

```
    //reply =0;
    Seq_No =0;
    printf("PFCBus UDP Packet Test Receive Program\n");


    Receive_data_Setup();
    Send_data_Setup_1();
    Send_data_Setup_2();
    Send_data_Setup_3();

    while (1) {

        //printf("receiving\n");

        memset (&data, 0, sizeof(data));

        //Await message
        data = Receive_data();
        Send_Data (data);

        //memset (&surfaces, 0, sizeof(surfaces));
    }

    close(send_socket1);
    close(send_socket2);
    close(send_socket3);
    close(receive_socket);
    return 0;

}
```

Appendix E

Source Code - PFCNode.c

The PFCNode.c program was written by Cary Feldstein. It is the actual flight controller. This program receives flight data from the PFCBus.c program and then decides on particular flight control surface positions to correctly control the aircraft. Those positions are then put into a different data packet and sent to the PFCVoter.c program.

```
// PFCBus Node program
// Responsible for converting flight data packets into control surface packets

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>


#include "PFCBus.h"

#define SOCK_PREF "/tmp/#"
#define PORT_DEF 5052
#define IP_ADDR "192.168.1.20"

int sr, ss, Seq_No, reply, address_size, bytes;
char rec[44];
int bAllowBroadcasts = 1;
AircraftAttitude data;
AircraftFlightSurfaces surfaces;

struct sockaddr_in raddr, saddr;

void Receive_data_Setup ()
// Sets up receiver socket
{

    sr = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);
    raddr.sin_family = AF_INET;
    raddr.sin_port = 5050;
    raddr.sin_addr.s_addr = INADDR_ANY;

    if (bind(sr, (struct sockaddr *) &raddr, sizeof(raddr)) != 0 )
        printf("Bind Receive Error \n");
        //memset (&data, 0, sizeof(data));
}


void Send_data_Setup ()
// Sets up sender socket
{
    ss = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);
    //setsockopt (ss, SOL_SOCKET, SO_BROADCAST, (const char*)&bAllowBroadcasts,
sizeof(bAllowBroadcasts));

    saddr.sin_family = AF_INET;
    saddr.sin_port = PORT_DEF;
    //saddr.sin_addr.s_addr = INADDR_ANY;

    if ( inet_aton(IP_ADDR, &saddr.sin_addr) == 0)
                    printf("address error");

    if (bind(ss, (struct sockaddr *) &saddr, sizeof(saddr)) != 0 )
        printf("Bind Send Error \n");

    memset (&surfaces, 0, sizeof(surfaces));

}

void Print_AircraftAttitude (AircraftAttitude data)
{

    printf("Seq. Num = %u \n", data.SequenceNo);
    printf("Pitch = %f \n", data.Pitch);
    printf("Roll = %f \n", (double) data.Roll);
```

```
    printf("Heading = %f \n", (double) data.Heading);
    printf("Altitude = %f \n", (double) data.Altitude);
    printf("Airspeed = %f \n", (double) data.Airspeed);

}
AircraftAttitude Receive_data ()
{
    memset (&raddr, 0, sizeof(raddr));
    address_size = sizeof(raddr);
    bytes = recvfrom(sr, (char*)&data, sizeof(data), 0, &raddr, &address_size);
    //printf ("Received data packet #%d of size %d.\n", reply, sizeof(data));
    reply++;

    //Print_AircraftAttitude (data);

    return data;
}


AircraftFlightSurfaces PFC_Process (AircraftAttitude data)
{

    surfaces.SequenceNo = data.SequenceNo;

    //Process Pitch
    if (data.Pitch > 1.0)
    {
       surfaces.Elevator = -2000;
    }
    else if (data.Pitch < -1.0)
    {
       surfaces.Elevator = 2000;
    }
    else surfaces.Elevator =0;

    if (data.Roll > 1.0)
    {
       surfaces.Aileron = -10000;
    }
    else if (data.Roll < -1.0)
    {
       surfaces.Aileron = 10000;
    }
    else surfaces.Aileron =0;


    surfaces.Rudder = 200;

    if (data.Altitude > 510)
    {
       surfaces.Throttle = 6000;
    }
    else if (data.Altitude < 480)
    {
       surfaces.Throttle = 16383;
    }
    else surfaces.Throttle =12000;
    //surfaces.Throttle = 16383;
    surfaces.ElevatorTrim = 0;

    return surfaces;

}

void Send_Surfaces (AircraftFlightSurfaces surface)
{
        //int size = sizeof(surface);
        //printf("size 12 = %d\n", size);
```

```
    sendto(ss, (const char*)&surface, sizeof(surface), 0, (struct sockaddr *)&saddr,
sizeof(saddr));
        //printf("size 13 = %d\n", sizeof(surface));
        //printf("Sent data packet #%d of size %d\n", (unsigned int)surface.SequenceNo,
sizeof(surface));
        //printf("size 14 = %d\n", size);
}


int main(int argc, char* argv[])
{
    reply =0;
    Seq_No =0;
    printf("PFCBus UDP Packet Test Receive Program\n");
        //printf("size 1 = %d\n", sizeof(surfaces));


    Receive_data_Setup();
        //printf("size 2 = %d\n", sizeof(surfaces));
    Send_data_Setup();
        //printf("size 3 = %d\n", sizeof(surfaces));

    //bzero(&addr, sizeof(addr));



//30


    memset (&data, 0, sizeof(data));
        //printf("size 4 = %d\n", sizeof(surfaces));
    memset (&surfaces, 0, sizeof(surfaces));

        //printf("size 5 = %d\n", sizeof(surfaces));

    while (1) {

        //printf("receiving\n");

                memset (&data, 0, sizeof(data));
                //printf("size 6 = %d\n", sizeof(surfaces));
        memset (&surfaces, 0, sizeof(surfaces));
                //printf("size 7 = %d\n", sizeof(surfaces));

        //Await message

        data = Receive_data();
                //printf("size 8 = %d\n", sizeof(surfaces));
        surfaces = PFC_Process (data);
                //printf("size 9 = %d\n", sizeof(surfaces));
        Send_Surfaces (surfaces);
                //printf("size 10 = %d\n", sizeof(surfaces));


        memset (&surfaces, 0, sizeof(surfaces));

                //printf("size 11 = %d\n", sizeof(surfaces));




    }

    close(ss);
    close(sr);
```

```
    return 0;
}
```

Appendix F

Source Code - PFCVoter.c

The PFCVoter.c program was written by Cary Feldstein. This program would listen for and receive incoming control surface packets from three PFCNodes. Once two packets are received it is possible for a majority vote would reach an agreement. If this vote succeeds then there is no need to wait for the third packet to arrive. If it does not succeed then there is a need to wait for the third packet. Once an agreement is made, the agreed upon values are sent back to Microsoft Flight Simulator via the PFCBus.cpp program.

```
// PFCVoter receive vote send program

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <pthread.h>

#include "PFCBus.h"

#define SOCK_PREF "/tmp/#"
#define CYEG_PORT 5052
#define CYYJ_PORT 5051
#define CYZZ_PORT 5049

int socket_receive_CYEG, socket_receive_CYYJ, socket_receive_CYZZ, socket_send;
int reply, address_size;
long long last_sent_Seq_No;
int bytes_CYEG, bytes_CYYJ, bytes_CYZZ;
char rec[44];
int bAllowBroadcasts = 1;

//MUTEX Variables!
AircraftFlightSurfaces data_CYEG, data_CYYJ, data_CYZZ;
AircraftFlightSurfaces surfaces;

pthread_mutex_t mutex_CYEG = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_CYYJ = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_CYZZ = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_surfaces = PTHREAD_MUTEX_INITIALIZER;

struct sockaddr_in receive_addr_CYEG, receive_addr_CYYJ, receive_addr_CYZZ, send_addr;

void Receive_data_CYEG_Setup ()
// Setup the CYEG socket and initialize the CYEG data
{

    socket_receive_CYEG = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);
    receive_addr_CYEG.sin_family = AF_INET;
    receive_addr_CYEG.sin_port = CYEG_PORT;
    receive_addr_CYEG.sin_addr.s_addr = INADDR_ANY;

    if (bind(socket_receive_CYEG, (struct sockaddr *) &receive_addr_CYEG,
sizeof(receive_addr_CYEG)) == -1 )
        printf("Bind Receive Error \n");


        pthread_mutex_lock( &mutex_CYEG );

        memset (&data_CYEG, 0, sizeof(data_CYEG));
        data_CYEG.SequenceNo = -1;

        pthread_mutex_unlock( &mutex_CYEG );

}


void Receive_data_CYYJ_Setup ()
// Setup the CYYJ socket and initialize the CYYJ data
{

    socket_receive_CYYJ = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);
    receive_addr_CYYJ.sin_family = AF_INET;
    receive_addr_CYYJ.sin_port = CYYJ_PORT;
```

```
    receive_addr_CYYJ.sin_addr.s_addr = INADDR_ANY;

    if (bind(socket_receive_CYYJ, (struct sockaddr *) &receive_addr_CYYJ,
sizeof(receive_addr_CYYJ)) == -1 )
        printf("Bind Receive Error \n");

        pthread_mutex_lock( &mutex_CYYJ );

        memset (&data_CYYJ, 0, sizeof(data_CYYJ));
        data_CYYJ.SequenceNo = -1;

        pthread_mutex_unlock( &mutex_CYYJ );
}


void Receive_data_CYZZ_Setup ()
// Setup the CYZZ socket and initialize the CYZZ data
{

    socket_receive_CYZZ = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);
    receive_addr_CYZZ.sin_family = AF_INET;
    receive_addr_CYZZ.sin_port = CYZZ_PORT;
    receive_addr_CYZZ.sin_addr.s_addr = INADDR_ANY;

    if (bind(socket_receive_CYZZ, (struct sockaddr *) &receive_addr_CYZZ,
sizeof(receive_addr_CYZZ)) == -1 )
        printf("Bind Receive Error \n");

        pthread_mutex_lock( &mutex_CYZZ );

        memset (&data_CYZZ, 0, sizeof(data_CYZZ));
        data_CYZZ.SequenceNo = -1;

        pthread_mutex_unlock( &mutex_CYZZ );
}



void Send_data_Setup ()
// Setup the send data socket which returns the final voted on packet
{
    socket_send = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);

    send_addr.sin_family = AF_INET;
    send_addr.sin_port = 5054;

    if ( inet_aton("142.104.107.34", &send_addr.sin_addr) == 0)
            printf("address error");

    if (bind(socket_send, (struct sockaddr *) &send_addr, sizeof(send_addr)) == -1 )
        printf("Bind Send Error \n");

    pthread_mutex_lock( &mutex_surfaces );

        memset (&surfaces, 0, sizeof(surfaces));
        //surfaces.SequenceNo = -1;
        last_sent_Seq_No = -1;

        pthread_mutex_unlock( &mutex_surfaces );

}

void Print_AircraftSurface (AircraftFlightSurfaces data)
{

    printf("Seq. Num = %u \n", data.SequenceNo);
    printf("Elevator = %f \n", data.Elevator);
    printf("Aileron = %f \n", (double) data.Aileron);
    printf("Rudder = %f \n", (double) data.Rudder);
    printf("throttle = %f \n", (double) data.Throttle);
```

```
    printf("elevator trim = %f \n", (double) data.ElevatorTrim);
}

int Vote (AircraftFlightSurfaces data1, AircraftFlightSurfaces data2,
AircraftFlightSurfaces data3)
//Implementation of the Vote Scheme. Returns 0 if no vote can take place.
// Returns 1, 2 or 3 indicating which of the above data inputs can be used as a
majority vote
// Must be called with all 3 incoming data structs locked.
{
        if ( (int)last_sent_Seq_No < (int)data1.SequenceNo ) {
                // we know data1 is a current packet

                if ( (int)last_sent_Seq_No < (int)data2.SequenceNo ) {
                        // then we can vote using 1 and 2

                        if ((memcmp (&data1, &data2, sizeof (data1))) == 0) return 1;
                        // if 1 and 2 are the same then we've got a majority and we're
done

                        else if ( (int)last_sent_Seq_No < (int)data3.SequenceNo ) {
                                // since 1 and 2 aren't the same, see if we can use 3

                                if ((memcmp (&data1, &data3, sizeof (data1))) == 0)
return 1;
                                // if 1 and 3 are the same then we've got a majority and
we're done.

                                else if ((memcmp (&data2, &data3, sizeof (data2))) == 0)
return 2;
                                // if 2 and 3 are the same then we've got a majority and
we're done.

                                else return 0;
                                // can't vote so return 0
                        }

                        else return 0;
                                // we can't use 3 so 1 and 2 don't agree and 3 isn't up
to date.
                                // we have no majority, so send dummy response to
indicate no vote

                }

                else if ( (int)last_sent_Seq_No < (int)data3.SequenceNo ) {
                                // since 2 isn't up to date see if we can use 3

                        if ((memcmp (&data1, &data3, sizeof (data1))) == 0) return 1;
                                // if 1 and 3 are the same then we've got a majority and
we're done.

                        else return 0;
                        // can't vote so return 0
                }

                else return 0;
                //we can't use 2 or 3 so we can't vote

        }

        else if ( (int)last_sent_Seq_No < (int)data2.SequenceNo ) {
                // can't use 1 so see if we can use 2

                if ( (int)last_sent_Seq_No < (int)data3.SequenceNo ) {
                        // since we can use 2, see if 3 useable too

                        if ((memcmp (&data2, &data3, sizeof (data2))) == 0) return 2;
                                // if 2 and 3 are the same then we've got a majority and
we're done.
```

```
                        else return 0;
                                // can't vote so return 0
                }

                else return 0;

        }

        else return 0;

}

//TODO after this thread is complete it needs to be replicated twice for the other
//two computers.
void *Receive_data_CYEG (void *arg)
{

        AircraftFlightSurfaces data_temp;

        while (1) {

                //struct argstruct *myarg = arg;

                //printf("receiving from CYEG\n");
                address_size = sizeof(receive_addr_CYEG);
                bytes_CYEG = recvfrom(socket_receive_CYEG, (char*)&data_temp,
sizeof(data_temp), 0, &receive_addr_CYEG, address_size);
                printf ("Received data packet #%d of size %d, from CYEG\n", (unsigned
int)data_temp.SequenceNo, sizeof(data_temp));

                pthread_mutex_lock( &mutex_CYEG );

                data_CYEG = data_temp;
                //printf("updated cyeg\n");

                pthread_mutex_unlock( &mutex_CYEG );
        }

    return NULL;
}

void *Receive_data_CYYJ (void *arg)
{

        AircraftFlightSurfaces data_temp;

        while (1) {

                //struct argstruct *myarg = arg;

                //printf("receiving from CYEG\n");
                bytes_CYYJ = recvfrom(socket_receive_CYYJ, (char*)&data_temp,
sizeof(data_temp), 0, &receive_addr_CYYJ, sizeof(receive_addr_CYYJ));
                printf ("Received data packet #%d of size %d, from CYYJ\n", (unsigned
int)data_temp.SequenceNo, sizeof(data_temp));

                pthread_mutex_lock( &mutex_CYYJ );

                data_CYYJ = data_temp;

                pthread_mutex_unlock( &mutex_CYYJ );
        }

    return NULL;
}
void *Receive_data_CYZZ (void *arg)
{
        AircraftFlightSurfaces data_temp;
```

```
        while (1) {

                //struct argstruct *myarg = arg;

                //printf("receiving from CYEG\n");
                bytes_CYZZ = recvfrom(socket_receive_CYZZ, (char*)&data_temp,
sizeof(data_temp), 0, &receive_addr_CYZZ, sizeof(receive_addr_CYZZ));
                printf ("Received data packet #%d of size %d, from CYZZ\n", (unsigned
int)data_temp.SequenceNo, sizeof(data_temp));
                //Print_AircraftSurface (data_temp);
                pthread_mutex_lock( &mutex_CYZZ );

                data_CYZZ = data_temp;

                pthread_mutex_unlock( &mutex_CYZZ );
        }

    return NULL;
}



void Send_Surfaces (AircraftFlightSurfaces surfaces)
{

    sendto(socket_send, (const char*)&surfaces, sizeof(surfaces), 0, (struct sockaddr
*)&send_addr, sizeof(send_addr));
        last_sent_Seq_No = surfaces.SequenceNo;
        //printf("Sent packet number: %d\n", last_sent_Seq_No);

}


int main(int argc, char* argv[])
{
        int vote;
        pthread_t tchild_CYEG, tchild_CYYJ, tchild_CYZZ;

    printf("PFCBus Voter Program\n");

        //Setup Sockets
        Receive_data_CYEG_Setup();
        Receive_data_CYYJ_Setup();
        Receive_data_CYZZ_Setup();
    Send_data_Setup();


        //Start up listener threads
        if (pthread_create(&tchild_CYEG, NULL, &Receive_data_CYEG, NULL) != 0)
                perror("PThreads error");

        if (pthread_create(&tchild_CYYJ, NULL, &Receive_data_CYYJ, NULL) != 0)
                perror("PThreads error");

        if (pthread_create(&tchild_CYZZ, NULL, &Receive_data_CYZZ, NULL) != 0)
                perror("PThreads error");


        //Main loop
    while (1) {

                // printf("receiving\n");

                //memset (&data_CYEG, 0, sizeof(data_CYEG));
         //memset (&rec, 0, sizeof(rec));

         //Await message
         //printf("receiving2\n");
         //data = Receive_data();
```

```
        // printf("received data\n");

                pthread_mutex_lock( &mutex_CYEG );
                pthread_mutex_lock( &mutex_CYYJ );
                pthread_mutex_lock( &mutex_CYZZ );
                //printf("Last packet number: %d\n", last_sent_Seq_No);
                //printf("Current packet number: %d\n", data_CYZZ.SequenceNo);

                //TODO add majority vote logic! then you're done!

                vote = Vote (data_CYEG, data_CYYJ, data_CYZZ);

                switch (vote) {

                        case 1: Send_Surfaces (data_CYEG);
                                        printf("Sending CYEG\n");
                                        break;
                        case 2: Send_Surfaces (data_CYYJ);
                                        printf("Sending CYYJ\n");
                                        break;
                        case 3: Send_Surfaces (data_CYZZ);
                                        printf("Sending CYZZ\n");
                                        break;
                        default: break;
                }

                pthread_mutex_unlock( &mutex_CYEG );
                pthread_mutex_unlock( &mutex_CYYJ );
                pthread_mutex_unlock( &mutex_CYZZ );
         //memset (&surfaces, 0, sizeof(surfaces));

    }

    close(socket_send);
    close(socket_receive_CYEG);
        close(socket_receive_CYYJ);
        close(socket_receive_CYZZ);
    return 0;

}
```