

SOFT AFDX (AVIONICS FULL DUPLEX SWITCHED ETHERNET) END  
SYSTEM IMPLEMENTATION WITH STANDARD PC AND ETHERNET CARD

A THESIS SUBMITTED TO  
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES  
OF  
MIDDLE EAST TECHNICAL UNIVERSITY

BY

EMRE ERDİNÇ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR  
THE DEGREE OF MASTER OF SCIENCE  
IN  
ELECTRICAL AND ELECTRONICS ENGINEERING

MAY 2010

Approval of the thesis:

**SOFT AFDX (AVIONICS FULL DUPLEX SWITCHED ETHERNET) END  
SYSTEM IMPLEMENTATION WITH STANDARD PC AND ETHERNET CARD**

submitted by EMRE ERDİNÇ in partial fulfillment of the requirements for the  
degree of Master of Science in Electrical and Electronics Engineering Department,  
Middle East Technical University by,

Prof. Dr. Canan Özgen  
Dean, Graduate School of Natural and Applied Sciences \_\_\_\_\_

Prof Dr. İsmet Erkmen  
Head of Department, Electrical and Electronics Engineering \_\_\_\_\_

Prof. Dr. Hasan Güran  
Supervisor, Electrical and Electronics Engineering Dept, METU \_\_\_\_\_

Examining Committee Members

Prof. Dr. Semih Bilgen  
Electrical and Electronics Engineering Dept, METU \_\_\_\_\_

Prof. Dr. Hasan Güran  
Electrical and Electronics Engineering Dept, METU \_\_\_\_\_

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı  
Electrical and Electronics Engineering Dept, METU \_\_\_\_\_

Asst. Prof. Dr. Şenar Ece Schmidt  
Electrical and Electronics Engineering Dept, METU \_\_\_\_\_

MSc. Mert KOLAYLI  
Avionics Design Engineer, TUSAS \_\_\_\_\_

Date: \_\_\_\_\_

**I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.**

Name, Last name : Emre ERDİNÇ

Signature :

## **ABSTRACT**

### **SOFT AFDX (AVIONICS FULL DUPLEX SWITCHED ETHERNET) END SYSTEM IMPLEMENTATION WITH STANDARD PC AND ETHERNET CARD**

Erdoğan, Emre

M.Sc. Department of Electrical and Electronics Engineering

Supervisor: Prof. Dr. Hasan Güran

May 2010, 107 pages

ARINC 664/AFDX (Avionics Full Duplex Switched Ethernet) protocol is a leading onboard communication technology in civil aviation. As AFDX is a new technology, unit cost of the hardware devices are high and protocol is open to changes. This thesis discusses the design of an AFDX End System application for test environment with a software based solution with cheap COTS (Commercial off-the shelf) equipment, explains the implementation of the software and analysis the performance.

Keywords: Avionics Full Duplex Switched Ethernet, Avionic data buses, Soft AFDX

## ÖZ

### STANDART PC VE ETHERNET KARTI İLE YAZILIMSAL AFDX (AVIONICS FULL DUPLEX SWITCHED ETHERNET) UÇ SİSTEM UYGULAMASI

Erdoğan, Emre

M.Sc. Department of Electrical and Electronics Engineering

Supervisor: Prof. Dr. Hasan Güran

Mayıs 2010, 107 sayfa

ARINC 664/AFDX (Avionics Full Duplex Switched Ethernet) protokolü, sivil havacılıkta önder bir uçak içi haberleşme teknolojisidir. Çok yeni bir teknoloji olduğundan donanım cihazlarının birim fiyatları yüksektir ve protokol değişikliklere açıktır. Bu tez çalışmasında, bir test ortamı için RAHAT (RAfta HAZır Ticari) ucuz ekipman kullanılarak yazılım tabanlı bir AFDX uç sistem tasarımı tartışılmış, uygulama yazılımı yapılmış ve performans analizleri gerçekleştirilmiştir.

Anahtar Kelimeler: Avionics Full Duplex Switched Ethernet, Avionic Veriyolları, Yazılımsal AFDX

## **ACKNOWLEDGMENTS**

The author wishes to express his gratitude to his supervisor Prof. Dr. Hasan Cengiz Gran for his guidance, advice, criticism, encouragements, insight and his tolerance throughout the research.

He would like to express his appreciation to the members of TAI, Turkish Aerospace Industries for their technical support and tolerance during this study.

The author would also like to thank his mother Neriman Erdi, his father Nedim Erdi and especially to his wife Selen Erdi for their endless support and patience. And lastly, he wishes to thank his friends Grsu Karateke, Cokun elik and Aykut Erden for their encouragement and support.

## TABLE OF CONTENTS

ABSTRACT .....	iv
ÖZ .....	v
ACKNOWLEDGMENTS .....	vi
TABLE OF CONTENTS .....	vii
LIST OF TABLES .....	x
LIST OF FIGURES .....	xi
LIST OF ABBREVIATIONS .....	xiii
LIST OF CHAPTERS	
1. INTRODUCTION.....	1
2. AVIONICS DATA BUSES .....	6
2.1. Types of Data Bus .....	7
2.1.1.Unidirectional Data Bus.....	7
2.1.2.Bidirectional Data Bus .....	8
2.2. Some of the Most Used Avionics Communication Protocols.....	9
2.2.1.ARINC 429 .....	10
2.2.2.MIL-STD-1553 .....	14
2.3. Avionics Full-Duplex Switched Ethernet (AFDX).....	20
2.3.1. History .....	20
2.3.2. Characteristics .....	20
2.3.3. AFDX Network Components.....	21
2.3.4. Comparison with other Avionics Buses .....	26
2.3.5. AFDX Solutions.....	27
3. FRAME STRUCTURE OF AFDX AND COMPARISON WITH THE RELATED COMMUNICATION STANDARDS.....	29

3.1. General .....	29
3.2. Physical Layer .....	30
3.3. Data Link Layer .....	30
3.3.1 Source Address .....	32
3.3.2 Destination Address .....	32
3.3.3 Type .....	33
3.3.4 Integrity Check .....	33
3.3.5 Redundancy Management .....	34
3.3.6 Flow Regulation .....	35
3.3.7 Flow Scheduling .....	35
3.3.8 Data Link Layer Overview .....	35
3.4 Network Layer .....	37
3.4.1 IP Structure .....	38
3.4.2 IP Source Address: .....	38
3.4.3 IP Destination Address .....	39
3.5 Transport Layer .....	39
3.6 Application Layer .....	41
4. IMPLEMENTATION .....	42
4.1. Platform .....	42
4.2. Tools .....	43
4.2.1 Development Environment .....	43
4.2.2 Data Link Layer Frame Access .....	43
4.3. Configuration .....	46
4.3.1. Installation of DLL .....	46
4.3.2. Compiler Settings .....	46
4.3.3. Linker Settings .....	47
4. 4 AFDX Transmitter .....	47
4.4.1 API Stack Overview .....	47
4.4.2 Application Flow Chart .....	50
4.4.3 Application Class List .....	51
4.4.4 Detailed Explanation of the AFDX Transmitter ApplicationN ....	61



4.5 AFDX Receiver.....	64
4.5.1 API Stack Overview.....	64
4.5.2 Application Flow Chart.....	67
4.5.3 Application Class List.....	68
4.4.4 Detailed Explanation of the AFDX Receiver ApplicationN.....	78
5. PERFORMANCE .....	81
5.1. Latency .....	81
5.2. MAC Constraints .....	84
5.3. Jitter.....	85
5.3.1 The Effect of Number of AFDX Ports on Jitter.....	86
5.3.2 The Effect of Number of Virtual Links on Jitter.....	89
5.3.3 The Effect of Lmax on Jitter .....	92
5.3.4 The Effect of BAG on Jitter .....	95
6. CONCLUSION .....	98
REFERENCES.....	102
AFDXAPI.H FILE OF AFDX TRANSMITTER .....	104
AFDXAPI.H FILE OF AFDX RECEIVER .....	106

## LIST OF TABLES

Table 2.1 Criteria for an Avionics Data Bus.....	7
Table 2.2 SSM Meaning for different data types.....	14
Table 2.3 Comparison Results .....	27
Table 5.1 Transmitter Latency Test Results Sample .....	83

## LIST OF FIGURES

Figure 1.1 Independent Avionics .....	2
Figure 2.1 Unidirectional data bus .....	8
Figure 2.2 Bidirectional data bus .....	9
Figure 2.3 ARINC 429 cabling .....	10
Figure 2.4. ARINC 429 Message Format .....	11
Figure 2.5. Binary data allocation .....	12
Figure 2.6. BCD data allocation .....	12
Figure 2.7 MIL-STD-1553B Bus concept .....	16
Figure 2.8 MIL-STD-1553B Coupling Types .....	16
Figure 2.9 MIL-STD-1553B Word formats .....	17
Figure 2.10 Message Sequence .....	19
Figure 2.11. Sample AFDX Network .....	21
Figure 2.12. AFDX End System .....	21
Figure 2.13. Physical Cable and Virtual Links .....	23
Figure 2.14. Bandwidth of the Virtual Links .....	24
Figure 2.15. Round Robin .....	25
Figure 2.16. AFDX Switch .....	25
Figure 3.1 A brief description in layered architecture perspective .....	30
Figure 3.2 Ethernet Frame (Data Link Layer) .....	31
Figure 3.3. AFDX Frame (Data Link Layer) .....	31
Figure 3.4 MAC Source Address .....	32
Figure 3.5 MAC Destination Address .....	33
Figure 3.6 Network Redundancy Concept .....	34
Figure 3.7 AFDX Transmitter Data Link Layer Overview .....	36
Figure 3.8 AFDX Receiver Data Link Layer Overview .....	37
Figure 3.9 AFDX Frame (Network Layer) .....	37
Figure 3.10 IPv4 Structure .....	38
Figure 3.11 IP Source Address .....	39
Figure 3.12 IP Destination Address .....	39
Figure 3.13 AFDX Frame (Transport Layer) .....	39
Figure 3.14 UDP Header .....	40
Figure 3.15 Allocation of SAP and AFDX Port Numbers .....	40
Figure 3.16 Port Allocation Range for IP Unicast or Multicast .....	40
Figure 3.17 AFDX Message Structure .....	41
Figure 4.1. Winpcap Capture Stack .....	45
Figure 4.2. AFDX Transmitter class hierarchy .....	49
Figure 4.3. AFDX Transmitter Application Flow Chart .....	50
Figure 4.4. AFDX Transmitter Class Diagram .....	52

Figure 4.5. AFDX Transmitter Sequence Diagram .....	61
Figure 4.6. AFDX Transmitter Configuration File .....	62
Figure 4.7. AFDX Receiver API Stack Overview .....	66
Figure 4.8. AFDX Receiver Application Flow Chart .....	67
Figure 4.9. AFDX Receiver Class Diagram.....	69
Figure 4.10 AFDX Receiver Sequence Diagram.....	78
Figure 5.1 Full Bandwidth Usage Test .....	85
Figure 5.2 The Effect of Number of AFDX Ports on Maximum Jitter.....	88
Figure 5.3 The Effect of Number of AFDX Ports on Average Jitter.....	89
Figure 5.4 The Effect of Number of Virtual Links on Maximum Jitter .....	91
Figure 5.5 The Effect of Number of Virtual Links on Average Jitter.....	92
Figure 5.6 The Effect of Lmax on Maximum Jitter .....	94
Figure 5.7 The Effect of Lmax on Average Jitter .....	94
Figure 5.8 The Effect of BAG on Maximum Jitter.....	97
Figure 5.9 The Effect of BAG on Average Jitter .....	97

## LIST OF ABBREVIATIONS

AEEC .....	Airlines Electronic Engineering Committee
AFDX.....	Avionics Full Duplex Switched Ethernet
API .....	Application Programming Interface
ARINC .....	Aeronautical Radio Incorporated
BC .....	Bus Controller
BCD .....	Binary Coded Decimal
BM .....	Bus Monitor
BNR .....	Two's complement binary
BNRZ.....	Bipolar Non Return to Zero
CAN .....	Controller Area Network
CSV .....	Comma-Separated Values
CPU .....	Central Processing Unit
DAL .....	Design Assurance Level
DAS.....	Data Acquisition System
DLL.....	Dynamic Link Library
EASA .....	European Aviation Safety Agency
FAA.....	Federal Aviation Administration
FADEC.....	Full Authority Digital Engine Control
FCS.....	Frame Check Sequence
ID .....	Identification
IEEE .....	Institute of Electrical and Electronics Engineers
IETF .....	Internet Engineering Task Force
IFG .....	Inter Frame Gap
IMA.....	Integrated Modular Avionics
IP .....	Internet Protocol

IP .....	Intellectual Property
ISO .....	International Organization for Standardization
Kbps .....	Kilo bits per seconds
LRU.....	Line Replaceable Unit
MAC.....	Medium Access Control Protocol
NIC.....	Network Interface Card
NPF .....	Netgroup Packet Filter
OS.....	Operating System
OSI .....	Open System Interconnection
PCI .....	Peripheral Component Interconnect
PMC .....	PCI Mezzanine Card
PHY.....	Physical Layer
RM .....	Redundancy Management
RT.....	Remote Terminal
RTCA .....	Radio Technical Commission for Aeronautics
RTOS .....	Real Time Operating System
SAE .....	Society of Automotive Engineers
SDI .....	Source/Destination Identifier
SFD .....	Start Frame Delimiter
SMTP .....	Simple Mail Transfer Protocol
SN.....	Sequence Number
SNMP.....	Simple Network Management Protocol
SSM.....	Sign/Status Matrix
TCP .....	Transmission Control Protocol
TFTP .....	Trivial File Transfer Protocol
TTL .....	Time To Live
UDP.....	User Datagram Protocol
VL .....	Virtual Link
VMEbus .....	Versa Module Eurocard Bus

# **CHAPTER 1**

## **INTRODUCTION**

The term “avionics” is the combination of the aviation and electronics, which could be defined as electronics of aircrafts, artificial satellites and spacecrafts. Collinson mentions that the word avionics “was first used in the USA in the early 1950s and has since gained wide scale usage and acceptance” [1].

The sensors and instrumentation structure in the aircrafts became more and more complicated in parallel with the development of the aircrafts day by day. In 1783, the Montgolfier brothers used a barometer to measure altitude. From about 1914, first gyro systems were used. “World War 2 drove a number of important advances including navigation aids, airborne radar and electronic warfare equipment” [2].

In the beginning of the second half of 20th century, aircraft avionics were composed of a few separate, analog systems such as radar, navigation and communication equipment and cockpit displays, connected by dedicated wiring. During 1960’s and 1970’s, the number of avionics increased per aircraft, digital technology appeared and systems began to be more complex. Also with the rise of digital technology, equipment was started to be designed to share information with each other. This communication needed increased the number of wirings in the aircraft which resulted in an increase of power consumption and weight. After

1970s, the use of data bus was introduced in aviation and this helped to reduce number of wirings and simplify total design and also maintenance.

Avionics architectures were also affected with these revolutions. The earliest architecture was independent avionics architecture in which each equipment had its own functionality independent of other similar or different equipment. Following figure represents a sample of independent avionic architecture.

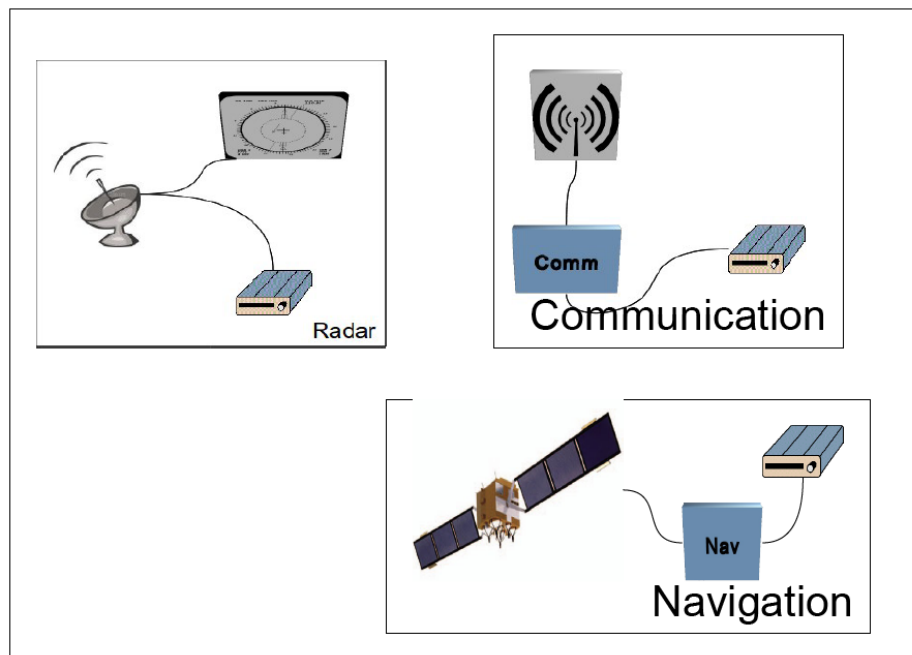


Figure 1.1 Independent Avionics

In the next decades, the integration of the different kind of systems emerged as a Federated Avionics Systems. Federated System Architectures have separate subsystems implementing functions using dedicated components, dedicated modules, LRUs, and software. Federated Architectures do not share or time-share component or information across subsystems in the avionics suite



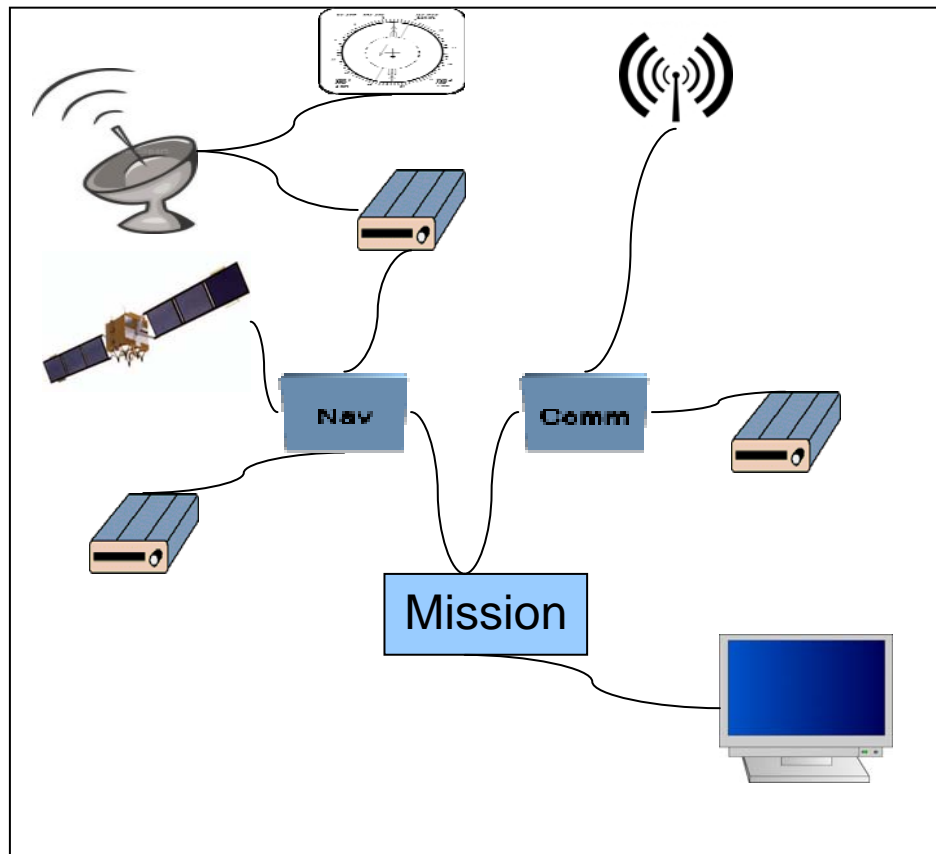


Figure 1.2 Federated Avionics

Integrated Architectures share “components” to support multiple functions. Integrated front ends simultaneously share or time-share the same antennas, RF/EO modules, processor modules and data buses. Integrated Software/Data/Control Architectures share information and control across subsystems or functions.

Last generation avionics architecture at the time of writing of this thesis was Integrated Modular Avionics (IMA). IMA combines LRU’s into software packages running on a computer. IMA defines the separation of the resources and enables certification independently [3]. “In the IMA architecture model, most special-to-purpose controllers are replaced by common standardized platforms (so-called IMA modules) that usually host applications of several systems. For inter-system communication, either IMA module-internal communication or standardized aircraft networking technology with guaranteed bandwidth and high availability is used” [4]. New architectures require new communication skills. Avionics Full Duplex

Switched Ethernet (AFDX) is a suitable data communication specification for IMA which supports means for addressing partitions.

AFDX is the leading edge avionics network technology that is chosen by the greatest aircraft companies like Airbus and Boeing. The first application of AFDX was started with A380 of Airbus and continued with A400M of Airbus Military and Boeing 787.

As the AFDX is a very new technology and is considered to be the future of the aviation, more companies started to focus on components and development of AFDX systems.

In this thesis we aim to study well known avionics data buses. Describe their basic specifications, characteristics and usage. Moreover we aim to study the features of the ARINC 664/AFDX specification in details and investigate layers of AFDX and messaging specifications, give a comparison with other avionics buses and underline its superiorities. Finally we aim to develop software running on a standard PC with a Windows XP operating system that uses standard Ethernet cards to implement AFDX communications and make performance analysis.

Chapter 2 gives a brief background on different avionic data buses. Describes the specifications of AFDX, makes a comparison between AFDX and other buses and lists advantages and disadvantages of both hard AFDX solutions and soft AFDX solutions.

Chapter 3 looks more closely into the AFDX frame structure and investigates the basic differences with the standard UDP/IP frames on Ethernet.

Chapter 4 explains the development environment, tools and the software developed for this thesis.

Chapter 5 gives the performance criteria for AFDX End System and describes the performance test conducted.

Finally Chapter 6 summarizes the thesis and concludes with comments on the test results and on the performance and states some future work directions.

## **CHAPTER 2**

### **AVIONICS DATA BUSES**

Wide usage of information sharing in avionics systems brought defining safety-critical communication protocols. Most modern avionics flight control systems are designed with a group of central computers named as mission computer, flight control computer, central control computer, etc. that are connected to sensors and actuators using point-to-point connections. To decrease the weight and increase flexibility, broadcast communication protocols is replacing the point-to-point architecture. As an example, the military transport aircraft C130J's avionic systems were previously using discrete wiring and were upgraded to the MIL-STD-1553B bidirectional data bus. This upgrade returned a significant weight saving, and a very high amount of work hour was saved to wire the harnesses. However, the bus handling became more complex.

The communication bus handling has grown up as a new discipline besides the engineering of the integrating complex systems. Some main basic rules (Table 2.1) have been defined to certificate an avionics data bus.

Table 2.1 Criteria for an Avionics Data Bus [5]

<b>Criterion</b>	<b>Selected Evaluation Factors</b>
Safety	Availability and reliability, Partitioning, Failure detection, Common cause/mode failures, Bus expansion strategy, Redundancy management
Data Integrity	Maximum error rate, Error recovery, Load analysis Bus capacity, Security
Performance	Operating speed, Bandwidth, Schedulability of messages, Bus length and max. load, Retry capability, Data latency
Electromagnetic Compability	Switching speed, Wiring, Pulse rise and fall times,
Design Assurance	Compliance with standards (such as DO-254 & DO-178B)
Configuration Management	Change control, compliance with standards, documentation, interface control, etc.
Continued Airworthiness	Physical degradation, in-service modifications and repairs, impact analysis, etc.

## **2.1. Types of Data Bus**

### **2.1.1. Unidirectional Data Bus**

In a unidirectional data bus, the system is driven by a single transmitter and a number of receivers that monitor the line and listen for all or specific data. If it is necessary to communicate back to the transmitter node, a separate transmission line should be defined. The simplicity of this topology makes the bus more reliable and easy to design and implement. But with respect to bidirectional buses, unidirectional buses need more wiring; resulting in more weight, more cost. The following figure represents a unidirectional data bus, LRU-1 is the driver (transmitter) and other LRUs are receivers.

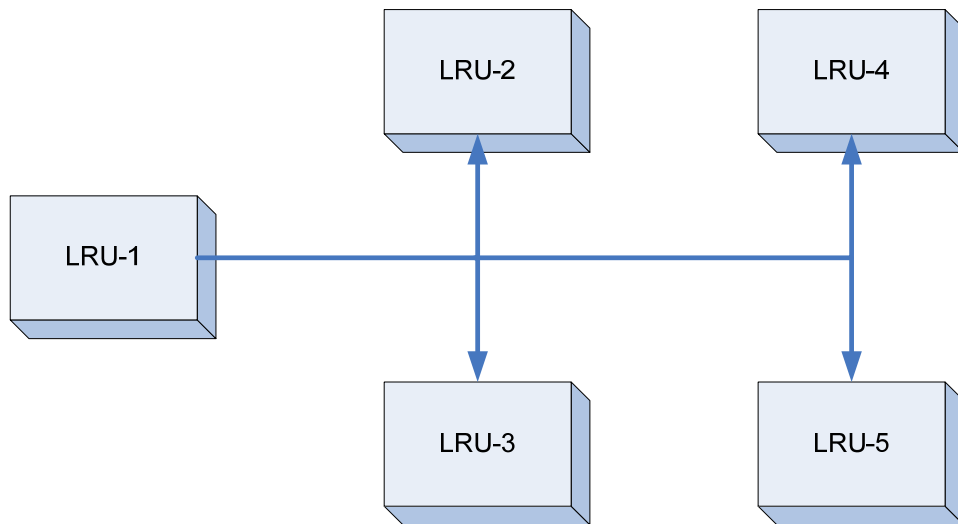


Figure 2.1 Unidirectional data bus

### 2.1.2. Bidirectional Data Bus

In a bidirectional data bus, each node can both receive and transmit. Arbitration of the bus may be master/slave or multi-master type.

In a master/slave mode type bus, there should always be a master that is responsible for all messaging on the bus. The master can initiate a data transfer from master to slave, slave to master or a slave to another slave. It is possible to design the bus with one or more backup masters that can take the control of the bus with a predefined scenario and control the messaging on the bus.

In a multi-master type bus, there is more than one node which can start a transmission. For this kind of buses arbitration mechanism should be defined. This mechanism may be realized electrically as in CAN bus, or in a time sharing basis.

Also the messaging can be half-duplex or full-duplex.

Following figure represents a bidirectional data bus. Each LRU is able to transmit and receive data on the same media.

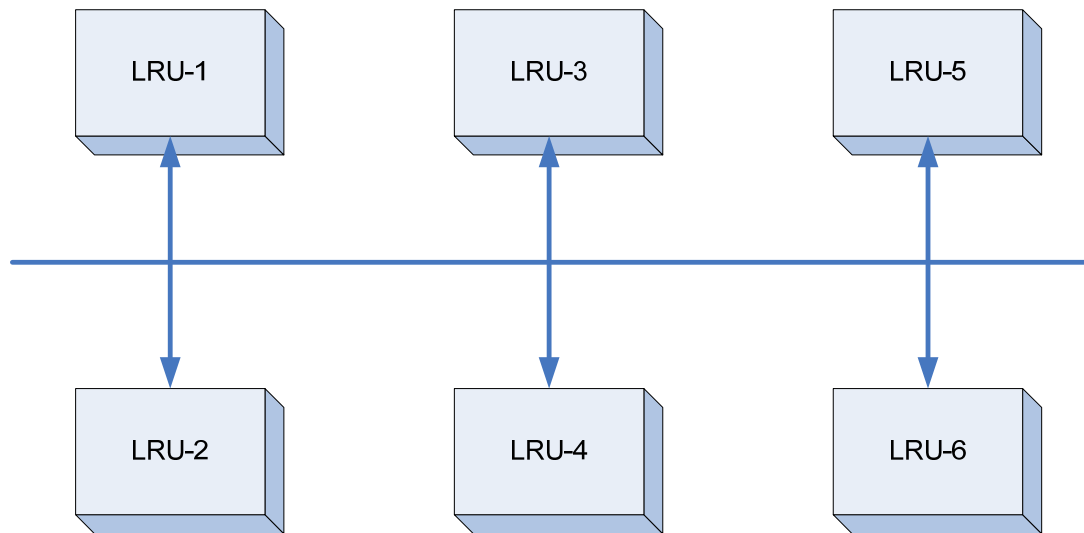


Figure 2.2 Bidirectional data bus

## 2.2. Some of the Most Used Avionics Communication Protocols

As the number of functions added to an air vehicle increase, to decrease the pilot workload, many complex functions that use lots of data are ported to the computers. As a result, data transfer for an aircraft increases, lots of intelligent LRUs appear and the need to communicate all of the LRUs becomes more and more important and complex.

The definition of the protocols makes it easier to integrate the systems and experience sharing for an aircraft development which gets more complicated in each new product. These definitions are published by some leading aviation committees like ARINC that cooperates with RTCA/EUROCAE, IEEE, Society of Automotive Engineers (SAE), EIA/ANSI/ISO Standards Organizations, etc.

These organizations develop and publish specifications for data communication for general and special purposes. Some of these protocols are widely used and some others are used for special applications. Following sections introduce into most common used protocols in aviation world.

### 2.2.1. ARINC 429

ARINC 429 is one of the most used data transfer protocol in aviation and according to ARINC 429 Tutorial of Condor Engineering [6], “is the most commonly used data bus for commercial and transport aircraft.”

ARINC 429 was developed from the old ARINC 419 Specification which was released in 1966 and had a final revision in 1983. ARINC 419 was defining four different wiring topologies which were also including ARINC 429 wiring type. In 1978, Aeronautical Radio Incorporated published the first release of ARINC 429 specification which was adopted by the AEEC in 1995. Specification is comprised of 3 parts; “ARINC Specification 429, Part 1-15: Functional Description, Electrical Interface, Label Assignments and Word Formats” which addresses the buses physical parameters, label and address assignments, and word formats, “ARINC Specification 429, Part 2-15: Discrete Word Data Standards” which defines the formats of words with discrete word bit assignments and “ARINC Specification 429, Part 3-15: File Data Transfer Techniques” which defines link layer file data transfer protocol for data block and file transfers [7].

The first issue about ARINC 429 specification is the physical layer characteristics. 78 ohm twisted shielded cable is used to connect the nodes. Shields should be grounded in both ends and in each junction as given in the figure.

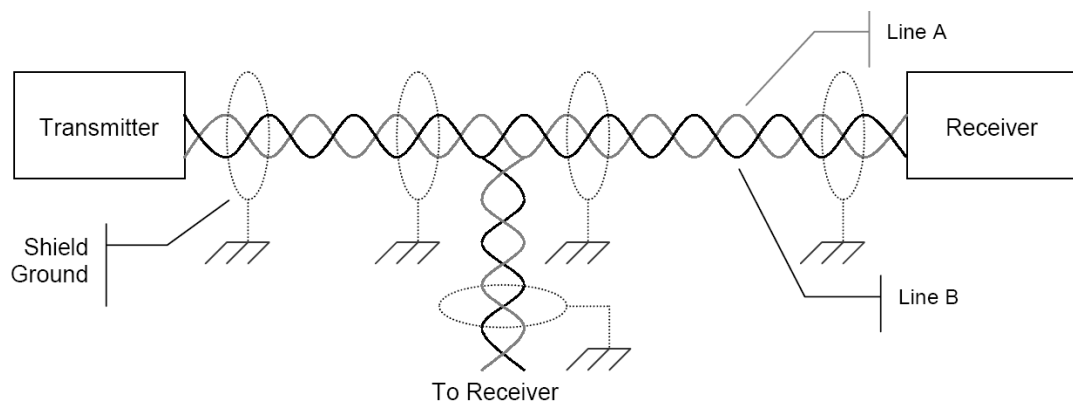


Figure 2.3 ARINC 429 cabling [7]



ARINC 429 is a differential line with bipolar non return to zero (BNRZ) signaling. Differential voltage between two lines is +10V (between 7,25 and 11V), -10V (between -7,25 and -11V) or 0 V (between -0,5 and 0,5V) for high, low and null respectively. 32 bit words are separated with 4 bit times of null gaps and each bit has a transition from the zero volts according to BNRZ which together help synchronization without a clock signal.

ARINC 429 is a simplex data bus, the transmitter starts unidirectional message flow to at least one but up to 20 receivers. Line length is not specified in the specification which depends on number and location of receivers.

Two bit rates are defined in the specification; high speed 100 kbps and low speed 12-14,5 kbps. Rise and fall times of the signal is  $10 \pm 5 \mu\text{s}$  for low speed and  $1,5 \pm 0,5 \mu\text{s}$  for high speed [8].

32 bit message consists of Label, SDI (Source/Destination Identifier), Data, SSM (Sign/Status Matrix) and Parity fields as given in the Figure 2.4. Label and parity are required fields in the message in contrast with optional SSM, SDI and data which can be used for different purposes.



Figure 2.4. ARINC 429 Message Format

8 bit Label identifies the data and message format which is generally expressed in 3 octal digits. Label is the first transmitted field that informs the receivers about the remaining of the message. Bit order of the message is least significant bit first but this order shows difference only for the label, of which the 8<sup>th</sup> bit is transmitted first.

Source/Destination Identifier is used to identify either the source of the data or the destination receiver. As it is optional it may also be added to the data in order to increase the resolution.

The 19 bits Data field starts from the 11<sup>th</sup> bit and continues up to 29<sup>th</sup> bit. This may be considered as a disadvantage that the resolution is not high enough for sensitive parameters. As ARINC 429 specification is very flexible, data may be used in different formats such as two's complement binary (BNR), decimal coded binary (BCD), discrete data, maintenance data and acknowledgement and ISO Alphabet #5 character data.

For binary format, generally 16 bits of the 19 bit is used as left aligned data and the remaining 3 bits (bits 11, 12 and 13) are padded with 0. But it is also possible to use all 19 bits for data. The most significant bit (29<sup>th</sup> bit) is the sign bit. A 1 in the sign bit indicates a negative number or a direction; "South", "West", "Left", "From" or "Below". A zero is used to indicate a positive number or "North", "East", "Right", "To" or "Over".

29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11
Sign	Value																Pad	

Figure 2.5. Binary data allocation

When BCD data format is used, the data field is divided into 5 subfields each representing a BCD digit. The most significant digit is 3 bits and it is possible to put numbers 0 to maximum 7. Other digits are 4-bit standard BCD digits.

29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11
BCD Digit 1			BCD Digit 2				BCD Digit 3			BCD Digit 4				BCD Digit 5				

Figure 2.6. BCD data allocation

In a Discrete data format ARINC 429 word, bits are named one by one to represent a specific Boolean data. Generally, bit assignment starts with the 11<sup>th</sup> bit

and unused bits are padded with 0. It is also common that discrete data are combined with binary or BCD data that is called as mixed data format. When the data format is mixed, discrete data starts from least significant bit of data and binary or BCD part is assigned from the most significant part.

Maintenance data or acknowledgement requires two way communications which can be established by using two ARINC 429 channels. Williamsburg/Buckhorn is a bit-oriented file transfer protocol where more than 21 bits are necessary to transfer also used in maintenance data or acknowledgement. This file transfer mechanism uses handshake, that's why, requires two ARINC 429 channels.

Another field in the ARINC 429 word is Sign/Status Matrix, bits 30 and 31, has different meanings according to data type. The sign or direction of the data, status of the transmitting equipment or validity of the data transmitted may be indicated with this two bit field. The meaning of the field according to data type is given in Table 2.2.

The 32<sup>nd</sup> bit of the word is parity bit. Odd parity is used in ARINC 429 messages.

Table 2.2 SSM Meaning for different data types

BIT		BNR Data Type	BCD Data Type	Discrete Data Type
31	30			
0	0	Failure Warning	Plus, North, East, Right, To, Above	Verified Data, Normal Operation
0	1	No Computed Data	No Computed Data	No Computed Data
1	0	Functional Test	Functional Test	Functional Test
1	1	Normal Operation	Minus, South, West, Left, From, Below	Failure Warning

### 2.2.2. MIL-STD-1553

US Department of Defense published MIL-STD-1553 specifications in 1973 as a standard of US Air Force that defines mechanical, electrical and functional properties of a time division command response multiplex bus. It was designed to be used in military aviation but it is still being widely used in civil and transport aircrafts and also for in other sectors like railway systems. The first application of MIL-STD-1553 was F-16 Fighting Falcon. Then, it was used in F-18 Hornet and many other programs.

MIL-STD-1553A of 1975 was revised as MIL-STD-1553B in 1978. One of the most important differences between two versions is; MIL-STD-1553B defines protocol more strictly and clearly than MIL-STD-1553A and therefore enables an easier integration of subsystems from different companies. Also redundancy requirement of MIL-STD-1553B is a major difference. MIL-STD-1553B has six revisions from 1978 to date.

Bi-phase Manchester coded data is transmitted via a cable pair that has a 70-80  $\Omega$  -typically 78  $\Omega$ - impedance at 1 MHz. The bit rate is 1 Mbps for the bus. Bus

can have double or triple redundant media as using various pairs of cables independently. Common usage is dual redundant bus. For dual redundant usages, one of the lines is active and the alternate bus always stays silent and is used only when a failure occurs in the active bus. General naming for redundant lines in a MIL-STD-1553B bus is; Primary bus (Bus A) and Secondary bus (Bus B).

The messages are transmitted over 16 bits words (command, data or status). Each word starts with a 3  $\mu$ s synchronization pulse, 1,5  $\mu$ s low and 1,5  $\mu$ s high, and ends with a parity bit. Parity for MIL-STD-1553B is odd. In practical, each word can be considered as 20 bits: 3 bits for synchronization, 16 bits of payload and 1 bit for parity check. The words in a message are transmitted consecutively and there is a 4  $\mu$ s delay between each word. A remote terminal is required to respond a command in maximum 12  $\mu$ s.

The elements of the MIL-STD-1553B bus are BC (bus controller), RT (remote terminal) and BM (bus monitor). For a MIL-STD-1553B network, BC and RT are essential.

Bus controller is responsible to control all communication in the bus, emission or reception is done with a command of the bus controller. There can only be one bus controller for a bus. Also backup bus controller (BBC) is defined as a backup for the bus which is able to take the control [9].

Remote terminals are 5 bit addressed slave devices responding to the bus controller. Remote terminals count has been limited with 31 devices for each line [10].

There is no limitation in number for bus monitors. System integrator is free to install unlimited number of bus monitors to the bus in order to record traffic or simply not to use a bus monitor. Bus monitors are devices that have no address and never respond to any message. Alignment of the devices in the bus is given in the following figure.

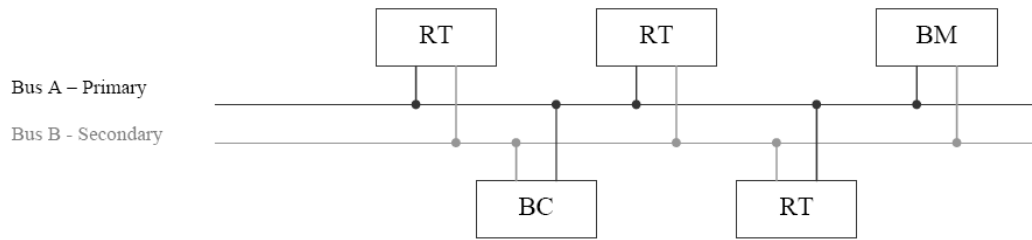


Figure 2.7 MIL-STD-1553B Bus concept

Connection of devices to the bus can be done in two types; direct coupling and transformer coupling. Direct coupling is connecting the conductors directly whereas transformer coupling is connecting a terminal to the bus over a transformer. Stub length for direct coupling is 1 foot and this length increases up to 6 feet for a transformer coupling. Following figure shows both coupling types.

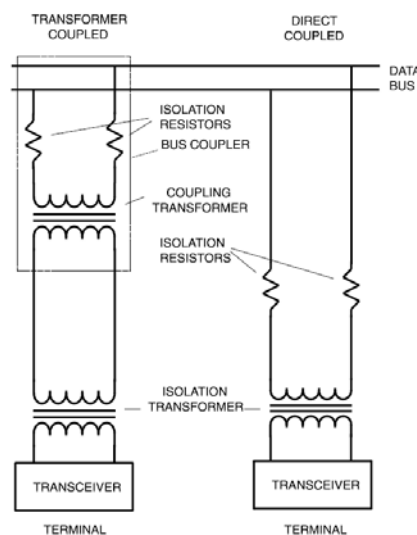


Figure 2.8 MIL-STD-1553B Coupling Types

There are three types of words defined for MIL-STD-1553B bus; command word, data word and status word.

Bus controller always initiates the messaging by a command word. The command word is formed as follows; The first 5 bits are the address of the remote terminal (0-31). The sixth bit is “0” for reception and “1” for transmission. The

direction of the flow is always decided with the remote terminal view. The 5 following bits indicate the position at which to supply or acquire the data (subaddress) in the terminal (1-30) or the type of the message is a mode code (0, 31). The final 5 bits indicate the number of the words to wait for (1-32). If all are “0”, indicate 32 words to transmit or receive. In the case of the mode code, these bits are the number of the mode code; such as Initiate Self Test, Transmit BIT Word, etc.

The status word is formed as; the first 5 bits are the remote terminal address that is replying. 9<sup>th</sup> to 19<sup>th</sup> bits are used as flags and as reserved bits. Following figure shows each word type with bit field indications.

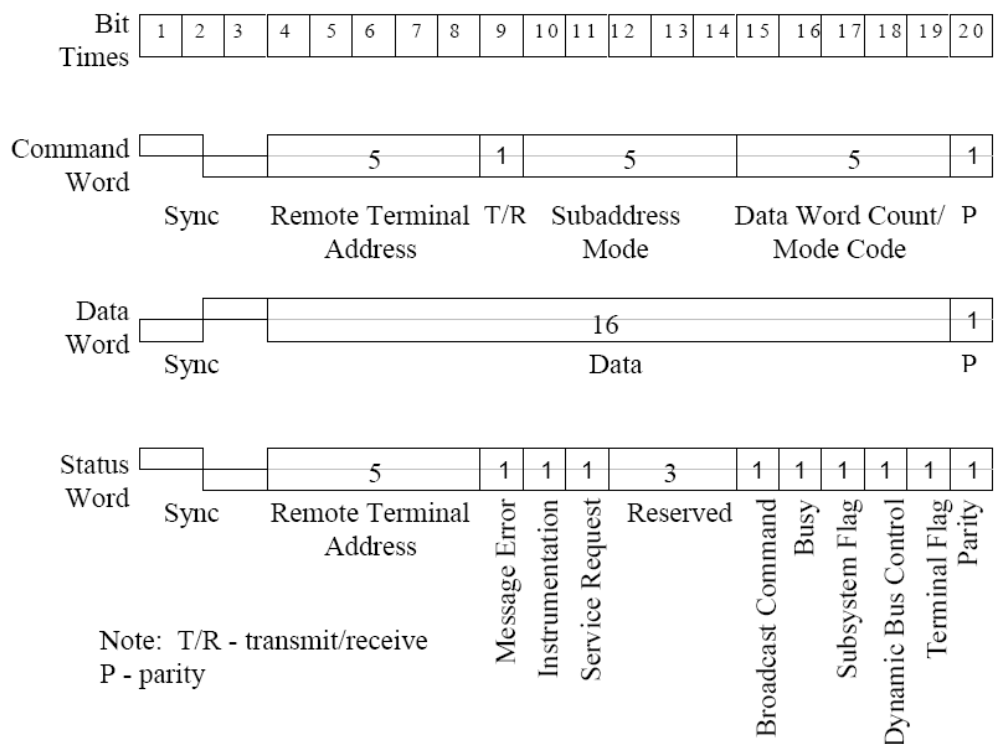


Figure 2.9 MIL-STD-1553B Word formats

There are 9 types of transaction between bus controller and remote terminals.

- BC to RT: The bus controller sends one command word that includes the remote terminal address, subaddress and number of data words to

be received by the RT which is immediately followed by 1 to 32 data words. The remote terminal which is selected sends only one status word to the bus controller that informs about status of the transaction and terminal.

- RT to BC: The bus controller sends one word to the remote terminal that includes the remote terminal address, subaddress and number of data words to be transmitted by the RT. The remote terminal sends only one status word that informs about status of the transaction and terminal which is immediately followed by 1 to 32 data words to the bus controller.
- Mode Code with / without data word: The bus controller sends one command word with a subaddress/ mode field filled with 0 or 32. The command can be followed by a word dependent to the selected mode code. The remote terminal responds with a status word that can be followed by a single word of data according to mode code type.
- RT to RT: The bus controller sends one command word for the receiving remote terminal to receive data followed by one command word to the transmitting remote terminal to transmit data. The transmitting terminal sends one word followed by 1 to 32 words of data to the receiving terminal. The receiving terminal sends its status word finally.
- BC to RTs (Broadcast Data): This functionality did not exist at MIL-STD-1553A. The bus controller sends one command word to terminal 31, in reality this indicates that the command word is sent as broadcast type, followed by 1 to 32 data words. All the remote terminals accept the data without replying with a status word. This functionality can be used for the actualization of the whole system such as time information.
- Broadcast Mode Code with / without data word: The bus controller sends one command word with a subaddress/ mode field filled with 0 or 32. The command can be followed by a word dependent to the selected mode code.



The sequence for each transaction is defined in the standard. The sequences guarantee that the terminal is working properly and ready to transmit or receive data. The request issued at the end of the transmission shows that the data is received and the result of the transmission of the data is legitimate. This sequence proves the high-integrity of the MIL-STD-1553B.

Remote terminal device can not originate the transmission by itself. The requests of the transmissions are generated by the bus controller to the terminals. The high-priority messages appear more frequently compared to the low-priority ones but the protocol does not specify any time-share between word types. This is decided by the system architectures by taking into account that the absence of the reply indicates failure.

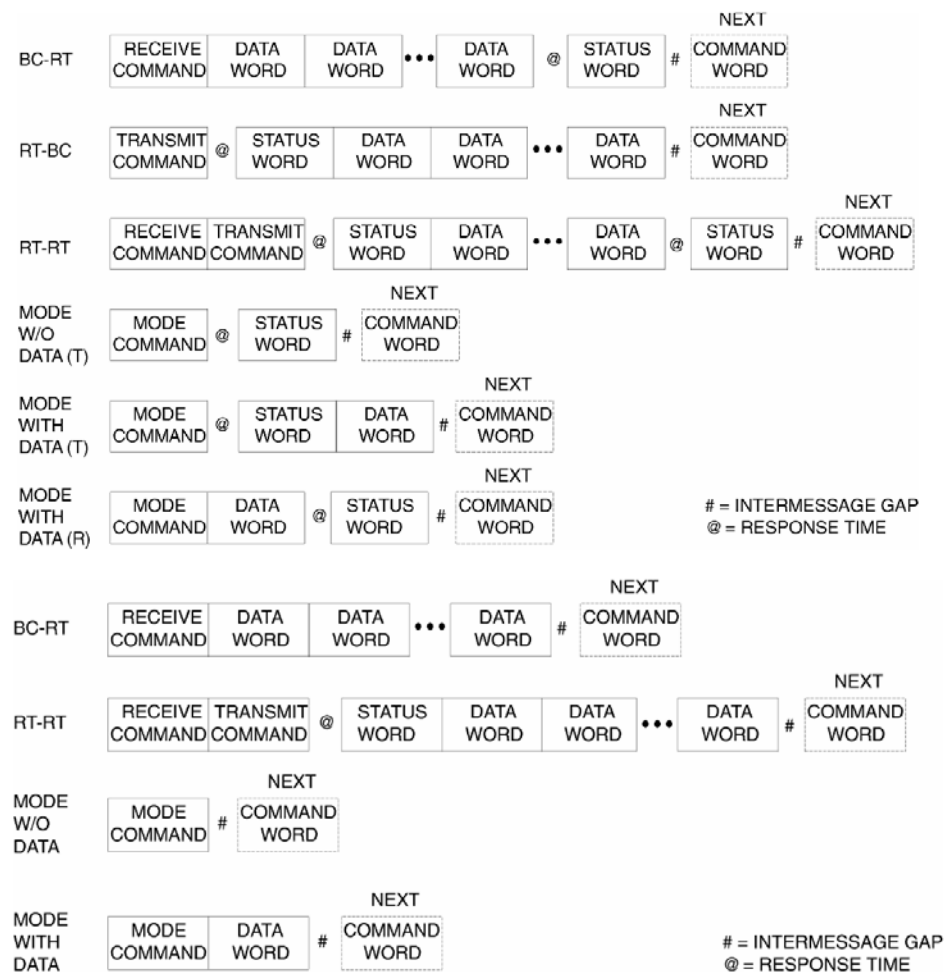


Figure 2.10 Message Sequence

## **2.3. Avionics Full-Duplex Switched Ethernet (AFDX)**

### **2.3.1. History**

As described in the other network types, limitations due to the topology or protocol forced the system integrators to develop a new definition by taking the advantage of the flexibility and wide usage in terrestrial networks and readily available software of IEEE 802.3, Ethernet.

“ARINC 664 Aircraft Data Network specification defines electrical characteristics and protocol recommended for commercial avionics. It is Ethernet for avionics, which is usually shunned by avionics engineers for its non-determinism. This was overcome when Airbus created Avionics Full Duplex Ethernet (AFDX™) and the Airlines Electronic Engineering Committee (AEEC) adapted it in ARINC 664 Part 7, published on June 27, 2005. AFDX is a deterministic protocol for real time application on Ethernet media.” [11]

### **2.3.2. Characteristics**

AFDX is a serial data bus which supports data transmission at 10 or 100 Mbps rates over a copper or fiber transmission medium. It is a deterministic network, which guarantees the bandwidth of each logical communication channel, called a Virtual Link (VL) with traffic flow control. The jitter and transmit latency are defined and limited. Packets are received in the same order that they are transmitted. Carrying the same information at the same time over two redundant channels ensures the reliability and availability of the AFDX standard; each AFDX channel has to be a dual redundant channel.

These characteristics make AFDX to ensure “a BER as low as  $10^{-12}$  while providing a bandwidth up to 100 Mbps.” [12]

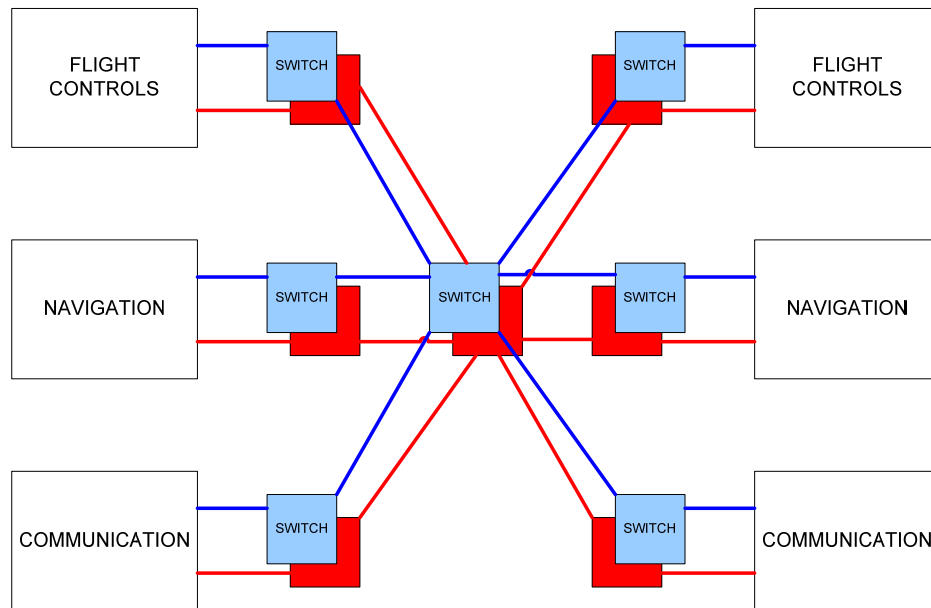


Figure 2.11. Sample AFDX Network

### 2.3.3. AFDX Network Components

#### 2.3.3.1. End System

The AFDX End System (ES) is the part of an avionics system or an avionics subsystem that connects logical unit to the physical AFDX network. An End System implements protocol specific functions and carries out messaging features. Functions may be classified as transmitting function and receiving functions.

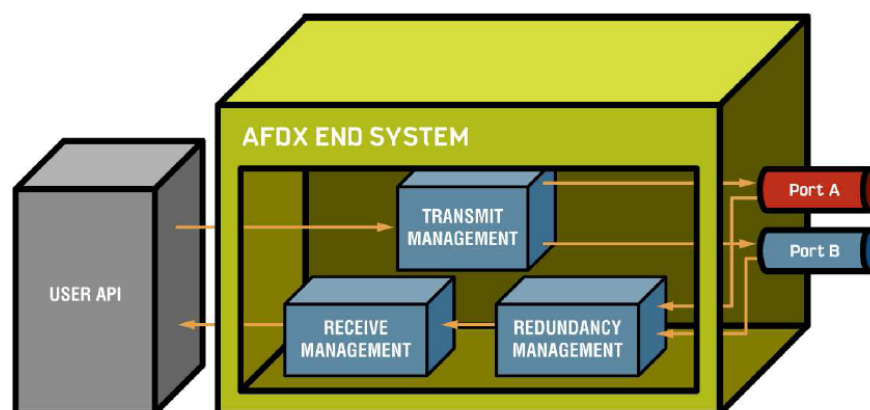


Figure 2.12. AFDX End System [13]

“Transmit Management allows creation of VL, transmitting of data, and scheduling of messages onto the network interconnect media. Redundancy Management allows gathering of the correctly ordered data using both port A and port B in case of data corruption. Receive Management allows correctly ordered data to reach the API” [13]. The requirements for an AFDX End System which will be described later are addressed in ARINC 664 specification.

#### **2.3.3.2. Virtual Link**

The Virtual Link (VL) concept has been inspired from ARINC 429 protocol. Virtual Links are logical implementations of the unidirectional point-to-point physical connections of ARINC 429. A transmitting Virtual Link may be connected to unlimited number of receiving Virtual Links but a receiving Virtual Link shall be connected to only and only one transmitting Virtual Link.

Each interface of an AFDX End System is connected to the switch which connects the End System to the AFDX network. This connection is established via a single physical transmission line. However, definition of the Virtual Links enables establishing many separate and “isolated” logical connections between End Systems on the same physical medium.

“VLs make it possible to establish a sophisticated network communication while ensuring a deterministic behavior through VL bandwidth policing carried out by the switch.” [12] An AFDX End-System is required to support up to 128 VLs.

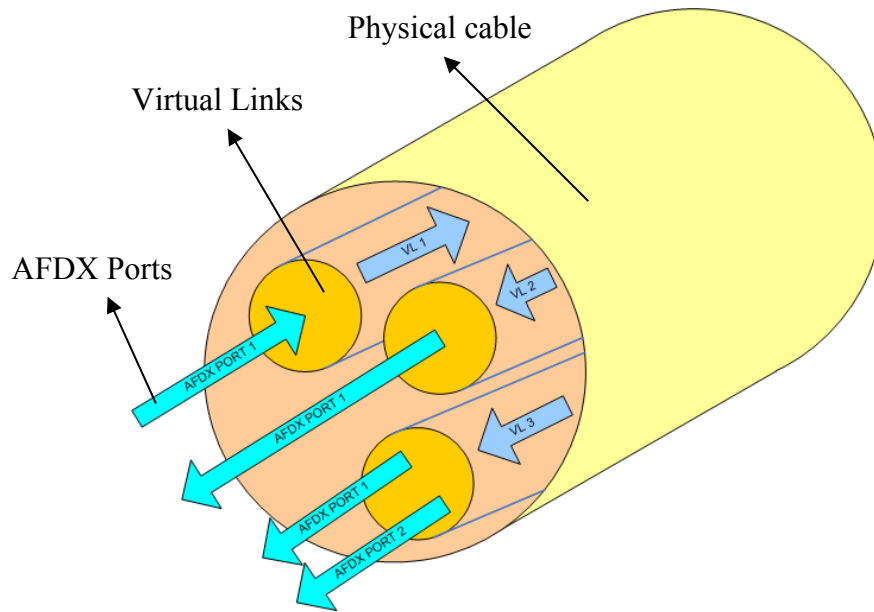


Figure 2.13. Physical Cable and Virtual Links

### Virtual Link Scheduling

An End System has to shape the generated traffic according configuration of the network and AFDX messaging properties. In the AFDX specifications, the bandwidth of the Virtual Links is fixed by defining a time window Band Allocation Gap (BAG) and a jitter that tolerates the delays which occur on the medium (Figure 2.14). In each VL there are multiple frames that have to be ordered into a single flow of frames. The BAG defines the minimum time slot between two consecutive frames which has a value in the range of 1 to 128 ms with the powers of 2; 1ms, 2 ms, 4 ms, 8 ms, 16 ms, 32 ms, 64 ms and 128 ms. “The variation (i.e., standard deviation) in the packet arrival times is called jitter.” [14] In AFDX messaging, it is expected to observe a frame of the same VL in the interval of BAG time after the last frame and BAG time plus maximum jitter time after the last frame.

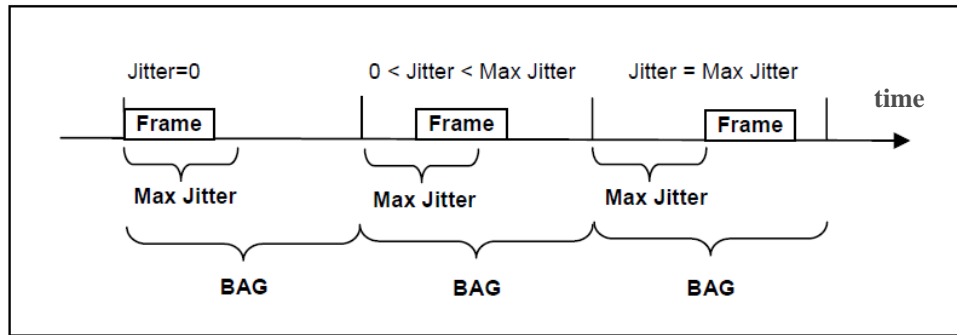


Figure 2.14. Bandwidth of the Virtual Links

### Sub Virtual Links

“To accommodate less critical data communication needs, AFDX also allows for the construction of sub virtual links (sub-VLs).” [15] Sub Virtual Links are data queues that share a single Virtual Link. A Virtual Link reads each data queue in a round robin manner and services the available data to the network according to its BAG definition. Sub Virtual Link and normal port communication cannot be assigned to a Virtual Link at the same time; if sub Virtual Links are assigned to a Virtual Link, it can only serve to these sub Virtual Links. On the other hand any sub Virtual Link cannot distribute its messages across different Virtual Links. Sub-VLs are useful for non critical or non time critical data transmission.

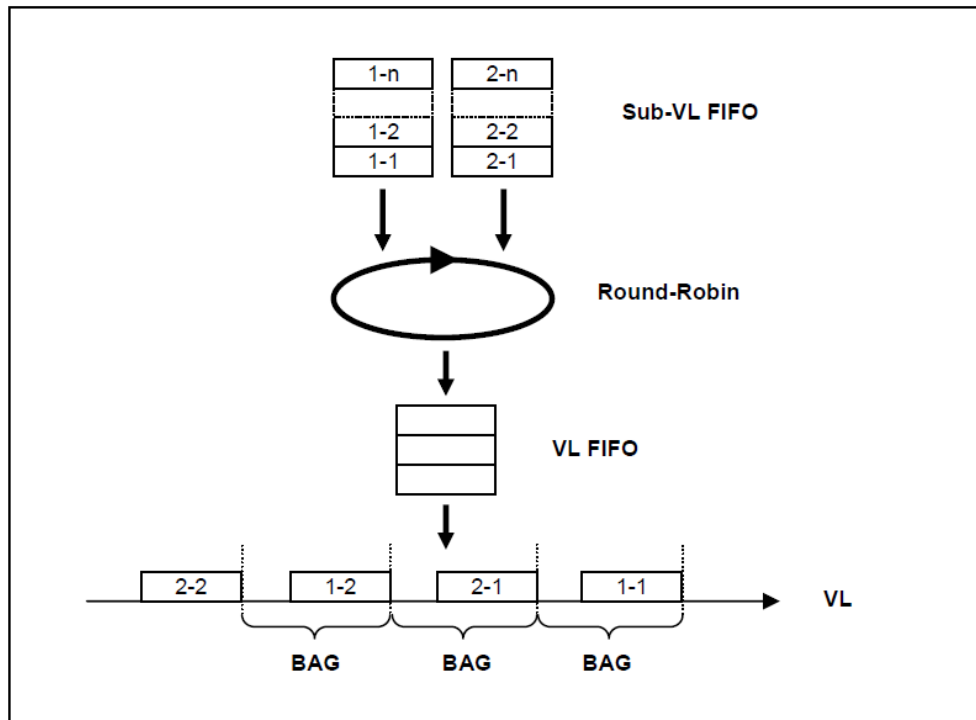


Figure 2.15. Round Robin

### 2.3.3.3. AFDX Switch

AFDX switch is the central element of the star topology of AFDX network that interconnects the End System to the End System and network to the network.

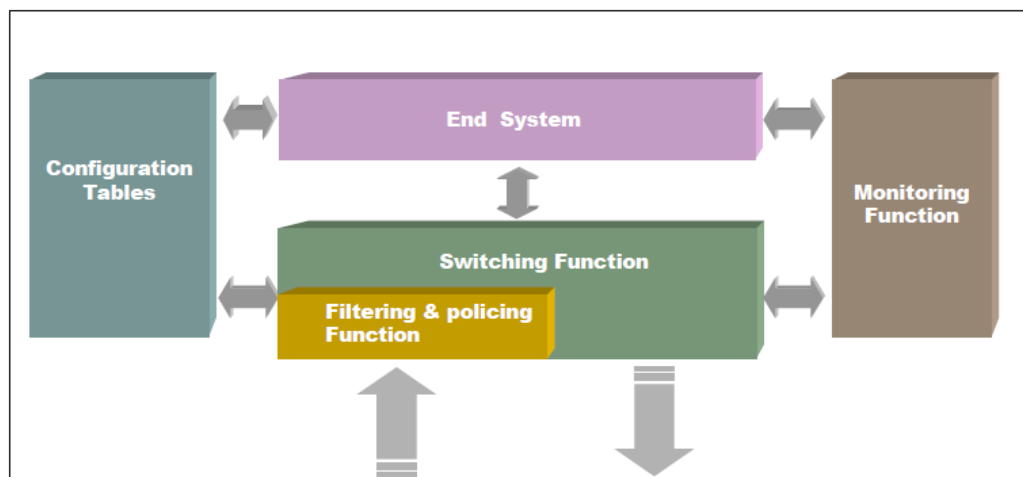


Figure 2.16. AFDX Switch [16]

The functions assigned to an AFDX Switch are; filtering, policing and switching valid incoming frames to the correct destinations according to network configuration. All incoming frames enter to the filtering and policing functional block that drops the invalid frames and switching functional block deliver the valid frames to the correct destinations. Information about the frames is stored in the configuration tables. Switch is configured with “static” configuration tables according to the network architect’s definitions.

Monitoring Function is used to monitor and log all switch operations and service health and network status data to the subsystems.

AFDX Switch plays the most important role for the shaped traffic feature of the AFDX network.

#### **2.3.4. Comparison with other Avionics Buses**

Schuster and Verma [11] made a comparison between AFDX and two major avionics data buses; ARINC 429 which is commonly used in civil aircrafts and MIL-STD-1553B which is commonly used in military aircrafts. They started to evaluation with six stakeholders; Performance, Reliability, Security & Certifiability, Cost, Evolvability & Flexibility and Supportability & Logistics. Then they listed 13 criteria; Transmission Speed, Throughput, Latency, Quality of Service (QoS), Partitioning, Redundancy, Topology, Harness Requirements, Software (SW) Development, Weight, Hardware (HW) Reliability, Software Reliability and COTS Availability. They weighted each criterion according to stakeholders with subjective approach according to their experience and graded them with four numbers; 0 for not linked, 1 for possibly linked, 3 for moderately linked and 9 for strongly linked.

According to their evaluation result “AFDX bus provides more redundancy, security, speed, determinism, and long-term cost effectiveness for the cumulative



required support (wire, hardware, and software).” Following table shows the results of their evaluation.

Table 2.3 Comparison Results [11]

CRITERIA	BUS SCORE			PRIORITIES
	ARINC 429	MIL-STD-1553B	AFDX	
Transmission Speed	1	3	9	5%
Throughput	1	3	9	4%
Latency	9	1	3	9%
QoS	9	1	3	4%
Partitioning	3	1	9	12%
Redundancy	9	3	9	9%
Topology	3	3	9	12%
Harness Requirements	1	3	9	6%
SW Development	3	3	9	6%
Weight	3	3	3	2%
HW Reliability	3	3	3	12%
SW Reliability	3	3	3	11%
COTS Availability	3	3	9	7%
<b>SCORE</b>	<b>4</b>	<b>2,5</b>	<b>6,7</b>	

### 2.3.5. AFDX Solutions

#### 2.3.5.1. Hard AFDX Stack

Most common implementation type of AFDX is with specific hardware in PMC, PCI, PCI-X, VMEbus or other form factors. The electronics circuitry may use standard Ethernet chips with a microcontroller or a DSP or implement with an FPGA with use of specific IP (Intellectual Property) core. Some FPGA companies like Actel publishes design guidelines for AFDX End System implementation. [15]

Pickles [13] lists the advantages of a hard AFDX solution as reduced host processing requirement and easy distribution as a COTS PMC style board.

He also lists the disadvantages as possible hardware design change requirements due to being new technology of AFDX, component obsolescence issues that may require redesign and re-certification of the product and costs of creating and maintaining firmware code, which is typically written in low-level assembly language.

#### **2.3.5.2. Soft AFDX Stack**

Soft AFDX stack is another idea to implement End System in the host computer along with application software. This implementation option uses processing resources of computer to realize protocol requirements and readily available Ethernet hardware for interconnection to the physical medium.

For Soft AFDX, Pickles [13] starts to list the advantages with cheap and readily available hardware. He implies the easily updating opportunity to faster transmission rates without changing the design. He adds that there is no obsolescence issue and possible protocol changes may be implemented in a high-level language, such as C, C++ or Ada.

He also highlights the disadvantages of Soft AFDX stack as the need for more processing power for protocol implementation on the host and necessity to Ethernet driver and protocol stack optimization for hard performance requirements of the standard such as latency.

## **CHAPTER 3**

### **FRAME STRUCTURE OF AFDX AND COMPARISON WITH THE RELATED COMMUNICATION STANDARDS**

#### **3.1. General**

OSI (Open System Interconnection) Reference Model is composed of seven layers as given below;

- Physical Layer
- Data Link Layer
- Network Layer
- Transport Layer
- Session Layer
- Presentation Layer
- Application Layer

On the other hand, TCP/IP Model is composed of 4 layers given below;

- Link Layer
- Internet Layer
- Transport Layer
- Application Layer

AFDX is also an open standard, inspired with these layered models and common protocols. Following figure gives a brief description in layered architecture perspective.

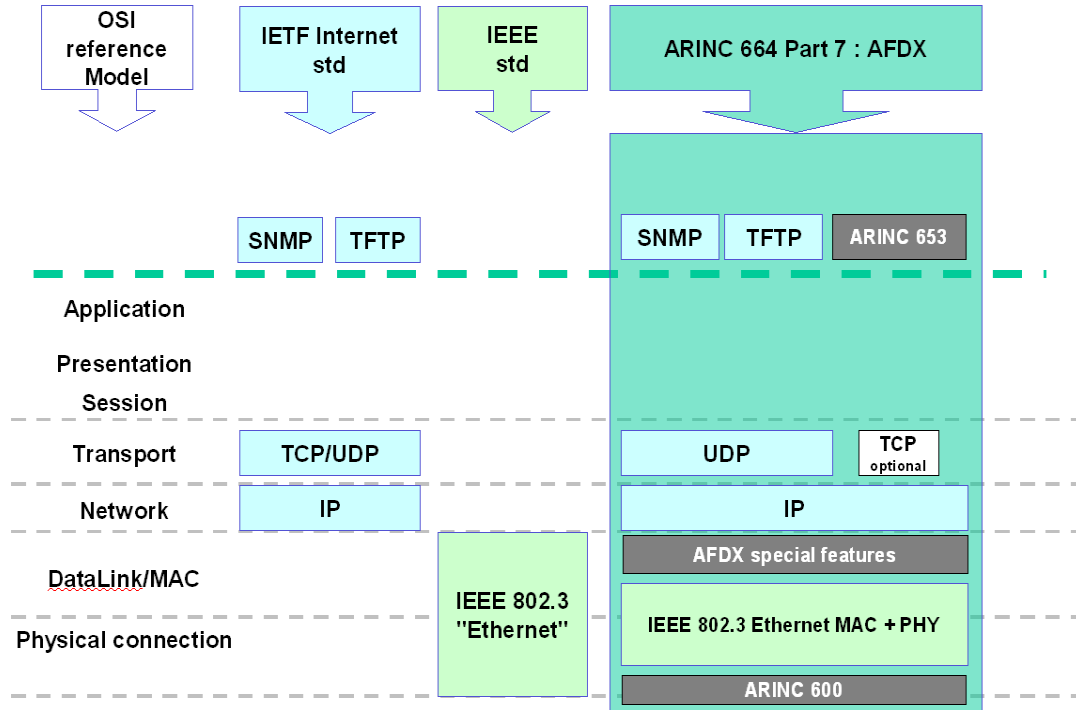


Figure 3.1 A brief description in layered architecture perspective

### 3.2. Physical Layer

Physical layer is not restrictedly specified for AFDX, but should be any of the ARINC 664 Part 2 defined solutions. The physical layer that is used in modern day computers is acceptable. Connection with CAT-5, CAT-6 or fiber media may be used with a star topology. The physical layer part of Ethernet 802.3 is used for AFDX systems.

### 3.3. Data Link Layer

In Draft 3 of Project Paper 664: Aircraft Data Network, Part 7 - Avionics Full Duplex Switched Ethernet (AFDX) Network [16], it is denoted that IEEE Standard 802.3, 2000 Edition, is considered an integral part of ARINC 664 specification.

A standard Ethernet frame is given below;

7 (bytes)	1	6	6	2	45-1499	4	12
Preamble	S F D	Destination Address	Source Address	T y p e	Ethernet Payload	F C S	IFG

Figure 3.2 Ethernet Frame (Data Link Layer)

Ethernet frame has a header;

- Destination MAC Address (6 bytes)
- Source MAC Address (6 bytes)
- Ethernet Type (2 bytes) And a trailer;
- Frame Check Sequence (4 bytes)

Payload part is minimum 46 bytes and maximum 1500 bytes.

For AFDX this structure is same with a difference, just after the end of the payload part and before the FCS, a Sequence Number is inserted which will be described later.

7 (bytes)	1	6	6	2	45-1499	1	4	12
Preamble	S F D	Destination Address	Source Address	T y p e	AFDX Payload	SN	F C S	IFG

Figure 3.3. AFDX Frame (Data Link Layer)

### 3.3.1 Source Address

The specification specifies the MAC Source address as an Individual and Locally Administered address compliant with IEEE 802.3 protocol.

Ethernet MAC controller Identification 48-bits						
Constant field: 24-bits	Network_ID 8 bits		Equipment_ID 8-bits		Interface_ID 3-bits	Constant field: 5-bits
	Constant field 4-bits	4-bits	3-bits	5-bits		
"0000 0010 0000 0000 0000 0000"	"0000"					"00000"

Figure 3.4 MAC Source Address

Network ID and Equipment ID are assigned by the network and equipment designers.

Interface ID is 3 bits long but has two options;

‘001’ The Ethernet MAC controller is connected to the network A

‘010’ The Ethernet MAC controller is connected to the network B

Network identification is used for redundancy and other bit combinations are not used.

The main difference from standard Ethernet is the ability of the end-user to configure the source MAC address according to his network by either jumper set or soft pre-configuration.

### 3.3.2 Destination Address

The specification specifies the MAC Destination address as a Group and Locally Administered address compliant with IEEE 802.3, which carries the Virtual Link information in the last 16 bits.

48-bits	
Constant field 32-bits	Virtual Link Identifier 16-bits
xxxx xx11 xxxx xxxx xxxx xxxx xxxx	

Figure 3.5 MAC Destination Address

Each End System should get "constant field" and "Virtual Link Identifier" values from the system integrator. The values are not specified in ARINC 664. The constant field should be the same for each End System in any given AFDX network. The least significant bit of the first byte indicates the group address (always = 1).

In order to use the standard Ethernet frame, MAC group addresses should be changed to send frames from End System to End System (s).

The second to least significant bit of the first byte indicates the locally administered address (always = 1).

### 3.3.3 Type

2 byte type part of the frame is always 0x800, indicating IPv4.

### 3.3.4 Integrity Check

In addition to the frame check sequence bytes of the standard Ethernet frame, frame sequence number (SN) is 1 byte long and should be located just before the MAC CRC field as illustrated in the Figure 3.3. Sequence number is used for integrity check.

The transmitter is in charge of putting a sequence number per VL basis in a range of 0 to 255. Transmitter initializes the sequence number of each Virtual Link as 0 after each End System reset and increments by one for each consecutive frame

of the same Virtual Link. When the value reaches 255, transmitter wraps-around the sequence number to 1 instead of 0.

When the frame is valid according to sequence number, integrity check function passes the frame to redundancy management. When the sequence number is faulty, it drops the invalid frame and informs network management. Frames with a sequence number 1 or 2 more than the sequence number of last received frame are accepted as valid frames. Increment with one or 2 must be conducted with the care of wrapping-around to 1 after 255.

“This function increases integrity robustness by, for example, eliminating stuck frames or single abnormal frames and reducing the impact of a babbling switch. Loss of one single frame is considered as a normal event due to a non-zero Bit Error Rate.” [16]

### 3.3.5 Redundancy Management

As another difference from standard Ethernet, AFDX has a defined redundancy management. The interconnections of End Systems are established through two different redundant networks. This protects the loss of communication from single failure of a cable or switch.

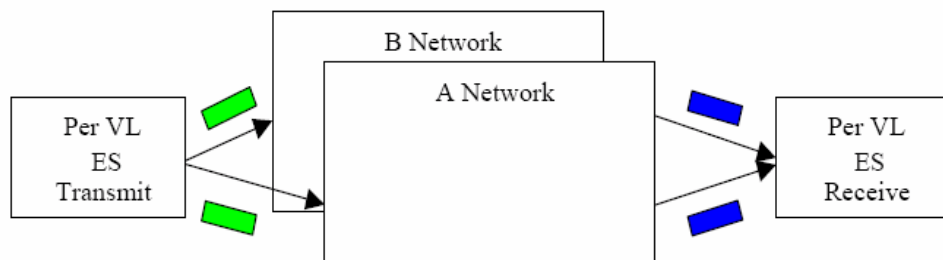


Figure 3.6 Network Redundancy Concept

According to the specification [16] on a per VL basis, the End System should be able to receive:



- a redundant VL and deliver to the application one of the redundant data (RM active)
- a redundant VL and deliver to the application both redundant data (RM not active)
- a non redundant VL on either attachment and submit data from it to the application (in this case, RM can be active or not).

### **3.3.6 Flow Regulation**

ARINC 664 defines fixed BAG for each Virtual Link. If application generates frames exceeding this BAG, in the data link layer, End System makes a regulation as one frame per gap basis. Flow regulation is only applicable to transmitter side End System.

### **3.3.7 Flow Scheduling**

As several Virtual Links defined in an End System with a single physical output (except the redundant) there should be a scheduling control for each Virtual Link to direct the physical layer.

In a transmitting end system with multiple VLs, the Scheduler multiplexes the different flows coming from the Regulators.

The End System should regulate transmitted data on a per VL basis, since this Traffic Shaping Function (exact knowledge of flow characteristics) is the basis of the determinism analysis. On a per VL basis the traffic regulator or traffic shaping function should shape the flow to send no more than one packet in each interval of BAG milliseconds.

### **3.3.8 Data Link Layer Overview**

Figure 3.7 represents the data link layer operations in the transmitter point of view of AFDX End System.

- Regulator regulates frames coming from upper layer
- Regulated frames in Virtual Link are put inside the BAG.
- Schedule Multiplexer multiplexes the frames coming from several Virtual Links and conducts to Redundancy Management.
- Redundancy Management unit produces two instances of the same frame if not disabled.
- Each MAC put its source MAC address to the frame and passes to the physical layer.

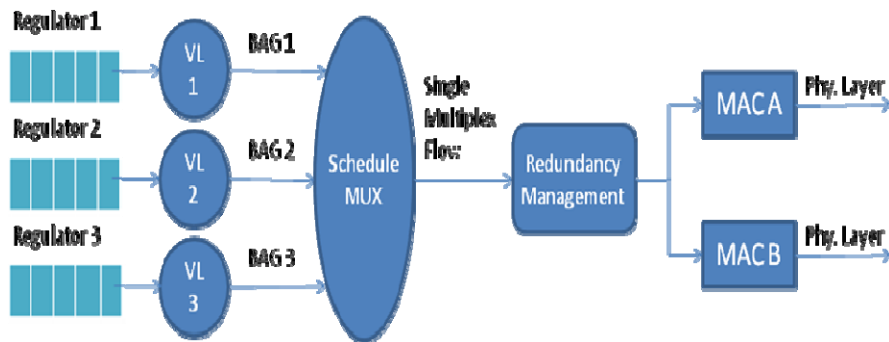


Figure 3.7 AFDX Transmitter Data Link Layer Overview

Figure 3.8 represents the data link layer operations in the transmitter point of view of AFDX End System.

- Received frames (according to MAC number, after CRC) are passed to Integrity Check.
- Integrity Check unit checks the Sequence Numbers and passes to Redundancy Management (RM) if not disabled. If RM is disabled, both frames pass to next step.
- Redundancy Management unit passes the first valid frame to demultiplexer.

- Demultiplexer delivers frames according to Virtual Links to the upper layer.

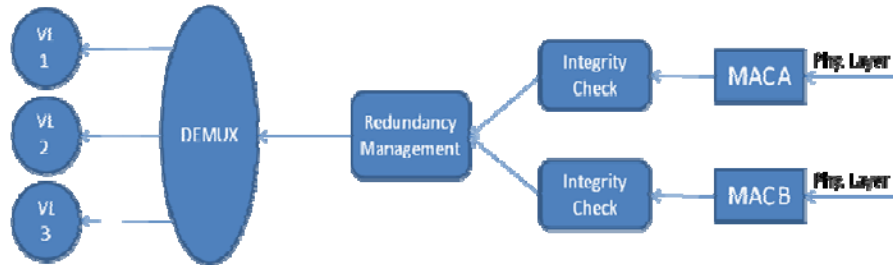


Figure 3.8 AFDX Receiver Data Link Layer Overview

### 3.4 Network Layer

From the transmission point of view; the IP network layer receives the packet from upper layer and determines whether it needs to be fragmented using the appropriate VL's Lmax value. The IP header is added, and IP checksum is calculated for each fragment.

From the reception point of view; the network layer is responsible for checking the IP checksum field and the packet reassembly, if necessary. The packet is passed to the upper (transport) layer.

AFDX frame with the bytes filled by network layer is given below;

7 (bytes)	1	6	6	2	20	9-1479	0-16	1	4	12
Preamble	S F D	Destination Address	Source Address	T y p e	IP Header	AFDX Payload	Pad	S N	F C S	IFG

Figure 3.9 AFDX Frame (Network Layer)

### 3.4.1 IP Structure

AFDX has standard IPv4 structure except not including option bytes (0 or more) between IP payload and IP destination address.

4-bits	4-bits	8-bits	16-bits	16-bits	3-bits	13-bits	8-bits	8-bits	16-bits	32-bits	32-bits	1-1479 bytes
Version	IHL	Type of service	Total length	Fragment identification	Control flag	Fragment offset	Time to live	Protocol	Header checksum	IP Source address	IP Destination address	IP Payload

Figure 3.10 IPv4 Structure

- Version: (IPv4 =4)
- IHL: IP Header Length, number of 4 byte blocks (20 bytes=5) Type of service: Not used
- Total Length: Total Length of the IP frame (header + payload)
- Fragmentation Identification: An id to the fragmented group given by transmitter
- Control flag: Not used
- Fragmentation Offset: Position of the fragment relative to original payload
- Time to live: Number of hops
- Protocol: TCP uses 6, UDP uses 17, ICMP uses 1.
- Header Checksum: Checksum for the IP header

### 3.4.2 IP Source Address:

The 32-bit IP source address should be a Class A and private Internet Unicast Address used to identify the transmitting partition associated with the End System.

Class A	Private IP Address	Network ID		Equipment ID		Partition ID	
1-bit	7-bits	8-bits		8-bits		8-bits	
"0"	"0001010"	"0000"	4-bits	3-bits	5-bits	3-bits	5-bits

Figure 3.11 IP Source Address

### 3.4.3 IP Destination Address

According to the specification, the IP destination address of the AFDX frame should be either the IP Unicast address to identify the target subscriber or an IP Multicast address compliant to the format shown in Figure below.

Class D	IP Multicast Identifier	
4-bits	28	
"1110"	"0000 1110 0000"	Virtual Link Identifier 16-bits

Figure 3.12 IP Destination Address

## 3.5 Transport Layer

The whole AFDX frame with the bytes filled by transport layer is given below;

7 (bytes)	1	6	6	2	20	8	1-1471	0-16	1	4	12
Preamble	S F D	Destinatio n Address	Source Address	Type	IP Header	UDP Header	AFDX Payload	Pad	S N	F C S	IFG

Figure 3.13 AFDX Frame (Transport Layer)

UDP Header is figured out as below;

16-bit	16-bit	16-bit	16-bit
Source Port Number	Destination Port Number	UDP Length	UDP Checksum

Figure 3.14 UDP Header

Port number allocation is defined in ARINC Specification 664, Part 4: Internet Based Address Structures and Assigned Numbers [17] according to the tables below;

Port range (decimal value)	Allocation range ARINC 664	Allocation range AFDX
0 – 1023	Administered by ICANN "Well-known" port number	Administered by ICANN "Well-known" port number
1 024 – 16 383	Registered by ICANN A664 assigned	Assigned by network manager
16 384 – 32 767	Registered by ICANN System integrator Or User defined	
32 768 – 65 535	Registered by ICANN Recommended for temporary port assignment	

Figure 3.15 Allocation of SAP and AFDX Port Numbers

Type of port	Type of communication	Port range	Commentaries
AFDX Communication port	AFDX ⇔ AFDX AFDX ⇔ Compliant network	1 024 – 65 535	Used for sampling and queuing communications
SAP	AFDX ⇔ AFDX AFDX ⇔ Compliant network	0 – 1023	Used for standard communications e.g Port 69 to open a TFTP, Data loading (ARINC 615A), SNMP, etc..
	AFDX ⇔ AFDX AFDX ⇔ Compliant network	1 024 – 65 535	Used for bi-directional communication: specific TFTP etc.,

Figure 3.16 Port Allocation Range for IP Unicast or Multicast

As AFDX is a closed network, network architect is free to choose port numbers from full range; 0 to 65535. However specification encourages the use of Dynamic/Private range of numbers only. The reason for this is to avoid possible conflicts with adjacent networks when an AFDX network is integrated with other networks through a gateway. Port numbers shall be identical in a Virtual Link.

### 3.6 Application Layer

ARINC 664 specification defines two messaging formats in the application layer; implicit and explicit. Explicit format includes format information overheads that define the transmitted message to enable the receiver to interpret the contents of the message accordingly. Because of the overheads, this format uses less part of the bandwidth effectively. On the other hand, implicit message format uses like well known service (WKS) concept of internet uses port and partition numbers to identify the contents of the payload.

Data alignment of the implicit format of AFDX is given in the figure 3.17. An AFDX payload is formed of a number of Functional Data Sets. Each Functional Data Set is formed of up to 4 Data Sets and a Functional Status Set. Functional Status Set is a 4 byte field that carries the status of each Data Set. If there are less than 4 Data Sets, the remaining Functional Status bytes should be filled with 0. For existing Data Sets the relevant byte is filled with the decimal enumerations as 0 for “No Data”, 3 for “Normal Operation”, 12 for “Functional Test” and finally 48 for “No Computed Data”. Data sets are collections of parametric data, for example altitude, airspeed, heading etc.

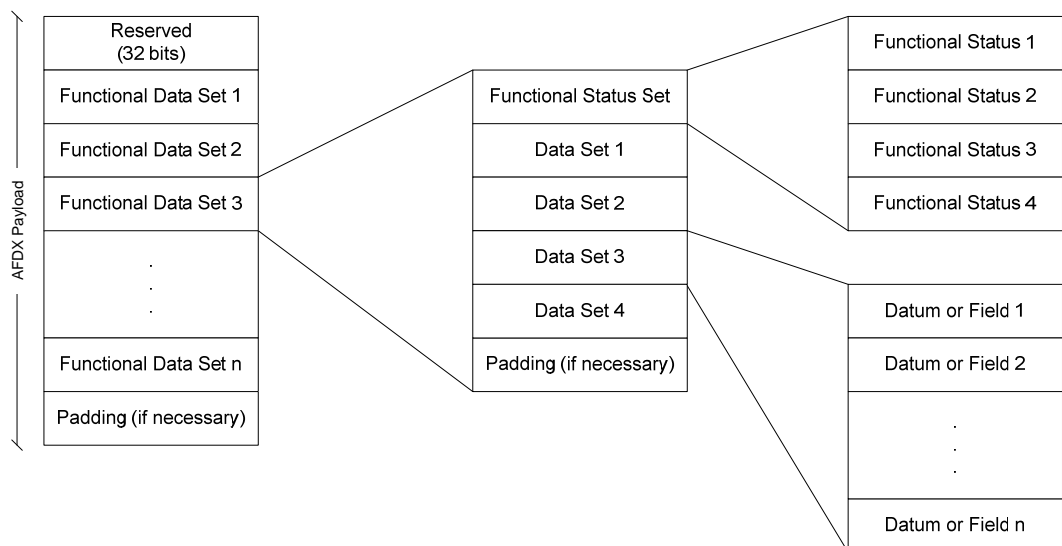


Figure 3.17 AFDX Message Structure

## **CHAPTER 4**

### **IMPLEMENTATION**

#### **4.1. Platform**

Usually system developers and system integrators choose realtime operating systems (RTOS) such as VxWorks of Wind River [18], Integrity of Green Hills [19] for flight critical software. However, for ground testing, besides RTOS's, it is also common to use other operating systems like Linux, Unix or Microsoft Windows. Most data acquisition systems (DAS) do not need realtime behavior but need data logging with timestamps.

As cost efficiency is a major aim in this thesis, the development and all tests were conducted in Microsoft Windows XP. As coding is performed with standard C/C++ libraries, it is also possible to port the stack to any operating system which is compatible with C/C++. On the other hand, it should be noticed that using realtime operating system is the necessary but not the sufficient condition for flight critical systems. As the development of the code is neither DO-178B certified (for any DAL Level), nor certifiable (not designed under rules –guidance- of certification) the software solution in this thesis is not assumed to be used in any flight critical system, but in ground systems for testing and code development purposes.



## **4.2. Tools**

### **4.2.1 Development Environment**

Executable application is developed with Microsoft Visual C++ 2008 Express Edition.

### **4.2.2 Data Link Layer Frame Access**

As the frames differ from standard TCP/IP stack of a computer, a data link layer access is necessary to implement AFDX framing. A network sniffer library for the AFDX receiver and a data link layer frame manipulation library for AFDX transmitter shall be used.

“Packet sniffers are hardware or software that read all of the packets on a communications channel; in the distant past, all such sniffing required expensive hardware whereas today the same functionality can be found with free software. Packet sniffers can display traffic in real-time, store the packets for later display, and provide detailed interpretation. Packet sniffers are useful tools for network monitoring, investigations, reconnaissance, or just to learn how the protocols work.” [20]

Risso and Degioanni [21] claim that “WinPcap architecture is the first open system for packet capture on Win32 and it fills an important gap between Unix and Windows. Furthermore WinPcap puts performance at the first place, thus it is able to support the most demanding applications.”

“Most real-time TCP/IP packet sniffing systems are based on the Unix/Linux packet capture library (libpcap) or its Windows counterpart (WinPcap). Using libpcap/winpcap software is possibly the best place to start.” [20]

“WinPcap is the industry-standard tool for link-layer network access in Windows environments: it allows applications to capture and transmit network packets bypassing the protocol stack, and has additional useful features, including kernel-level packet filtering, a network statistics engine and support for remote packet capture.” [22]

Most networking applications use sockets for network access which forces operating system to handle protocol specific jobs in the network stack. This is an easy way to program with limitations. Operating system provides filtered and stripped message to the application in reception. Also operating system fills headers of the frame itself.

Sometimes, programmer needs to access raw data on the line and to manipulate all the bit fields of the frame including headers and trailers.

The website of the WinPcap [22] addresses the purpose of WinPcap as to give this kind of access to Win32 applications and provides facilities to:

- capture raw packets, both the ones destined to the machine where it's running and the ones exchanged by other hosts (on shared media)
- filter the packets according to user-specified rules before dispatching them to the application
- transmit raw packets to the network
- gather statistical information on the network traffic

by means of a device driver that is installed inside the networking portion of Win32 kernels, plus a couple of DLLs.

“The basic structure of WinPcap retains the most important modules, a filtering machine, two buffers (kernel and user) and a couple of libraries at user level. However, WinPcap has some substantial differences in the structure and in the behavior of the capture stack, and can be seen as the evolution of BPF.” [21]

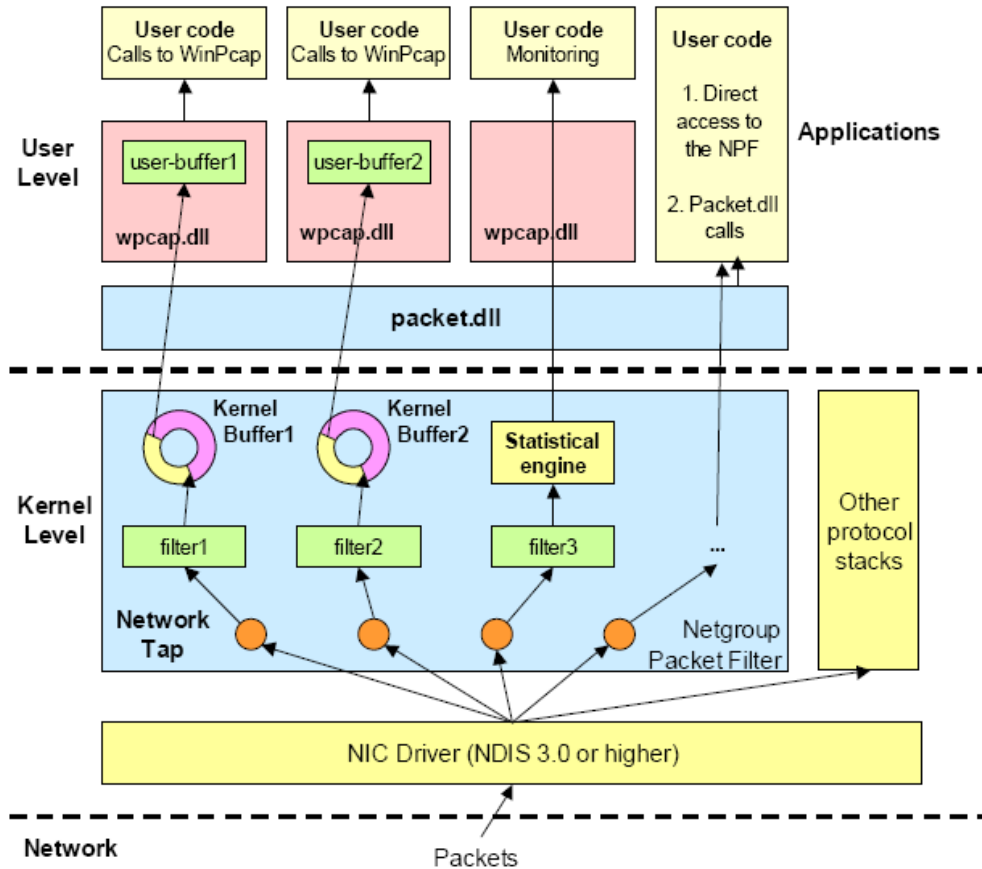


Figure 4.1. Winpcap Capture Stack

Risso and Degioanni [21] describe three modules of WinPcap in their paper. First module in the kernel level filters the packets, delivers raw packet to the user level and conducts some OS-specific code such as timestamp management.

Second module, packet.dll may be considered as an operating system abstraction module offering a system-independent API at the user level, coming under the form of Dynamic Link Libraries (DLLs).

Third module, wpcap.dll, is the other DLL which is hosted in the user level. This module provides operating system independent high-level functions such as filter generation, user-level buffering and raw packet reception and injection.

Performance analyses were conducted for different configurations using WinPcap. [21] [23]

It is mentioned in the official web site of WinPcap [22] that “Thanks to its set of features, WinPcap is the packet capture and filtering engine of many open source and commercial network tools, including protocol analyzers, network monitors, network intrusion detection systems, sniffers, traffic generators and network testers. Some of these tools, like Wireshark, Nmap, Snort, ntop are known and used throughout the networking community.”

WinPcap is also used in many academic researches and papers. [24], [25], [26], [27], [28].

### **4.3. Configuration**

To execute WinPcap library functions with Visual C++ development environment, following settings should be done.

It is assumed that the following files and folders are downloaded [22];

- WinPcap auto-installer (file: WinPcap\_4\_0\_2.exe)
- WinPcap Developer's Packs (folder: WpdPack)
- WinPcap Source Code Distributions (folder: WpcapSrc)

#### **4.3.1. Installation of DLL**

Execute file: WinPcap\_4\_0\_2.exe.

#### **4.3.2. Compiler Settings**

From project properties, compiler tab, include the following paths to “Additional Include Directories”

- \WpcapSrc \wpcap\Win32-Extensions
- \WpcapSrc \wpcap\libpcap
- \WpcapSrc \wpcap\libpcap\Win32\Include
- From project properties, preprocessor tab include the following preprocessor definitions
- WPCAP
- HAVE\_REMOTE (for remote access)

### **4.3.3. Linker Settings**

From project properties, linker tab, include the following path to “Additional Library Directories”

\ WpdPack\Lib

From project properties, linker tab, add the following options to “Command Line/Additional Options”

wpcap.lib

packet.lib

## **4. 4 AFDX Transmitter**

### **4.4.1 API Stack Overview**

AFDX Manager (AFDXTrasnmmitter.cpp) is the source code that AFDX network developer writes. This part is not intended to be a part of the thesis work but is essential to demonstrate and test the prepared code. AFDX Manager uses AFDX API to configure the AFDX stack and transmit AFDX messages.

AFDX Transmitter API (CAFDXTransmitterAPI.cpp) is the only interface between AFDX Stack and user code, namely AFDX Manager. It is a singleton class

that has an interface with all created Virtual Links; AFDX Stack and Network Interface Cards.

CVirtualLink class is the source for a Virtual Link object. API creates a Virtual Link object per a defined Virtual Link in the configuration file. Any Virtual Link includes a number of AFDXTxComPort objects as also defined in the configuration file.

AFDX Tx ComPort class (AFDXTxComPort.cpp) is the source for a AFDX Transmission Communication Port object. It includes three layers (transport layer, network layer, data link layer) for frame assembly. AFDXTxComPort object takes the AFDX payload from Virtual Link and passes it from each layer step by step.

UDP Class (CTransportLayer.cpp) adds UDP header to the payload which includes UDP Source Port Number, UDP Destination Port Number, UDP Length and calculates the UDP checksum.

IP Class (CNetworkLayer.cpp) calculates the 20 bytes of IP Header with AFDX parameters like Virtual Link number and calculates checksum.

MAC Class (CDataLinkLaye.cpp) puts the MAC header and sequence number. FCS calculation is achieved in the network interface card (NIC).

Finally AFDX Transmitter API uses Network Interface Card Interface class (NICInterface.cpp) to duplicate the frame for both A and B interfaces and to transmit with WinPcap library functions.

Figure 4.2 depicts the class hierarchy of the AFDX Transmitter Application.

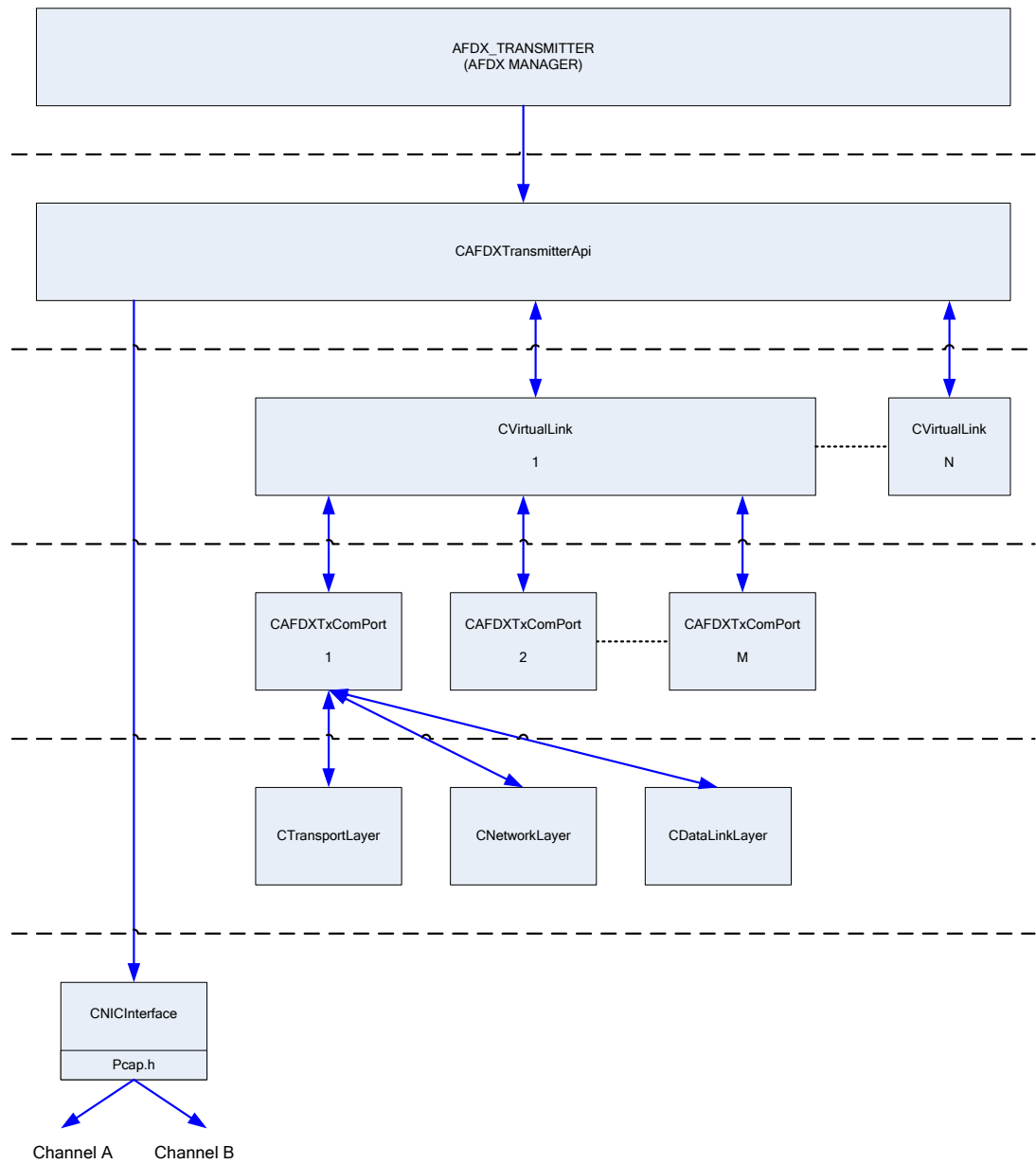


Figure 4.2. AFDX Transmitter class hierarchy

#### 4.4.2 Application Flow Chart

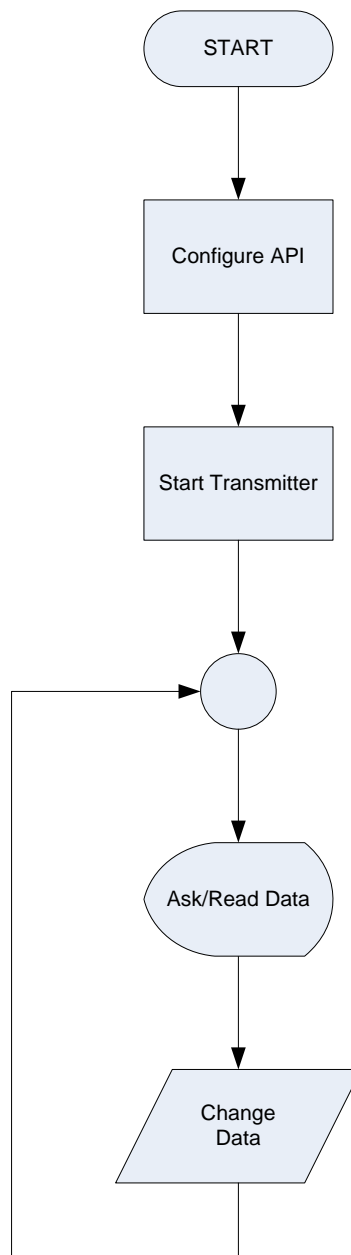


Figure 4.3. AFDX Transmitter Application Flow Chart

AFDX Manager is the code written by the user of the API. Uses only three methods of the API to configure API, to start it and to change any parametric value. Figure 4.3 shows the simple flow chart for an AFDX Transmitter application.



### 4.4.3 Application Class List

Here are the classes with brief descriptions:

- CAFDXTransmitterAPI
- CVirtualLink
- CAFDXTxComPort
- CTransportLayer
- CNetworkLayer
- CDataLinkLayer
- CNICInterface

Class Diagram of AFDX Transmitter API is given in the Figure 4.4.

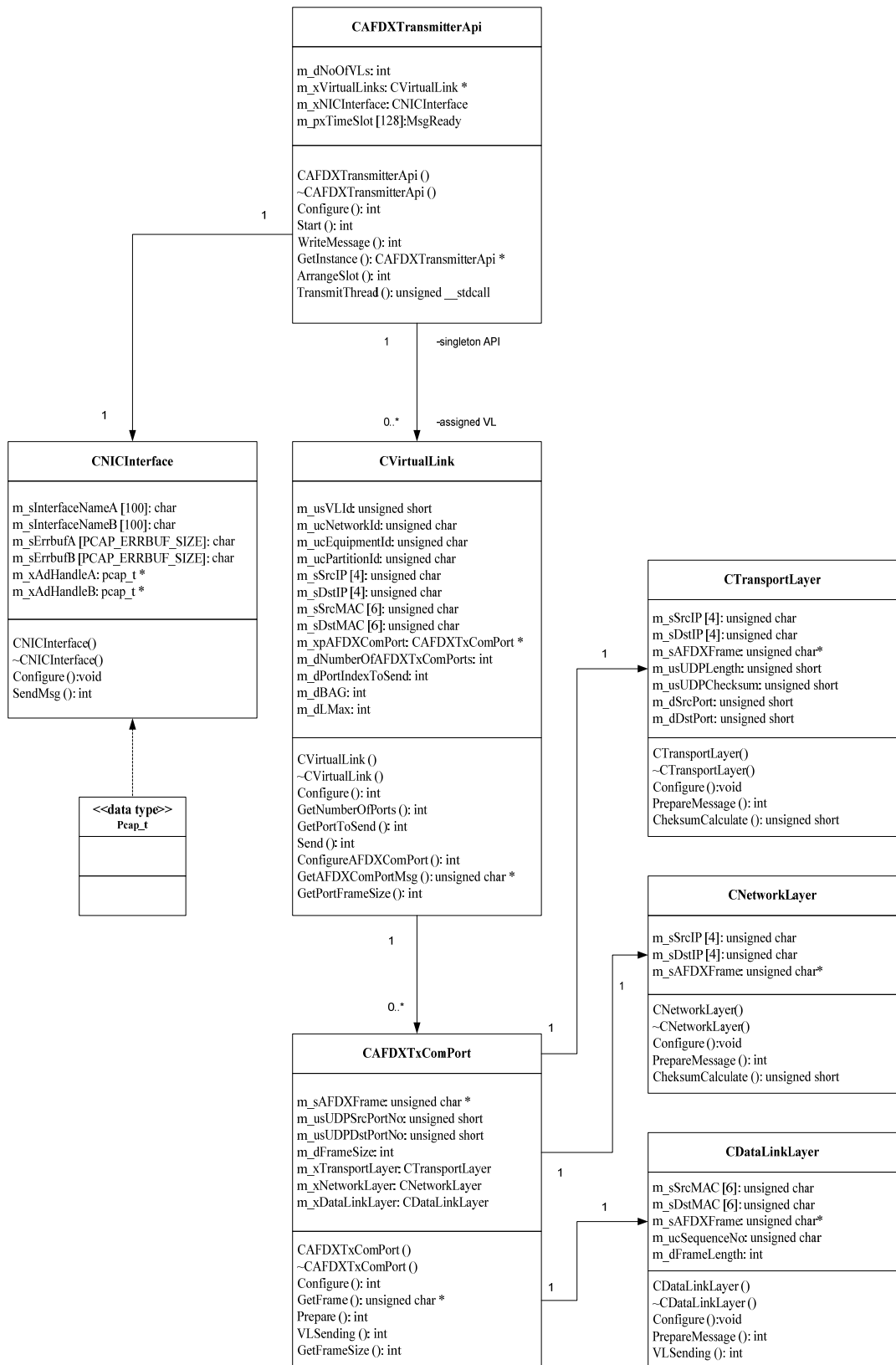


Figure 4.4. AFDX Transmitter Class Diagram

#### 4.4.3.1 CAFDXTransmitterAPI Class Reference

```
#include "AFDXTransmitterAPI.h"
#include "VirtualLink\VirtualLink.cpp"
#include "NICInterface\NICInterface.cpp"
#include <windows.h>
#include "process.h"
#include <iostream>
#include <fstream>
```

##### Public Member Functions

- CAFDXTransmitterAPI (void): The constructor of the class.
- ~CAFDXTransmitterAPI (void): The destructor of the class.
- int Configure (char \*sConfigFilePath): Configures the API.
- int Start (): Starts the transmitter.
- int WriteMessage (int dVLIndex, int dPortIndex, char \*sMsg): Writes a message into an AFDX port of a VL. Network programmer uses this method to change value of a port.

##### Static Public Member Functions

- static CAFDXTransmitterAPI \* GetInstance (): This method is used whenever a common instance of this class is wanted to be used. Because this is a singleton class, returns every time a pointer to the same instance

##### Private Member Functions

- int ArrangeSlot (int dVLNo, int dBAG): This private method is used by the API itself in configuration time. API arranges time slots for each Virtual Link with this internal method.

##### Static Private Member Functions

- static unsigned \_\_stdcall TransmitThread (void \*arg): This is the thread that is spawned for transmitter that calculates transmission times and transmits frames.

### **Private Attributes**

- int m\_dNoOfVLs: API keeps the number of virtual links in this attribute
- CVirtualLink \* m\_xVirtualLinks: List of virtual links
- CNICInterface m\_xNICInterface: Interface for the Ethernet cards.
- MsgReady m\_pxTimeSlot [128]: API predetermines the times to send each virtual link. This array of structure keeps the list of virtual links to be sent in each millisecond 0 to 127. API loops this array.

### **MsgReady Struct Reference**

- int dVLCCount: Number of virtual links to be sent for the time slot
- int \* dVLNo: Pointer to list the virtual links to be transmitted in the time slot.

#### **4.4.3.2 CVirtualLink Class Reference**

```
#include "VirtualLink.h"
#include "AFDXTxComPort\AFDXTxComPort.cpp"
```

### **Public Member Functions**

- CVirtualLink (void): Constructor of the class.
- ~CVirtualLink (void) : Destructor of the class.
- int Configure ( unsigned char ucNetworkId,  
unsigned char ucEquipmentId,  
unsigned char ucPartitionId,

```

        unsigned short usVLIId,
        int dNumberOfAFDXTxComPorts,
        int dBAG, int dLMax): AFDXTransmitterAPI

```

configures each virtual link with Configure method of the virtual link class.

- int GetNumberOfPorts (): Returns the number of AFDX communication ports defined in the virtual link.
- int GetPortToSend (): Returns the index of AFDX communication port which will be sent for current time slot. For each time slot, virtual link sends a port in a round robin manner.
- int Send (int dPortIndex, char \*sMessage): Prepares the message of the selected port to be sent.
- int ConfigureAFDXComPort ( int dIndex,
 unsigned short usUDPSrcPortNo,
 unsigned short usUDPDstPortNo,
 int dFrameSize): AFDX Transmitter
 API uses this method to configure each communication port.
- unsigned char \* GetAFDXComPortMsg (int dIndex): Calls GetFrame method of the communication port.
- int GetPortFrameSize (int dIndex): Calls GetFrameSize method of the communication port.

### **Private Attributes**

- unsigned short m\_usVLIId: Virtual link number
- unsigned char m\_ucNetworkId: Keeps Network Id in MAC source address.
- unsigned char m\_ucEquipmentId: Keeps Equipment Id in MAC source address.
- unsigned char m\_ucPartitionId: Keeps Partition Id in MAC source address.

- unsigned char m\_srcIP [4]: Source IP Address
- unsigned char m\_dstIP [4]: Destination IP Address built from virtual link no.
- unsigned char m\_srcMAC [6]: Source MAC Address build from Network Id, Equipment Id, Partition Id and a constant bit field.
- unsigned char m\_dstMAC [6]: Destination MAC Address build from Virtual Link Id and a constant bit field.
- CAFDXTxComPort \* m\_xpAFDXComPort: Pointer to AFDX Communication Ports defined in the virtual link.
- int m\_dNumberOfAFDXTxComPorts: Keeps number of ports.
- int m\_dPortIndexToSend: Keeps the next port to be sent.
- int m\_dBAG: Keeps the band allocation gap (BAG).
- int m\_dLMax: Keeps maximum frame size (Lmax).

#### 4.4.3.3 CAFDXTxComPort Class Reference

```
#include "AFDXTxComPort.h"
#include "TransportLayer\TransportLayer.cpp"
#include "NetworkLayer\NetworkLayer.cpp"
#include "DataLinkLayer\DataLinkLayer.cpp"
```

#### Public Member Functions

- CAFDXTxComPort (void): Constructor of the class.
- ~CAFDXTxComPort (void): Destructor of the class.
- int Configure ( unsigned short usUDPSrcPortNo,  
unsigned short usUDPDstPortNo,  
unsigned char \*sSrcIP,  
unsigned char \*sDstIP,  
unsigned char \*sSrcMAC,  
unsigned char \*sDstMAC,

int dFrameSize): This method is used to configure a single AFDX communication port.

- unsigned char \* GetFrame (): Returns the AFDX frame to be sent.
- int GetFrameSize (): Returns the frame size to be sent.
- int Prepare (char \*sMessage): Prepares the message to send. This method calls prepare methods of layer classes.
- int VLSending (): Calls VLSending method of the data link layer.

#### **Private Attributes**

- unsigned char \* m\_sAFDXFrame: Keeps the whole AFDX frame.
- unsigned short m\_usUDPSrcPortNo: Keeps UDP source port number.
- unsigned short m\_usUDPDstPortNo: Keeps UDP destination port number.
- int m\_dFrameSize: Keeps the AFDX frame size.
- CTransportLayer m\_xTransportLayer: An object to handle transport layer functions.
- CNetworkLayer m\_xNetworkLayer: An object to handle network layer functions.
- CDataLinkLayer m\_xDataLinkLayer: An object to handle data link layer functions.

#### **4.4.3.4 CTransportLayer Class Reference**

```
#include <TransportLayer.h>
```

#### **Public Member Functions**

- CTransportLayer (void): Constructor of the class.
- ~CTransportLayer (void): Destructor of the class.
- void Configure ( unsigned short dSrcPort,

unsigned short dDstPort,  
 unsigned char \*sSrcIP,  
 unsigned char \*sDstIP,  
 unsigned char \*sAFDXFrame): Configures the layer.

- int PrepareMessage (int dNoOfChars, char \*sMessage): Prepares the transport layer parameters of the message to sent.

### Private Member Functions

- unsigned short CheksumCalculate ( unsigned short len\_udp,  
 unsigned char src\_addr[],  
 unsigned char dest\_addr[],  
 bool padding,  
 unsigned char buff[]):  
 Calculates UDP Checksum. This is a private method and Prepare method calls this method internally.

### Private Attributes

- unsigned short m\_usUDPLength: Keeps UDP length data.
- unsigned short m\_usUDPChecksum: Keeps UDP checksum.
- unsigned short m\_dSrcPort: Keeps UDP source port number.
- unsigned short m\_dDstPort: Keeps UDP destination port number.
- unsigned char m\_sSrcIP [4]: Keeps source IP address for Checksum calculation.
- unsigned char m\_sDstIP [4] : Keeps destination IP address for Checksum calculation.
- unsigned char \* m\_sAFDXFrame: Points to the AFDX frame.

### 4.4.3.5 CNetworkLayer Class Reference

```
#include <NetworkLayer.h>
```



### **Public Member Functions**

- CNetworkLayer (void): Constructor of the class.
- ~CNetworkLayer (void): Destructor of the class.
- void Configure ( unsigned char \*sSrcIP,  
unsigned char \*sDstIP,  
unsigned char \*sAFDXFrame): Configures the layer.
- int PrepareMessage (int dUDPLength): Prepares the network layer parameters of the message to sent.

### **Private Member Functions**

- unsigned short CheksumCalculate ( unsigned short len\_ip\_header,  
unsigned short buff[]): Calculates IP Header Checksum. This is a private method and Prepare method calls this method internally.

### **Private Attributes**

- unsigned char m\_sSrcIP [4]: Keeps source IP address
- unsigned char m\_sDstIP [4]: Keeps destination IP address
- unsigned char \* m\_sAFDXFrame: Points to the AFDX frame.

#### **4.4.3.6 CDataLinkLayer Class Reference**

```
#include <DataLinkLayer.h>
```

### **Public Member Functions**

- CDataLinkLayer (void): Constructor of the class.
- ~CDataLinkLayer (void): Destructor of the class.

- void Configure ( unsigned char \*sSrcMAC,  
unsigned char \*sDstMAC,  
unsigned char \*sAFDXFrameA): Configures the layer.
- int PrepareMessage (int dLength): Prepares the data link layer parameters of the message to sent.
- int VLSending (): This method is used to prepare the sequence number. Sequence numbers are started from 0, incremented by one in each transmission of the virtual link and wraps around to 1 after 255.

#### **Private Attributes**

- unsigned char m\_sSrcMAC [6]: Source MAC Address
- unsigned char m\_sDstMAC [6]: Destination MAC Address
- unsigned char \* m\_sAFDXFrame: Points to the AFDX frame
- unsigned char m\_ucSequenceNo: Sequence number of the virtual link.
- int m\_dFrameLength: Length of the AFDX frame to be sent.

#### **4.4.3.7 CNICInterface Class Reference**

```
#include <NICInterface.h>
#include <pcap.h>
```

#### **Public Member Functions**

- CNICInterface (void): Constructor of the class.
- ~CNICInterface (void): Destructor of the class.
- int Configure (char \*sInterfaceNameA, char \*sInterfaceNameB): Configures the class. This method takes the names of the interface (Ethernet) cards and uses to send frame through.
- int SendMsg (unsigned char \*sMsg, int dLength): AFDX Transmitter API uses this method to send an AFDX frame through both interfaces. Puts the message and gives the number of bytes to be transmitted.

## Private Attributes

- `char m_sInterfaceNameA [100]`: Name of the Ethernet Card for interface A
- `char m_sInterfaceNameB [100]`: Name of the Ethernet Card for interface B
- `char m_sErrbufA [PCAP_ERRBUF_SIZE]`: An internally used buffer to keep possible errors for operations with Ethernet card for interface A
- `char m_sErrbufB [PCAP_ERRBUF_SIZE]`: An internally used buffer to keep possible errors for operations with Ethernet card for interface B
- `pcap_t * m_xAdHandleA`: Handle for interface A
- `pcap_t * m_xAdHandleB`: Handle for interface B

### 4.4.4 Detailed Explanation of the AFDX Transmitter Application

Figure 4.5 shows the sequence diagram of the application with AFDX Transmitter program.

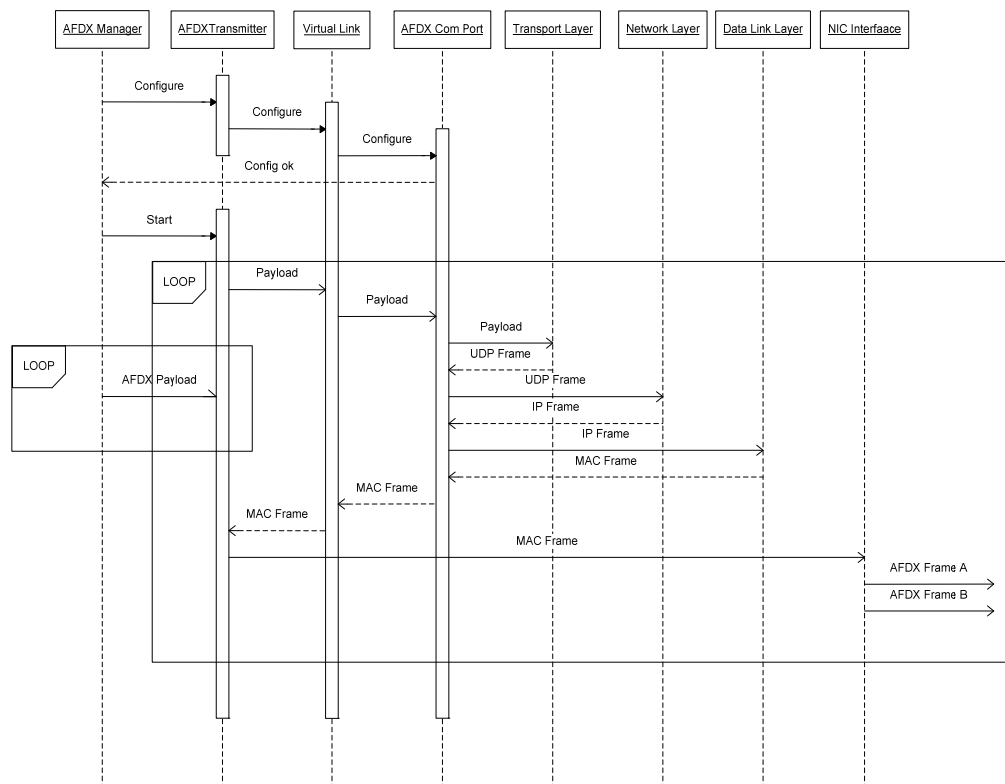
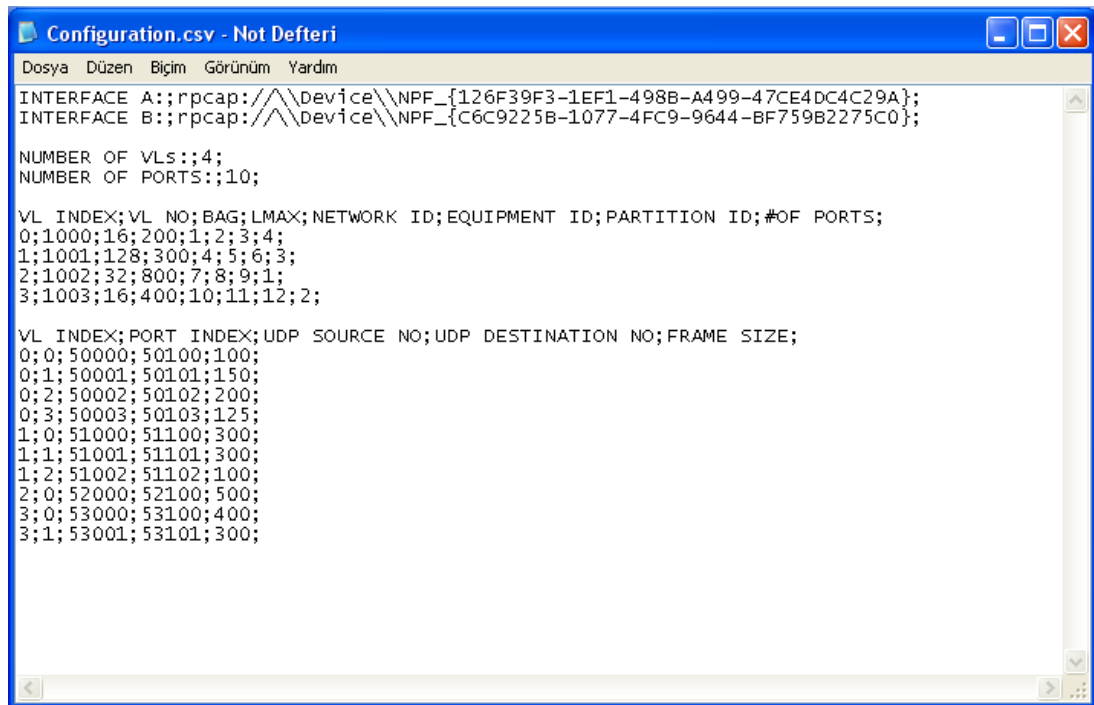


Figure 4.5. AFDX Transmitter Sequence Diagram

AFDX Manager is the source code written by the AFDX network developer. AFDX Manager configures the API with a configuration file in csv (comma-separated values) file format.

Figure 4.6 shows the appearance of a simple configuration file on a text editor. Second items of the first two row gives the interface names of the Ethernet cards, first for Interface A and second for Interface B. Number of Virtual Links and total number of AFDX communication ports are given below the interface names. Virtual Link number, BAG, Lmax, Network Id, Equipment Id, and number of ports for each Virtual Link information is given row by row as shown in the figure. After the end of the Virtual Link based configurations, port by port configurations start. Each port is identified by UDP Source / Destination number and frame size. Frame sizes for each AFDX communication port should be less than or equal to Lmax of relevant Virtual Link. Lmax for each Virtual Link should be selected no more than 1500 bytes.



```

Configuration.csv - Not Defteri
Dosya Düzen Biçim Görünüm Yardım
INTERFACE A::rpcap://\\Device\\NPF_{126F39F3-1EF1-498B-A499-47CE4DC4C29A};
INTERFACE B::rpcap://\\Device\\NPF_{C6C9225B-1077-4FC9-9644-BF759B2275C0};

NUMBER OF VLS::4;
NUMBER OF PORTS::10;

VL INDEX;VL NO;BAG;LMAX;NETWORK ID;EQUIPMENT ID;PARTITION ID;#OF PORTS;
0;1000;16;200;1;2;3;4;
1;1001;128;300;4;5;6;3;
2;1002;32;800;7;8;9;1;
3;1003;16;400;10;11;12;2;

VL INDEX;PORT INDEX;UDP SOURCE NO;UDP DESTINATION NO;FRAME SIZE;
0;0;50000;50100;100;
0;1;50001;50101;150;
0;2;50002;50102;200;
0;3;50003;50103;125;
1;0;51000;51100;300;
1;1;51001;51101;300;
1;2;51002;51102;100;
2;0;52000;52100;500;
3;0;53000;53100;400;
3;1;53001;53101;300;

```

Figure 4.6. AFDX Transmitter Configuration File

When the `Configure()` method of the `AFDXTranmitterAPI` is called by the application with a path of the configuration file, API opens the file and starts to configure itself, each Virtual Link and each port. Also with this configuration call, API prepares the schedule to transmit each port most efficiently.

In a scheduling work, it is a common and useful way to divide frames into minor and major frames when there are frames with different frequencies. Major frame defines the period of overall messaging, which repeats the same sequence of messages or processes. The minor frame defines the period inside the major frame. Minor frame is calculated with the greatest common divisor of all periods and the major frame is calculated with the least common multiple of all periods.

AFDX defines BAGs as 1 ms, 2 ms, 4 ms, 8 ms, 16 ms, 32 ms 64 ms and 128 ms, the minor frame for the schedule is 1 ms (the greatest common divisor of possible BAGs) and the major frame is 128 ms (the least common multiple of possible BAGs). The idea is ordering Virtual Links from smallest to biggest BAG and putting them to the empty slots (minor frames) one by one. A 1 ms BAG Virtual Link should be repeated in each minor frame whereas a 4 ms BAG Virtual Link should be inserted once for 4 minor frames. Each time inserting a Virtual Link, `ArrangeSlot()` method checks for best minor frame group to use the line effectively.

After the configuration is completed, API is ready to start to send AFDX frames. `Start()` method initiates the periodical transmission by spawning `TransmitThread()` method.

While the scheduler is running and transmitting AFDX frames, user application can change the values of the frames with `Write()` method asynchronously by pointing the message with Virtual Link index and port index.

AFDX scheduler uses performance counter of Windows to calculate the time to transmit frame. `QueryPerformanceCounter()` function gives the number of CPU clocks after the computer is powered on. As the time between to clock cycles differ

for different CPUs, QueryPerformanceFrequency() function which gives frequency of CPU is used to calculate time independent of the CPU speed. At the beginning of a minor frame, soft real time thread reads the starting clock ticks and calculates the end of the minor frame with the following formula;

$$EndClock = StartClock + \frac{Frequency}{1000}$$

As the frequency is the ticks per second and the duration of the minor frame is 1 ms, adding the division of frequency by 1000 to the clock number measured at the beginning gives the number of clocks for the end of the minor frame.

In a loop of 128 minor frames, scheduler measures the starting ticks and calculates the ending ticks, transmits the frames of Virtual Links for that minor frame and waits until 1 ms is completed. While transmitting a frame for a Virtual Link, each Virtual Link object counts the next AFDX communication port to be serviced in a round robin manner. Scheduler repeats 128 step of loop forever.

## **4.5 AFDX Receiver**

### **4.5.1 API Stack Overview**

AFDX Manager (AFDXReceiver.cpp) is the source code that AFDX network developer writes. This part is not intended to be a part of thesis work but is essential to demonstrate and test the prepared code. AFDX Manager uses AFDX API to configure the AFDX stack and receive AFDX messages.

AFDX API (CAFDXAPI class) is the only interface between AFDX Stack and user code, namely AFDX Manager. It is a singleton class that has an interface with all created Virtual Links, AFDX Stack.

AFDX Stack (CvirtualLink class) is an object of a Virtual Link. It has three layers (transport layer, network layer and data link layer) for frame disassembly and WinPcap library functions to receive the message. It takes the raw data from the interface, passes it from each layer step by step and supplies AFDX payload to API.

MAC Class (CDataLinkLayer class) checks the MAC header and sequence number. FCS calculation is achieved in the network interface card (NIC).

IP Class (CNetworkLayer class) checks the 20 bytes of IP Header with AFDX parameters and checksum.

UDP Class (CTransportLayer class) checks UDP header and the UDP checksum.

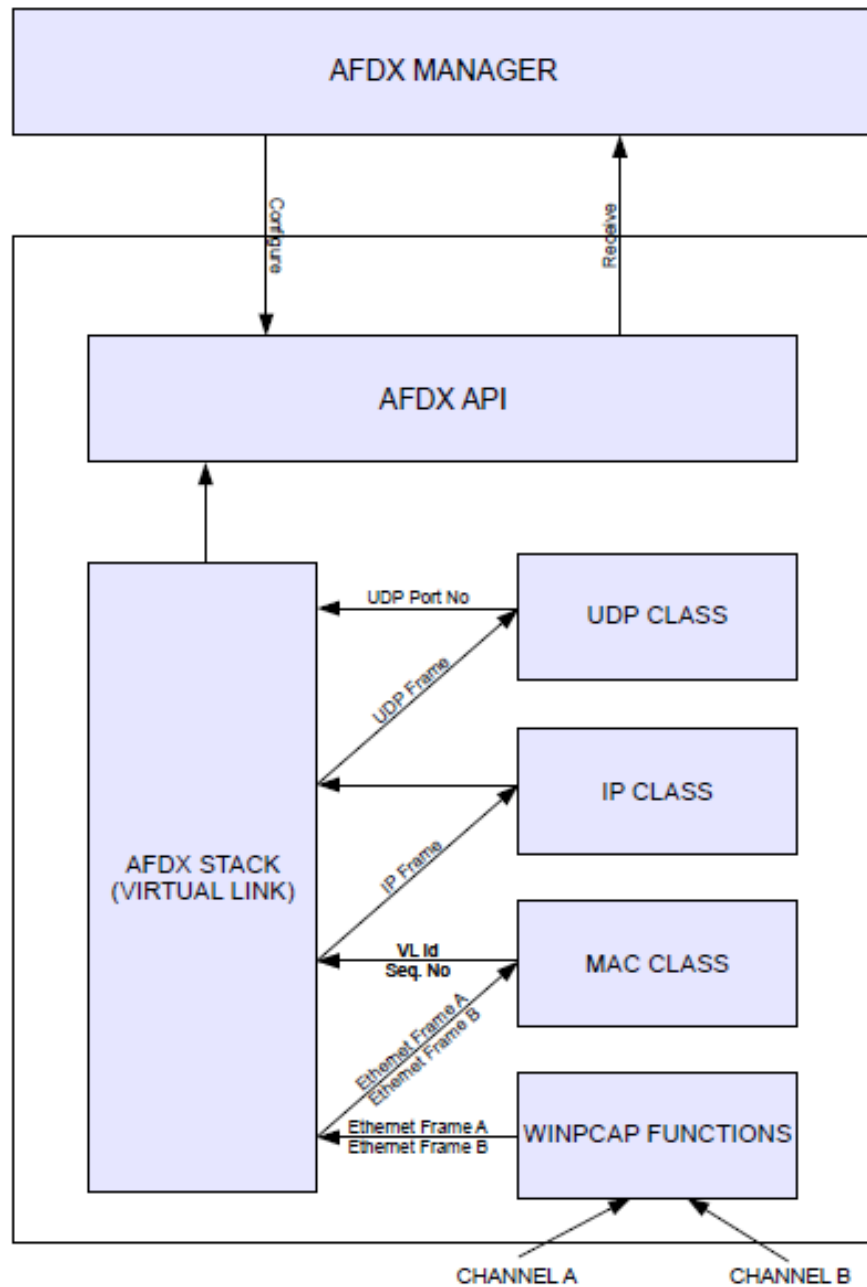


Figure 4.7. AFDX Receiver API Stack Overview



#### 4.5.2 Application Flow Chart

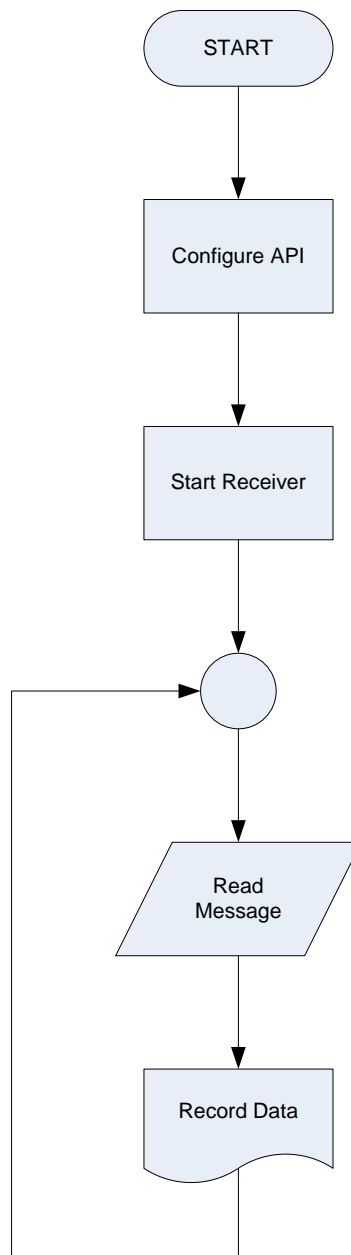


Figure 4.8. AFDX Receiver Application Flow Chart

AFDX Manager is the code written by the user of the API. Uses only three methods of the API to configure API, to start it and to read message from a Virtual Link. Figure 4.8 shows the simple flow chart for an AFDX Receiver application.

### 4.5.3 Application Class List

Here are the classes with brief descriptions:

- CAFDXReceiverAPI
- CVirtualLink
- CTransportLayer
- CNetworkLayer
- CDataLinkLayer

Class Diagram of AFDX Receiver API is given in the Figure 4.9.



Figure 4.9. AFDX Receiver Class Diagram

#### 4.5.3.1 CAFDXReceiverAPI Class Reference

```
#include <AFDXReceiverAPI.h>
#include "VirtualLink\VirtualLink.cpp"
#include <iostream>
#include <fstream>
#include <windows.h>
#include "process.h"
#include <time.h>
```

##### Public Member Functions

- `int Configure (char *sConfigFilePath):` Configures API with a configuration file. Input argument is path of the configuration file.
- `int Start ():` Starts the receiver API.
- `int Receive ( char *cInterface,`  
                  `unsigned short *dVLIId,`  
                  `char *sMsg`  
                  `int *dMsgLen`  
                  `unsigned char *ucSeqNo`  
                  `unsigned short *usUDPSrcPort,`  
                  `unsigned short *usUDPDstPort,`  
                  `struct timeval *ts):` Receiver method returns collected and controlled messages from API. Input parameters are pointers to let API fill with the receive results.

##### Static Public Member Functions

- `static CAFDXReceiverAPI * GetInstance ():` This method is used whenever a common instance of this class is wanted to be used. Because this is a singleton class, returns every time a pointer to the same instance.

## Private Member Functions

- `int DeviceConfiguration (char *sInterfaceNameA, char *sInterfaceNameB):`  
This private method is used by the API itself in configuration time. API arranges time slots for each virtual link with this internal method.

## Static Private Member Functions

- `static unsigned __stdcall ReceiveThreadA (void *arg):` API spawns this thread with `Start()` method for interface A to retrieve packets from Ethernet card.
- `static unsigned __stdcall ReceiveThreadB (void *arg) :` API spawns this thread with `Start()` method for interface B to retrieve packets from Ethernet card.

## Private Attributes

- `int m_dNoOfVLs:` Number of virtual links defined for API.
- `CVirtualLink * m_xVirtualLinks:` List of virtual links.
- `char m_sInterfaceNameA [100]:` Name of the Ethernet Card for interface A.
- `char m_sInterfaceNameB [100]:` Name of the Ethernet Card for interface B.
- `char m_sErrbufA [PCAP_ERRBUF_SIZE]:` An internally used buffer to keep possible errors for operations with Ethernet card for interface A.
- `char m_sErrbufB [PCAP_ERRBUF_SIZE]:` An internally used buffer to keep possible errors for operations with Ethernet card for interface B.
- `pcap_t * m_xAdHandleA:` Handle for interface A.
- `pcap_t * m_xAdHandleB:` Handle for interface B.
- `unsigned char m_sAFDXFrameA [1500]:` A pointer to keep received frames from interface A.
- `bool m_bFreshA:` Thread for interface A sets this flag whenever a message is ready for interface A.
- `unsigned short m_dVLOrderA:` Keeps the index of the virtual link that last received message from interface A belongs to.

- `int m_dMsgLengthA`: Keeps the length of the last received message from interface A.
- `unsigned char m_ucSeqNoA`: Keeps the sequence number of the last received message from interface A.
- `unsigned short m_usUDPSrcPortA`: Keeps the UDP source port number of the last received message from interface A.
- `unsigned short m_usUDPDstPortA`: Keeps the UDP destination number of the last received message from interface A.
- `unsigned char m_sAFDXFrameB [1500]`: A pointer to keep received frames from interface B.
- `bool m_bFreshB`: Thread for interface B sets this flag whenever a message is ready for interface B.
- `unsigned short m_dVLOrderB`: Keeps the index of the virtual link that last received message from interface B belongs to.
- `int m_dMsgLengthB`: Keeps the length of the last received message from interface B.
- `unsigned char m_ucSeqNoB`: Keeps the sequence number of the last received message from interface B.
- `unsigned short m_usUDPSrcPortB`: Keeps the UDP source port number of the last received message from interface B.
- `unsigned short m_usUDPDstPortB`: Keeps the UDP destination number of the last received message from interface B.

#### **ArgForThread Struct Reference**

- `CAFDXReceiverAPI* xThis`: A pointer to the `AFDXReceiverAPI` object
- `int dNwIndex`: Network index; 0 for A and 1 for B

#### **4.5.3.2 CVirtualLink Class Reference**

```
#include <VirtualLink.h>
#include "TransportLayer\TransportLayer.cpp"
```

```
#include "NetworkLayer\NetworkLayer.cpp"
#include "DataLinkLayer\DataLinkLayer.cpp"
#include "string.h"
#include <pcap.h>
```

## **Public Member Functions**

- CVirtualLink (void): Constructor of the class.
- ~CVirtualLink (void): Destructor of the class
- int Configure (unsigned short usVLId): Configures the virtual link object with virtual link number.
- int ReceiveA ( int dNoOfChars,  
char \*sMessage,  
unsigned char \*ucSeqNo,  
unsigned short \*usUDPSrcPort,  
unsigned short \*usUDPDstPort): Receive method for interface A. This method directly retrieves messages from interface card and organizes the AFDX stack functions.
- int ReceiveB ( int dNoOfChars,  
char \*sMessage,  
unsigned char \*ucSeqNo,  
unsigned short \*usUDPSrcPort,  
unsigned short \*usUDPDstPort): Receive method for interface B. This method directly retrieves messages from interface card and organizes the AFDX stack functions.
- unsigned short GetVLId (): Returns the virtual link Id.
- bool RedundancyCheck (unsigned char dSN): This method is used for redundancy check function of AFDX.

## **Private Attributes**

- unsigned char m\_ucNetworkId: Network Id for the received message retrieved from source MAC address.
- unsigned char m\_ucEquipmentId: Equipment Id for the received message retrieved from source MAC address.
- unsigned char m\_ucPartitionId: Partition Id for the received message retrieved from source MAC address.
- unsigned short m\_usVLId: Keeps the virtual link Id.
- unsigned short m\_usUDPSrcPortNo: Keeps UDP source port number of the AFDX frame.
- unsigned short m\_usUDPDstPortNo: Keeps UDP destination port number of the AFDX frame.
- unsigned char m\_sSrcIP [4]: Keeps source IP address
- unsigned char m\_sDstIP [4]: Keeps destination IP address
- unsigned char m\_sSrcMAC [6]: Keeps source MAC address
- unsigned char m\_sDstMAC [6]: Keeps destination MAC address
- unsigned char m\_sAFDXFrameA [1500]: A pointer to receive AFDX frames from interface A.
- unsigned char m\_sAFDXFrameB [1500] : A pointer to receive AFDX frames from interface B.
- CTransportLayer m\_xTransportLayerA: An object to handle transport layer functions for interface A.
- CNetworkLayer m\_xNetworkLayerA: An object to handle network layer functions for interface A.
- CDataLinkLayer m\_xDataLinkLayerA: An object to handle data link layer functions for interface A.
- CTransportLayer m\_xTransportLayerB: An object to handle transport layer functions for interface B.
- CNetworkLayer m\_xNetworkLayerB: An object to handle network layer functions for interface B.
- CDataLinkLayer m\_xDataLinkLayerB: An object to handle data link layer functions for interface B.



- unsigned char m\_ucSN: Sequence number of the last received AFDX frame for this virtual link.

#### 4.5.3.3 CTransportLayer Class Reference

#include < TransportLayer.h>

##### Public Member Functions

- CTransportLayer (void): Constructor of the class.
- ~CTransportLayer (void): Destructor of the class.
- void Configure (unsigned char \*sDstIP, unsigned char \*sAFDXFrame): This method is used to configure transport layer for the owner virtual link of this object.
- int CheckMessage (int dNoOfChars): Realizes message control operation for transport layer.
- unsigned short GetSrcPort (): Returns the UDP source port number of the AFDX frame received.
- unsigned short GetDstPort (): Returns the UDP destination port number of the AFDX frame received.

##### Private Member Functions

- unsigned short CheksumCalculate (      unsigned short len\_udp,  
   unsigned char    src\_addr[],  
   unsigned char dest\_addr[],  
   bool padding,  
   unsigned      char      buff[]):

Calculates the UDP checksum for received message.

##### Private Attributes

- unsigned short m\_usUDPLength: Keeps the UDP message length field of the received AFDX frame
- unsigned short m\_usUDPChecksum: Keeps the checksum field of the received AFDX frame.
- unsigned short m\_dSrcPort: Keeps UDP source port number field of the received AFDX frame.
- unsigned short m\_dDstPort: Keeps UDP destination port number field of the received AFDX frame.
- unsigned char m\_sSrcIP [4] : Keeps IP source address field of the received AFDX frame.
- unsigned char m\_sDstIP [4] : Keeps IP destination address field of the received AFDX frame.
- unsigned char \* m\_sAFDXFrame: Points to the received AFDX frame.

#### **4.5.3.4 CNetworkLayer Class Reference**

```
#include < NetworkLayer.h>
```

##### **Public Member Functions**

- CNetworkLayer (void): Constructor of the class.
- ~ CNetworkLayer (void): Destructor of the class.
- void Configure (unsigned char \*sDstIP, unsigned char \*sAFDXFrame): This method is used to configure network layer for the owner virtual link of this object.
- int CheckMessage (int dNoOfChars): Realizes message control operation for network layer.

##### **Private Member Functions**



## Private Attributes

- unsigned char m\_sSrcMAC [6]: Keeps source MAC address field of the received AFDX frame.
- unsigned char m\_sDstMAC [6] : Keeps destination MAC address field of the received AFDX frame.
- unsigned char \* m\_sAFDXFrame: Points to the received AFDX frame.
- unsigned char m\_ucSequenceNo: Sequence number of the last received AFDX frame from the owner interface, A or B. Used for integrity check.
- unsigned char m\_ucFalseSeqCount: Count of the false sequence number reception from the owner interface, A or B.

#### 4.5.4 Detailed Explanation of the AFDX Receiver Application

Figure 4.10 shows the sequence diagram of the application with AFDX Receiver program.

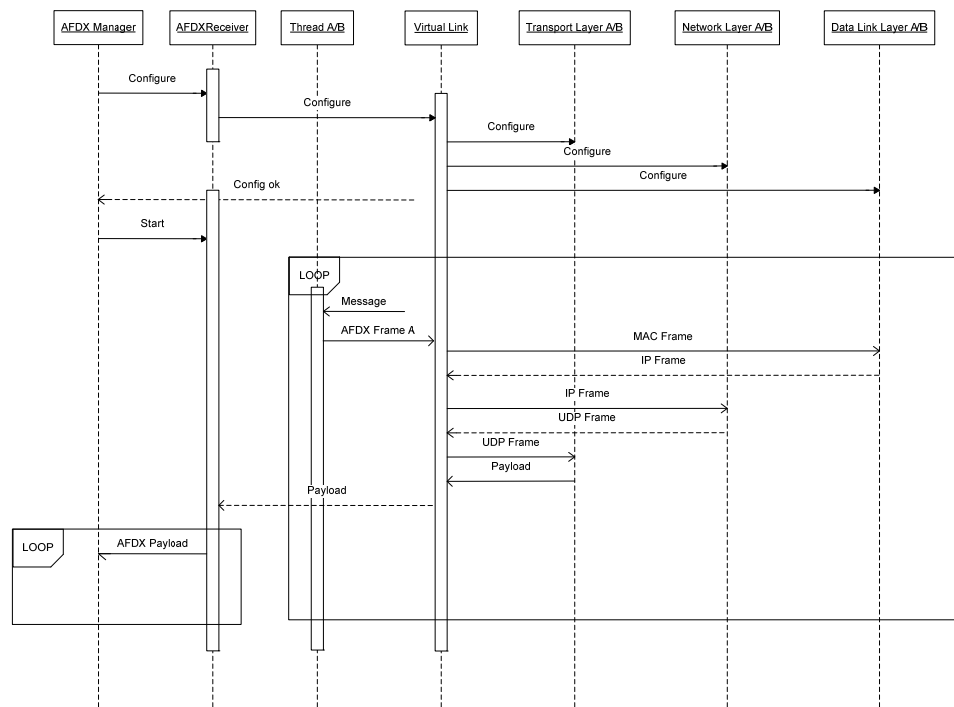


Figure 4.10 AFDX Receiver Sequence Diagram

AFDX Manager is the source code written by the AFDX network developer. AFDX Manager configures the API with a configuration file in csv (comma-separated values) file format.

The details of the configuration file and the fields will not be explained in details as the configuration file used for the AFDX Receiver application is same as the AFDX transmitter application.

When the Configure() method of the AFDXReceiverAPI is called by the application with a path of the configuration file, API opens the file and starts to read lines and configures itself and each Virtual Link.

After the configuration is completed, API is ready to start to receive AFDX frames. Start() method initiates listening both Ethernet cards in forever loops by spawning ReceiveThreadA () and ReceiveThreadB () methods.

ReceiveThreadA () and ReceiveThreadB () methods listen network interface cards A and B respectively non-persistently. Any thread, after receiving AFDX frame from the interface, checks the frame if it is an AFDX frame and then checks the Virtual Link identifier with the Virtual Link identifier list configured. If the Virtual Link identifier of the frame is valid for the End System, Virtual Link object passes the frame from each layer; data link layer, network layer and transport layer with CheckMessage() methods of each object respectively.

The first control is in the data link layer. Data link layer object checks data link layer properties of the message and performs integrity check as well. Integrity check is as defined in the specification; accept any frame with a sequence number one or two greater than last received frame.

After passing the frame from each layer, ReceiveThread sets the freshness flag (m\_bFreshA for thread A and m\_bFreshB for thread B) of message. On parallel, Receive() method of AFDXReceiverAPI looks for freshness flags of both networks and when any fresh flag is detected, passes received message from redundancy

management with RedundancyCheck() method of CVirtualLink object and returns the result to the application program. Redundancy management is performed according to specification; first valid frame wins.

## **CHAPTER 5**

### **PERFORMANCE**

AFDX Specification is developed for flight critical applications which impose determinism. In order to say that any AFDX End System is deterministic, the End System should meet the defined performance criteria. Specification defines three performance criteria for an End System of which first two are applicable for the receiving End System and all of the three are applicable for the transmitting End System; latency, MAC constraints and jitter.

#### **5.1. Latency**

AFDX specification defines latency for a transmitting End System as the time between when a message is ready to be transmitted and the completion of the transmission. Starting point of measurement is defined as the last bit of application data is available to the AFDX stack and the ending point is defined as the last bit of the corresponding AFDX frame is transmitted on the physical media.

Latency for a receiving End System is defined as the time between when a message is physically received from the line and the data is ready for application. Starting point of measurement is defined as the last bit of an Ethernet frame is received on the physical media attachment and the ending point is defined as the last bit of the corresponding data is available to the end-system hosted application.

Specification divides the latency into two parts; technological latency and frame delay and bound the technological latency as 150 micro seconds for both transmitting and receiving End Systems.

To perform the latency test for the developed application, transmitter and the receiver nodes are implemented in the same computer. Interface A and interface B of the computer is connected to each other with a cross Ethernet cable. The basic idea to use the same computer was to find an opportunity to use the clock tick counter of the CPU as the common time reference. Transmitter application fills the message's first 8 bytes with the performance clock counter value which is a 64 bit unsigned integer and transmits the frame to the media. On the other hand, when the receiving application receives the valid message after all processes are completed, it records the message in a text file with the CPU clocks retrieved just after the reception of the frame.

Specification says that technological latency is independent of traffic load and should be measured with an empty mailbox with no conflicting resource access. An empty mailbox means that there is no waiting message in the queue to transmit for the transmitter side and there is no waiting message to process for the receiver side. That's why test is performed with one Virtual Link which has a BAG of 128 ms and a message length of 1500 bytes which is the largest possible frame length.

To calculate the latency, high and low parts of the CPU clock counts that are received from the message which were filled by the transmitter were subtracted from the high and low parts of the CPU clock counts that written by the receiver. The result is divided by CPU frequency which is 2664100000 Hz for the testing computer and multiplied by 1000000 to obtain the results in micro seconds. As the latency definition of the specification, bit time is subtracted from the result. For a 10 Mbps link, a single bit lasts 0,1 micro seconds and 12000 bits which is the multiplication of 1500 bytes by 8 bits last 1200 micro seconds on the line. Total latency is obtained by subtracting bit time from the recorded time delta between transmitter and receiver.



According to the test results maximum latency is measured as 162,19 micro seconds which is the cumulative technological latency of transmitter and receiver. As the stack structures of transmitter and the receiver are very similar, it can be assumed that this latency is shared equally by each application. With this assumption maximum latency for both the transmitting and receiving End Systems is obtained as 81,10 micro seconds, minimum latency is obtained as 51,08 micro seconds and finally average latency is obtained as 51,08 micro seconds. The results are lower than the maximum tolerable technological latency which is defined as 150 micro seconds. Following table shows a small sample of the results of latency test. First column of the Table 5.1 is the most significant word of the number of CPU ticks on the transmitter side just before transmission and second column is the least significant word of the same data. Third and fourth columns also establish the time stamp when the application receives the message.

Table 5.1 Transmitter Latency Test Results Sample

<b>TX HI PART (tick count)</b>	<b>TX LO PART (tick count)</b>	<b>RX HI PART (tick count)</b>	<b>RX LO PART (tick count)</b>	<b>HI DELTA (usec)</b>	<b>LO DELTA (usec)</b>	<b>FRAME DELAY (usec)</b>	<b>CUM. TECH. LATENCY (usec)</b>	<b>TX TECH. LATENCY (usec)</b>
1643	445905072	1643	449457000	0	1333,26	1200	133,26	66,63
1643	786912656	1643	790445072	0	1325,93	1200	125,93	62,97
1643	1127915232	1643	1131527976	0	1356,08	1200	156,08	78,04
1643	1468919928	1643	1472462416	0	1329,71	1200	129,71	64,86
1643	1809923432	1643	1813423184	0	1313,67	1200	113,67	56,84
1643	2150927936	1643	2154408672	0	1306,53	1200	106,53	53,27
1643	2491933000	1643	2495438296	0	1315,75	1200	115,75	57,88
1643	2832937800	1643	2836443480	0	1315,90	1200	115,90	57,95
1643	3173942680	1643	3177439896	0	1312,72	1200	112,72	56,36
1643	3514947040	1643	3518528824	0	1344,46	1200	144,46	72,23
1643	3855952320	1643	3859462320	0	1317,52	1200	117,52	58,76
1643	4196957016	1643	4200443280	0	1308,61	1200	108,61	54,30
1644	242994336	1644	246495384	0	1314,16	1200	114,16	57,08
1644	583999088	1644	587468168	0	1302,16	1200	102,16	51,08
1644	925004160	1644	928496680	0	1310,96	1200	110,96	55,48
1644	1266008552	1644	1269498504	0	1309,99	1200	109,99	55,00
1644	1607014712	1644	1610505536	0	1310,32	1200	110,32	55,16
1644	1948020336	1644	1951617040	0	1350,06	1200	150,06	75,03
1644	2289022992	1644	2292590272	0	1339,02	1200	139,02	69,51
1644	2630028928	1644	2633594968	0	1338,55	1200	138,55	69,28
1644	2971035872	1644	2974536944	0	1314,17	1200	114,17	57,08
1644	3312037480	1644	3315535056	0	1312,85	1200	112,85	56,43
1644	3653044200	1644	3656529024	0	1308,07	1200	108,07	54,03
1644	3994047072	1644	3997527928	0	1306,58	1200	106,58	53,29
1645	40084592	1645	43579392	0	1311,81	1200	111,81	55,91

## 5.2. MAC Constraints

AFDX specification requires an End System to be able to both transmit and receive frames back to back with fixed inter frame gap and full bandwidth. The most challenging case for full bandwidth transmission is expressed as the smallest frame length with maximum number of frames. For instance for a 10 Mbps interface, with 64 bytes frames and 14800 frames per second occupies the full bandwidth. A frame with 64 bytes, added 12 bytes of inter frame gap, 7 bytes of preamble and one byte of start frame delimiter occupies 84 byte time in the line and lasts 0,672 micro seconds. That is the reason why about 14800 frames fill the whole bandwidth ( $14800 \times 0,672$  micro seconds = 9945,6 micro seconds per second). Specification also underlines that this performance requirement could be relaxed for transmission with the careful consideration of the designer.

In order to test MAC constraints requirement, configuration files were prepared for most challenging scenario described above. 14 Virtual Links each with 64 bytes of frames are configured. The BAG for each frame was selected as 1 millisecond. Therefore 14 Virtual Links occupied 9,41 Mbps. 10000 sample of transmitted frames for each configuration was collected with AFDXReceiver application and time gap between two consecutive Virtual Links are measured. As the BAG was selected as 1ms, total sample took 10 seconds. AFDXTransmitter application copied 8 bytes of (64 bit integer) CPU tick count to the first 8 bytes of the transmitting message in order to highlight the time difference between two transmitting frame. Figure 5.1 gives time gap of consecutive Virtual Links with respect to sample number. As the BAG is selected as 1 millisecond, it is expected to have 1 millisecond of gaps between frames displayed. The outer and dark blue points in the Figure 5.1 are formed with the time difference of two consecutive frames that were received by the receiver which was obtained with the reception time stamps, and the inner and pink points are formed with the time difference of two consecutive frames that were transmitted which was derived from the first 8 bytes of message.

The baud rate for the tests is selected as 10Mbps with the motivation of current real applications that all use 10 Mbps, A380 of Airbus and A400M of EADS, formerly Airbus Military.

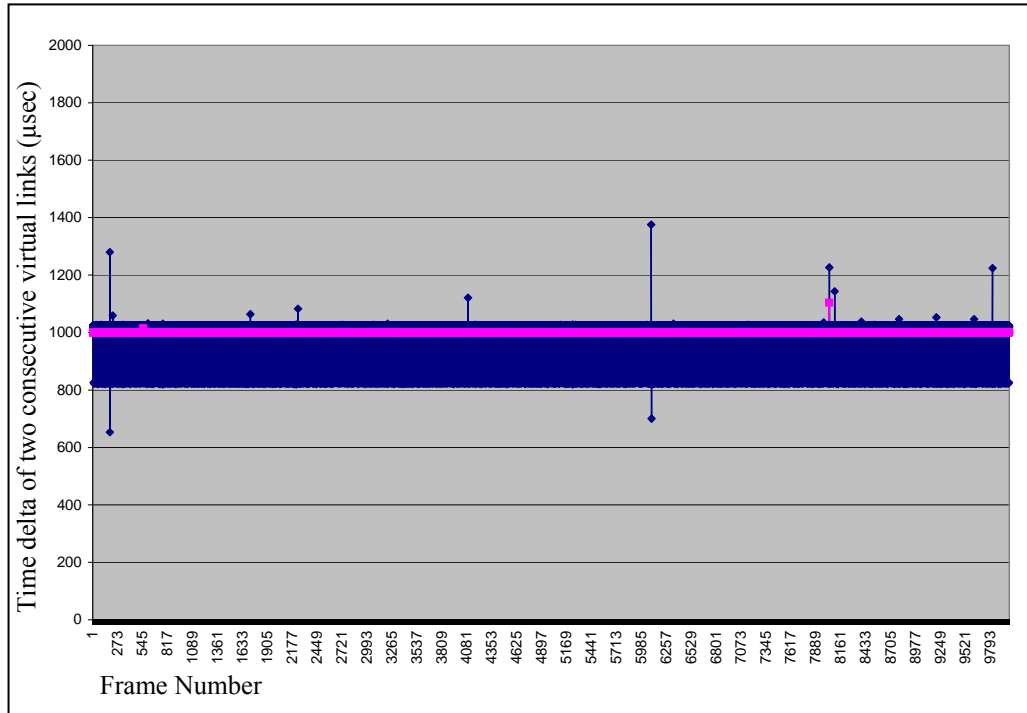


Figure 5.1 Full Bandwidth Usage Test

According to the graph, both AFDXTransmitter and AFDXReceiver applications can run under full bandwidth.

### 5.3. Jitter

Jitter is only defined for transmitting End System in the specification. ARINC 664 specification says that in transmission, the maximum allowed jitter on each VL at the output of the ES should comply with both of the following formulas:

$$\begin{cases} \text{max\_jitter} \leq 40\mu s + \frac{\sum_{i \in \{\text{set of VLs}\}} (20\text{Bytes} + L^{\text{max}}_{\text{bytes}}) \times 8 \text{ Bits/bytes}}{\text{NbW bits/s}} \\ \text{max\_jitter} \leq 500\mu s \end{cases}$$

For the given formula, maximum jitter is calculated in micro seconds, NbW is the bandwidth in bits per second, Lmax is the maximum allowable bytes for a Virtual Link and 40 micro seconds is the minimum technological jitter. In the commentary part of the specification, system integrator is advised to decide the maximum jitter under the consideration of given formulas. System integrator is free not to calculate the jitter according to first formula but select directly 500 micro seconds.

With the motivation of Airbus choice of 500 micro seconds of maximum jitter for communication with FADEC (Full Authority Digital Engine Control) of A400M, the success criteria for developed End System software for jitter will also be 500 micro seconds.

In order to examine jitter performance of the developed software, four characteristics of AFDX message will be considered; number of AFDX ports, number of Virtual Links, Lmax and BAG. For these four variables jitter performance will be examined.

Implementation of the experiments will be the same as previous implementation. AFDX related parameters will be changed, AFDXTransmitter application will be used to transmit messages and AFDXReceiver application will be used to receive and log. For each test case, each step was repeated 10 times and average of the results were used for analyze.

### 5.3.1 The Effect of Number of AFDX Ports on Jitter

Measuring the effect of the number of AFDX ports to the jitter was accomplished by keeping number of Virtual Links, Lmax and BAGs constant and changing number of AFDX ports. Throughout this experiment, Lmax was chosen as 100 bytes, and Virtual Link is configured with a BAG of 4 milliseconds.

In the first experiment, number of AFDX communication ports was selected as 5. For 5 AFDX com port, minimum jitter was measured as 0, namely delivery on time, maximum jitter was measured as 217,60 micro seconds and average jitter was calculated as 1,56 micro seconds.

Same experiment was repeated for 10 AFDX ports and maximum jitter was measured as 209,30 micro seconds and average jitter was calculated as 1,53 micro seconds.

Experiment was repeated for 15 AFDX ports and maximum jitter was measured as 261,10 micro seconds and average jitter was calculated as 1,60 micro seconds.

Fourth experiment was conducted with 20 AFDX ports and maximum jitter was measured as 274,80 micro seconds and average jitter was calculated as 1,64 micro seconds.

Next experiment was performed with 25 AFDX ports and maximum jitter was measured as 228,60 micro seconds and average jitter was calculated as 1,61 micro seconds

Finally experiment was repeated for 30 AFDX communication ports. Maximum jitter was measured as 235,40 micro seconds and average jitter was calculated as 1,60 micro seconds.

The sum of the six experiment results with graphs given in the Figure 5.2 which shows the change of the maximum jitter and the Figure 5.3 which shows the

change of the average jitter according to number of AFDX communication ports. Biggest value of the maximum jitter graph axis was chosen as 500 micro seconds which is the upper limit for acceptable jitter according to specification.

According to Figure 5.2, maximum fluctuates around 200-300 micro seconds which is in the limits of specification and according to Figure 5.3, average jitter is almost constant about 1,60 micro seconds. With the consideration of the specification, it can be said that number of AFDX communication ports does not singly affect the jitter. According to specification, any Virtual Link has authority to transmit only one AFDX port in a BAG, so sending different ports and different messages does not affect jitter.

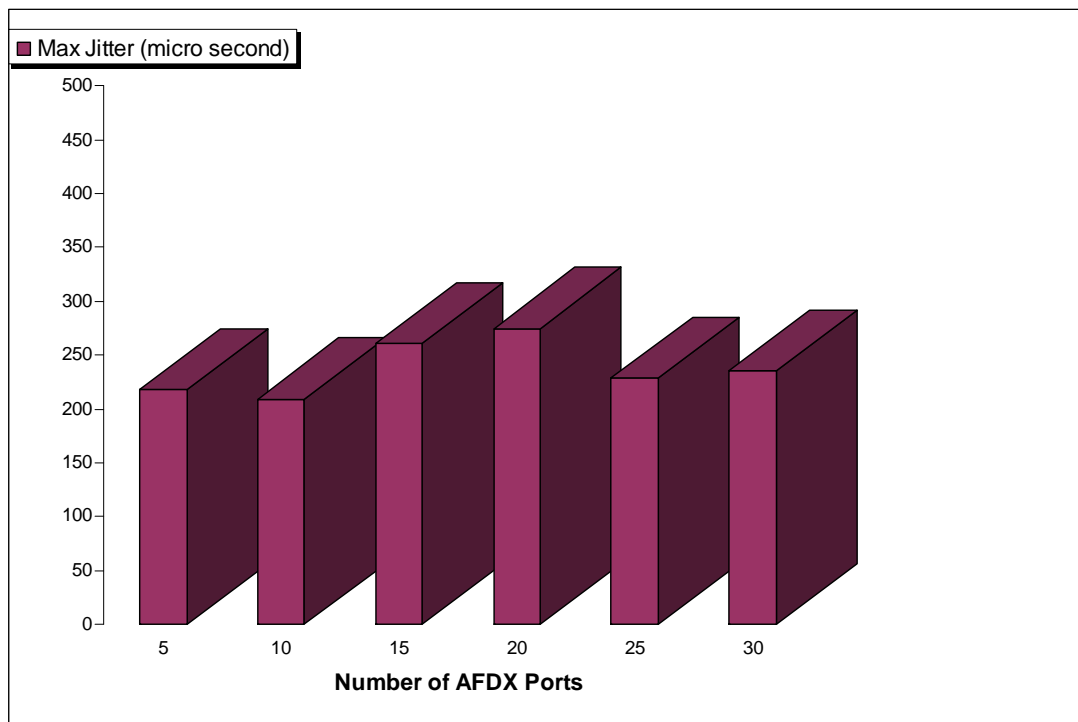


Figure 5.2 The Effect of Number of AFDX Ports on Maximum Jitter

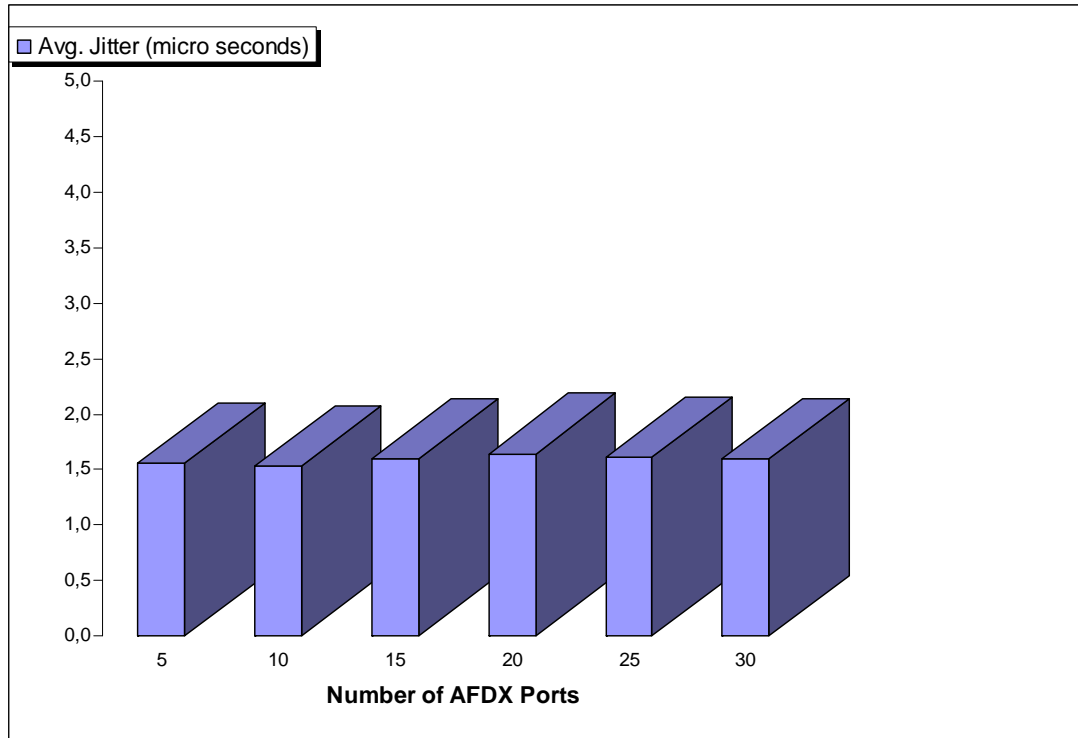


Figure 5.3 The Effect of Number of AFDX Ports on Average Jitter

With these findings, showing that number of AFDX communication ports does not affect the jitter, following experiments may be conducted without consideration of AFDX ports, one port may be used for one Virtual Link.

### 5.3.2 The Effect of Number of Virtual Links on Jitter

Measuring the effect of the number of Virtual Links to the jitter was accomplished by keeping the number of AFDX ports, Lmax and BAGs constant and changing the number of Virtual Links. Throughout this experiment, Lmax was chosen as 64 bytes and Virtual Links were configured with a BAG of 1 millisecond and only one AFDX port was used per a Virtual Link. In each experiment AFDXTransmitter application transmitted frames according to configuration file prepared for each case and AFDXReceiver application recorded a sample of 3000 frames per Virtual Link with time stamps. Each experiment was repeated 10 times and average values were considered.

In the first experiment, only one Virtual Link was configured to transmit and maximum jitter was measured as 149,40 micro seconds while average jitter was calculated as 1,77 micro seconds.

Same experiment was repeated with two Virtual Links with the same configuration; 64 bytes of Lmax, 1 millisecond of BAG and one AFDX port. For two Virtual Links, maximum jitter was measured as 461,20 micro seconds and average jitter was calculated as 1,58 micro seconds.

Configuration file of the transmitter was changed as 4 Virtual Links again for 64 bytes of Lmax, 1 millisecond of BAG and one AFDX port. Maximum jitter was measured as 139,10 and average jitter was calculated as 1,45 micro seconds respectively.

For a configuration file with 6 Virtual Links, 64 bytes of Lmax, 1 millisecond of BAG and one AFDX port, maximum jitter was measured as 396,10 micro seconds and average jitter was calculated as 1,57 micro seconds.

The same performance test was repeated for 8 Virtual Links. Maximum jitter was measured as 125,20 micro seconds and average jitter was calculated as 1,48 micro seconds.

Configuration file of the transmitter was changed as 10 Virtual Links. For this configuration maximum jitter was measured as 91,70 micro seconds and average jitter was calculated as 1,40 micro seconds.

Same experiment was repeated with 12 Virtual Links and maximum jitter was measured as 186,40 micro seconds. Average jitter was calculated as 39,93 micro seconds.

Finally, configuration file of the transmitter was changed as 14 Virtual Links again for 64 bytes of Lmax, 1 millisecond of BAG and one AFDX port. Maximum



jitter was measured as 224,90 and average jitter was calculated as 41,49 micro seconds.

The sum of the eight experiment results with two graphs are given in the Figure 5.4 showing maximum jitter and Figure 5.5 showing average jitter change with the number of Virtual Links. According to figure 5.4, maximum jitter shows differences around 90 and 470 micro seconds without a constant relation with the number of Virtual Links transmitted. Average jitter stays almost constant up to 10 Virtual Links and then dramatically increases. It can be commented that the number of Virtual Links has no effect on the jitter up to a saturation point of process and then increases jitter. Here, number 10 may not be considered as a critical Virtual Link number boundary, but BAG and Lmax should also be included in the calculations. 14 Virtual Links with 1ms of BAG and 64 bytes of frames means 72% of bandwidth usage and 14000 transmission process per second. Increasing the number of Virtual Links increases both bandwidth usage and process which results an increase in the average jitter after a point, 12 for 64 bytes of Lmax and 1 ms of BAG.

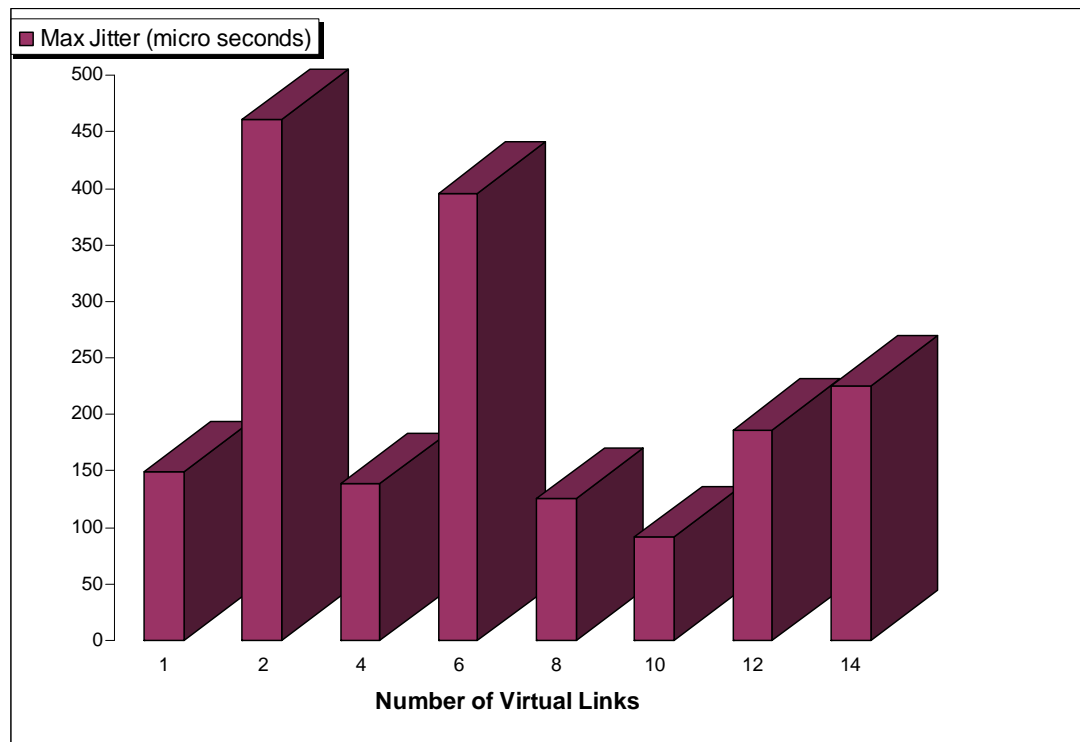


Figure 5.4 The Effect of Number of Virtual Links on Maximum Jitter

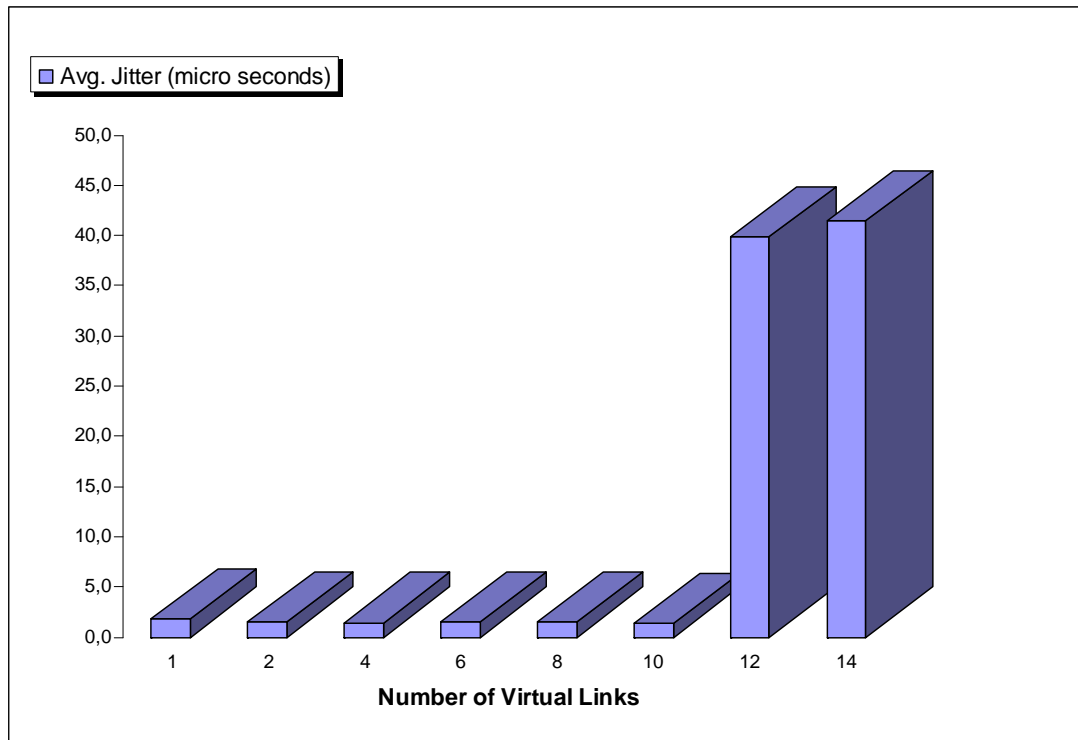


Figure 5.5 The Effect of Number of Virtual Links on Average Jitter

### 5.3.3 The Effect of Lmax on Jitter

Measuring the effect of Lmax to the jitter was accomplished by keeping number of AFDX ports, number of Virtual Links and BAG constant and changing the Lmax value. Throughout this experiment, one Virtual Links with single AFDX communication port was configured with a BAG of 16 milliseconds. Lmax value was changed and results were recorded. In each experiment AFDXTransmitter application transmitted frames according to configuration file prepared for each case and AFDXReceiver application recorded a sample of 3000 frames per Virtual Link with time stamps. Each experiment was repeated 10 times and average values were considered.

In the first experiment, Lmax was configured as 100 bytes of frames and maximum jitter was measured as 103,10 micro seconds. Average jitter was calculated as 2,00 micro seconds.

In the second experiment, Lmax was configured as 300. Maximum and average jitters were found as 116,00 and 2,04 micro seconds respectively.

The third experiment was conducted with Lmax equals to 600 bytes and maximum jitter was measured as 97,40. Average jitter was calculated as 1,89 micro seconds.

The same experiment was repeated with Lmax equals to 900 bytes. Maximum and average jitters were obtained as 99,30 and 1,81 micro seconds respectively.

The fifth experiment was performed with Lmax equals to 1200 bytes and maximum jitter was measured as 102,20 micro seconds. Average jitter was calculated as 1,80 micro seconds.

Finally Lmax was selected as 1500 bytes. Maximum and average jitters were obtained as 112,90 and 1,90 micro seconds respectively.

The sum of the six experiment results with two graphs are given in the Figure 5.6 showing maximum jitter and the Figure 5.7 showing average jitter change by to Lmax. According to figure 5.6, maximum jitter fluctuates around 100 micro seconds. Average jitter also fluctuates around 2 micro seconds and there is no indication that the jitter is affected by Lmax for this application.

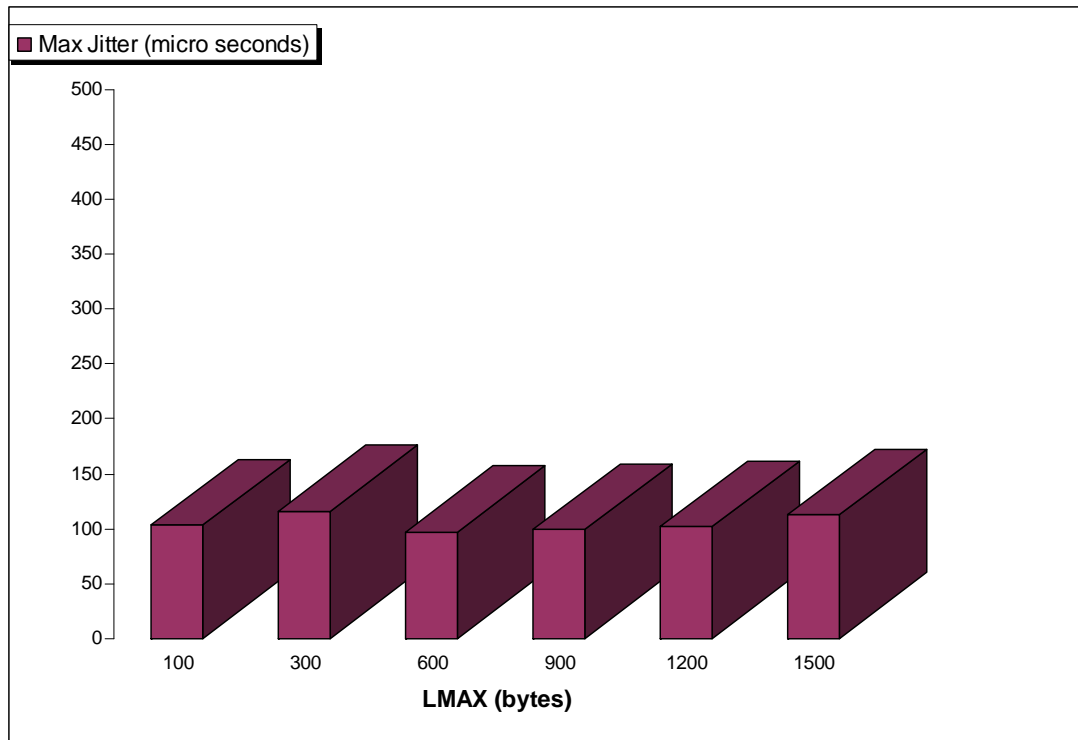


Figure 5.6 The Effect of Lmax on Maximum Jitter

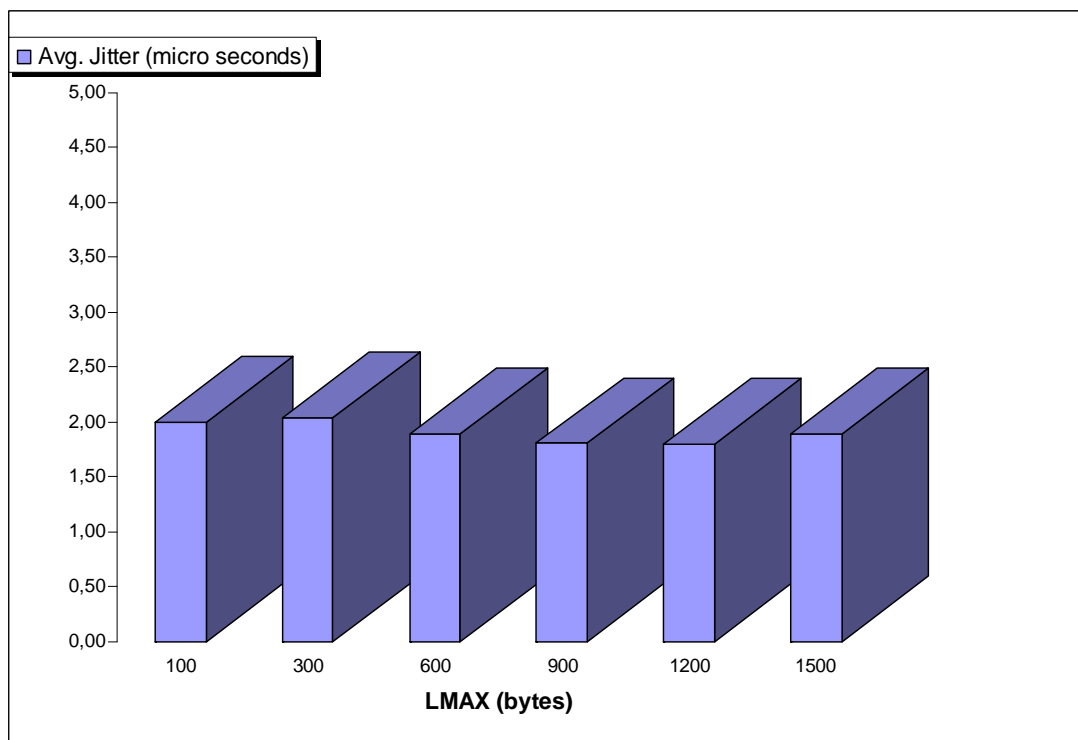


Figure 5.7 The Effect of Lmax on Average Jitter

#### **5.3.4 The Effect of BAG on Jitter**

The effects of all possible BAGs; 1, 2, 4, 8, 16, 32, 64 and 128 were measured by keeping number of AFDX ports, number of Virtual Links and Lmax constant and changing BAG in eight different experiments. Throughout this experiment, one Virtual Link with single AFDX communication port was configured with an Lmax of 500 bytes. BAG was changed and results were recorded. In each experiment AFDXTransmitter application transmitted frames according to configuration file prepared for each case and AFDXReceiver application recorded a sample of 3000 frames per Virtual Link with time stamps. Each experiment was repeated 10 times and average values were considered.

The first experiment was performed with a BAG equals to 1 millisecond. Maximum jitter was measured as 149,40 and average jitter was calculated as 1,77 micro seconds.

The second experiment was performed with a BAG equals to 2 milliseconds. Maximum jitter was measured as 416,40 and average jitter was calculated as 1,99 micro seconds.

The third experiment was performed with a BAG equals to 4 milliseconds. Maximum jitter was measured as 118,20 and average jitter was calculated as 1,59 micro seconds.

Next experiment was performed with a BAG equals to 8. Maximum jitter was measured as 256,90 and average jitter was calculated as 2,08 micro seconds.

The fifth experiment was performed with a BAG equals to 16 milliseconds. Maximum jitter was measured as 99,60 and average jitter was calculated as 2,17 micro seconds.

Next experiment was performed with a BAG equals to 32 milliseconds. Maximum jitter was measured as 268,20 and average jitter was calculated as 3,51 micro seconds.

Same experiment was repeated with a BAG of 64 milliseconds. Maximum jitter was measured as 125,50 and average jitter was calculated as 3,72 micro seconds.

The last experiment was performed with the maximum BAG that is defined in the specification; 128 milliseconds. Maximum jitter was measured as 102,90 and average jitter was calculated as 6,75 micro seconds.

The sum of the eight experiment results with two graphs given in the Figure 5.8 showing the maximum jitter and Figure 5.9 showing the average jitter change by BAG. According to the Figure 5.8, maximum jitter value fluctuates around 100 and 420 microseconds without a constant relation with the BAG. Figure 5.9 indicates that average jitter has an increasing trend with increasing BAG. The reason for this trend can be explained with the number of context switching. The application works with 1 millisecond of steps to count each type of BAG. When the BAG increases, number of steps also increases and possibility of interruptions to the execution of the application increases. That is why the average jitter increases with the increase of BAG. On the other hand maximum jitter is not related with the BAG.

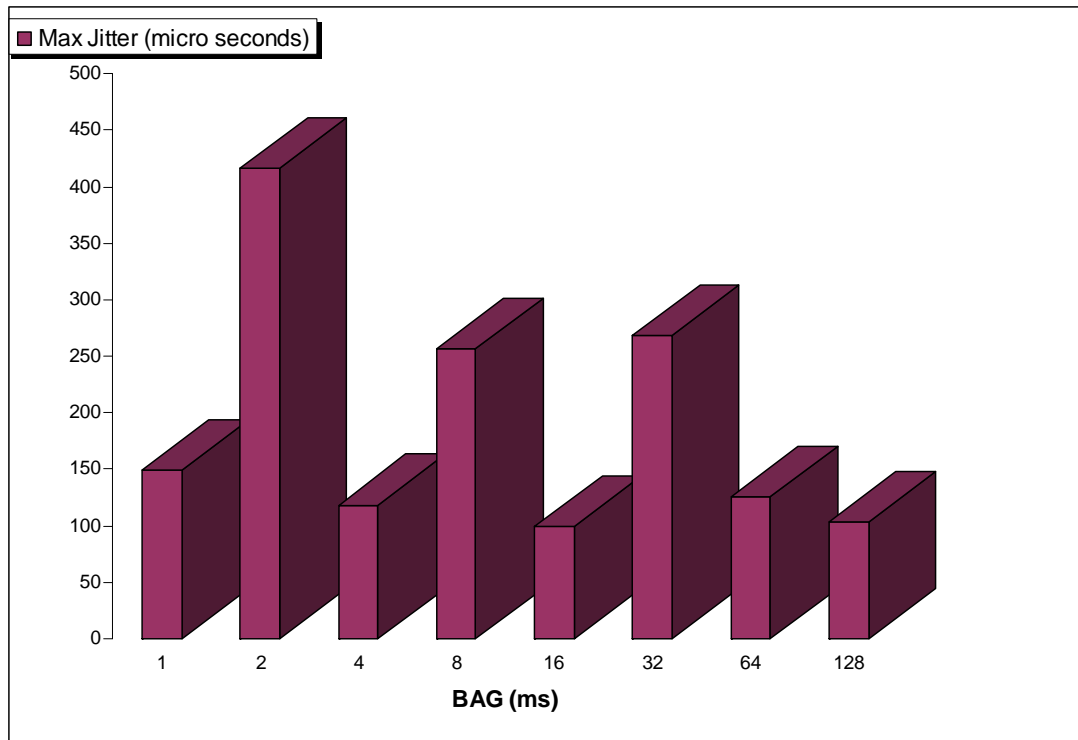


Figure 5.8 The Effect of BAG on Maximum Jitter

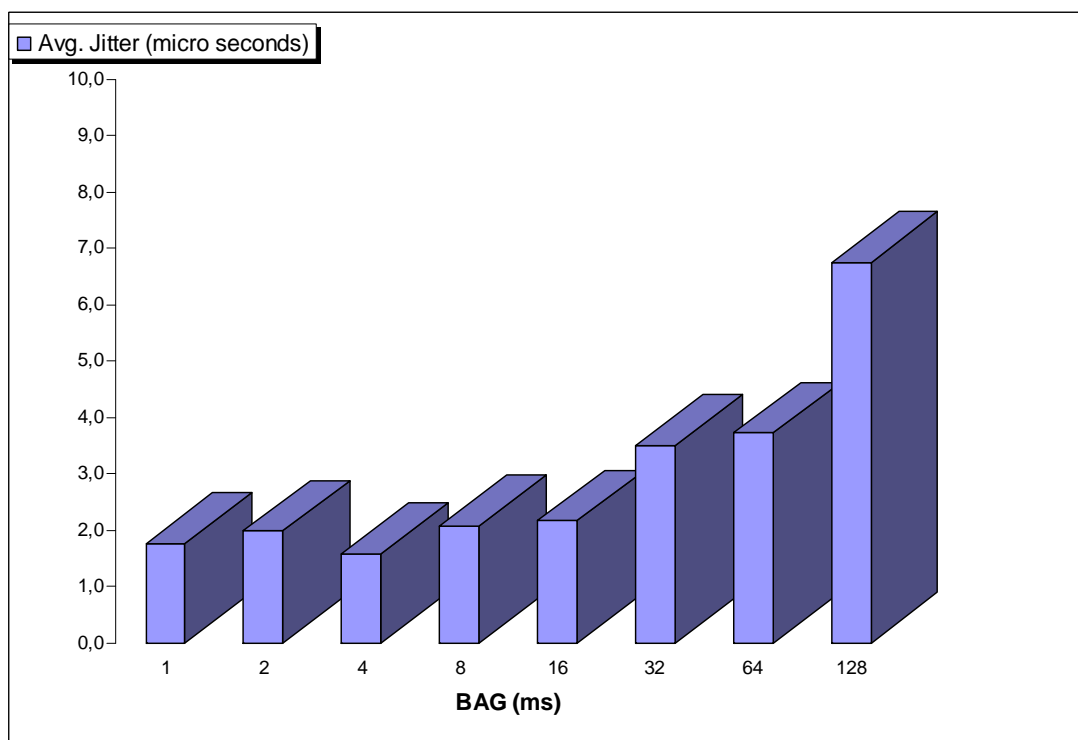


Figure 5.9 The Effect of BAG on Average Jitter

## **CHAPTER 6**

### **CONCLUSION**

ARINC 664/AFDX (Avionics Full Duplex Switched Ethernet) protocol is a leading onboard communication technology in civil aviation. The key features of AFDX are high transmission rate, redundancy, guaranteed bandwidth for data and determinism. Nowadays many commercial companies develop AFDX network components; switches and End Systems. End Systems use standard Ethernet infrastructure with improved characteristics.

The main contribution of this thesis is developing an AFDX End System with a standard PC with COTS Ethernet card. An object oriented software package with simple API was prepared for the developers to prepare small applications. The aim of the applications prepared with this software package is to enable the developer to test his real AFDX application during development phase and to enable test engineers to conduct ground test of AFDX networks. The improved characteristics of AFDX were implemented in the software that runs under non real time Windows XP operating system. The software was tested for performance requirements defined in the specification document [16].

An AFDX End System has only one redundant interface with the AFDX network. The entire generated network for a specific End System comes from and all the traffic generated by this End System is goes through this single redundant



interface. That is why throughout all performance tests, the unit under test is tested with a single computer simulating the switch interface.

Performance analysis shows that AFDX application running in a non realtime environment with standard PC and Ethernet card suffices latency requirements of ARINC 664 Specifications for both the receiver and the transmitter.

Also MAC constraints requirements of the specification are met by both the transmitter and the receiver applications. Test results show that increasing the bandwidth also affects jitter adversely but in the limitation of requirements.

The jitter requirement of the specification was tested against number of AFDX ports used, number of Virtual Links, Lmax and BAG.

The specification limits any Virtual Link to transmit only one AFDX port message in a BAG, hence changing number of AFDX ports does not have any effect on messaging physically, but only changes the message itself. Tests also proved that number of AFDX ports does not have any role in performance characteristics.

Increasing the number of Virtual Links not only increases the processes performed by the software but also bandwidth usage. As the CPU used for this application was selected more than sufficient, basically no effects of number of Virtual Links to jitter were observed up to a level but after a critical number of Virtual Links average jitter was observed as dramatically increasing within the boundaries of specification.

No changes observed for increasing Lmax but it was observed that increasing the value of BAG increases jitter in limits of specification.

For each test case, maximum jitter was not relatively changed with the parameter under test but showed undeterministic distribution within the specification boundary which is 500 micro seconds. It can be commented that maximum jitter is

not related with any parameter but the internal processes of the operating system itself.

All these tests were performed with a “clear” operating system that no application running besides the application under test. No antivirus programs were installed to the test environment and firewall was disabled. Many of windows services were disabled too. It was observed that windows services, especially the ones that are related with networking limit the performance of the application and result in unexpected interruptions on the transmission which shades determinism and performance characteristics.

The research, code development and testing of the codes show that it is possible to implement a software based AFDX End System with standard tools, standard equipment and it is also possible to run this software in a non realtime operating system with limitations.

Nevertheless non certifiable design method and unpredictable behavior of non realtime operating system restrict the use of such a system only for ground testing and code development phases but not in a real avionics system integrated on an aircraft. This stack is very cost and development time effective for a data acquisition system. Also realtime application developers can use this software during development and functional tests of their new software. This could shorten the development schedule of a software project.

Future research may be useful to improve the performance and functionalities of this software stack. Performance improvement may help the host computer also to perform other processes than AFDX End System. Other port types like SAP port and fragmentation feature may be included to the software. Special interest should be shown to control the unexpected behaviors of the operating system.

Also the stack may be ported to other operating systems and platforms for more flexibility. For instance, Linux or RTLinux could be good candidates to port this software. As Ethernet communication libraries are portable, which have the same API for DLL; it would be easy to port this part of the software. On the other hand timing and scheduling parts of the code should be changed according to timers for Linux and RTLinux. It would be much more successful to port the software to RTLinux with realtime libraries.

## REFERENCES

- [1] Collinson, R. P. G., "Introduction to Avionics Systems", Springer, 2003
- [2] Faulconbridge, R.I., "Avionics Principles", Argos Press, January 2007
- [3] "Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations", DO-297, August 2008
- [4] Ott, A., "System Testing in the Avionics Domain", Phd Thesis, Vorgelegt im Fachbereich 3 (Mathematik & Informatik) der Universitat Bremen, October 2007
- [5] Zalewski, J., Trawczy, D., Sosnowski, J., Kornecki, A., Sniezek, M. "Safety Issues in Avionics and Automotive Databuses", IFAC World Congress, July 2005
- [6] "ARINC 429 Protocol Tutorial (1500-029)", Condor Engineering, Rev. 1.07, July 2004
- [7] "ARINC Specification Tutorial", AIM GmbH, Rev 1.1, July 2001
- [8] "ARINC Specification 429, Part1-16", September 2001
- [9] "MIL-HDBK-1553A Notice 2", September 1978
- [10] "MIL-STD-1553B", November 1988
- [11] Schuster, T., Verma, D., "Networking Concepts Comparison for Avionics Architecture", 27<sup>th</sup> Digital Avionics Systems Conference, Pages: 1.D.1-1 – 1.D.1-11, October 2008
- [12] "AFDX/ARINC 664 Tutorial", Techsat, August 2008
- [13] Pickles, B., "Avionics Full Duplex Switched Ethernet (AFDX)", SBS Technologies, May 2006
- [14] Tanenbaum, A.S., "Computer Networks", Fourth Edition, Prentice Hall, 2003

- [15] "Developing AFDX Solutions Application Note AC221", Actel Corporation, March 2005
- [16] "Draft 3 of Project Paper 664 Aircraft Data Network Part 7 Avionics Full Duplex Switched Ethernet (AFDX) Network", September 2004
- [17] "Draft 3 of Supplement 2 to ARINC Specification 664 Part 4 Internet Based Address Structures and Assigned Numbers", October 2007
- [18] <http://www.windriver.com>, last visited 13/10/2009
- [19] <http://www.ghs.com>, last visited 13/10/2009
- [20] Kessler, G.C., "On Teaching TCP/IP Protocol Analysis to Computer Forensics Examiners", *Journal of Digital Forensic Practice*, 2(1), Pages: 43-53, March 2008
- [21] Risso, F., Degioanni, L. "An Architecture for High Performance Network Analysis", *Proceeding of the 6<sup>th</sup> IEEE Symposium on Computers and Communications*, July 2001
- [22] <http://www.winpcap.org>, last visited 21/09/2009
- [23] Degioanni, L., Baldi, M., Risso, F., Varenni, G. "Profiling and Optimization of Software-Based Network-Analysis Applications", *Proceeding of the 15<sup>th</sup> IEEE Symposium on Computer Architecture and High Performance Computing*, Page: 226-234, November 2003
- [24] Yüksel, E., Örencik, B. "Sanal Özel Ağ Tasarımı ve Gerçeklemesi", *Ağ ve Bilgi Güvenliği Ulusal Semp. (ABG 2005) Bildiriler Kitabı*, Pages: 114-118, June 2005
- [25] Zhao X., Li X. "Methods of Capturing Network Packets", *Application of Computers*, Vol.21 Issue 8, Pages: 242-243, 2004
- [26] Chen C., Wu S. "Design and Implementation of Message Monitor Based on WinPcap", *Science Technology and Engineering*, Vol.8 Issue 5, Pages: 1203-1207, 2008
- [27] Ballard, D. "Network Monitor", Ms Thesis, Rochester Institute of Technology B. Thomas Golsiano Collage of Computing and Information Sciences, February 2004
- [28] Xiao Y., Li, L., Wen, J. "Network Program Architecture Based Winpcap and Sock", *Ordinance Industry Automation*, Vol.24 Issue 5, Pages: 49-50, 2005

## APPENDIX A

### AFDXAPI.H FILE OF AFDX TRANSMITTER

```
/* *****
*
*      Class CAFDXTransmitterAPI (.h)
*
*      Project      : METU EEE, AFDX network monitor application with
*                    standard PC and ethernet card
*
*      Author       : Emre ERDINC
*      Date        : 04.08.2009
*
*      Description  : Main class to be included by tha application
*                    program
*
*      Modification History:
*
* *****
*/

/* *****
#pragma once
#include "VirtualLink\VirtualLink.cpp"
#include "NICInterface\NICInterface.cpp"
#include <windows.h>
#include "process.h"
#include <iostream>
#include <fstream>
/* *****

struct MsgReady
{
    int dVLCOUNT;      // number of VL's in the slot
    int *dVLNo;
}MSGREADY;

/* *****

class CAFDXTransmitterAPI
{
public:
    CAFDXTransmitterAPI(void);
    ~CAFDXTransmitterAPI(void);
    static CAFDXTransmitterAPI * GetInstance();
    int  Configure(char* sConfigFilePath);
```

```

    int Start();
    int WriteMessage(int dVLIndex, int dPortIndex, char* sMsg);

private:

    int m_dNoOfVLs;    // number of created virtual links
    CVirtualLink * m_xVirtualLinks;    // virtual link pointer

    CNICInterface m_xNICInterface;

    MsgReady m_pxTimeSlot[128];
    int ArrangeSlot(int dVLNo, int dBAG);

    static unsigned __stdcall TransmitThread(void* arg);
};

```

## APPENDIX B

### AFDXAPI.H FILE OF AFDX RECEIVER

```
/*
 *
 * Class CAFDXReceiverAPI (.h)
 *
 * Project      : METU EEE, AFDX network monitor application with
 *               standard PC and ethernet card
 *
 * Author       : Emre ERDINC
 * Date        : 04.08.2009
 *
 * Description  : Main class to be included by the application
 *               program
 *
 * Modification History:
 *
 */
/*****

/*****
#pragma once
#include "VirtualLink\VirtualLink.cpp"
#include <iostream>
#include <fstream>
#include <windows.h>
#include "process.h"
#include <time.h>

/*****

class CAFDXReceiverAPI
{
public:
    static CAFDXReceiverAPI * GetInstance();

    int  Configure(char* sConfigFilePath);
    int  Start();
    int  Receive(char *cInterface,unsigned short *dVLId, char
*sMsg,int *dMsgLen,unsigned char*ucSeqNo,unsigned short*
usUDPSrcPort,unsigned short* usUDPDstPort,struct timeval *ts);

    HANDLE m_threadA;
    HANDLE m_threadB;

private:
```



```

CAFDXReceiverAPI(void);
~CAFDXReceiverAPI(void);

int DeviceConfiguration(      char * sInterfaceNameA,
                              char * sInterfaceNameB);

int m_dNoOfVLs;    // number of created virtual links
CVirtualLink * m_xVirtualLinks;// virtual link pointer

char m_sInterfaceNameA[100]; // pcap interface device name
for interface A
char m_sInterfaceNameB[100]; // pcap interface device name
for interface B

char m_sErrbufA[PCAP_ERRBUF_SIZE]; // pcap error buffer for
network A
char m_sErrbufB[PCAP_ERRBUF_SIZE]; // pcap error buffer for
network B
pcap_t *m_xAdHandleA;    // pcap device handle for network A
pcap_t *m_xAdHandleB;    // pcap device handle for network B

unsigned char m_sAFDXFrameA[1500];// whole frame for network A
unsigned char m_sAFDXFrameB[1500];// whole frame for network B

bool m_bFreshA; // true when there is a message ready in NW A
bool m_bFreshB; // true when there is a message ready in NW B

static unsigned __stdcall ReceiveThreadA(void* arg);
static unsigned __stdcall ReceiveThreadB(void* arg);

// return parameters for A
unsigned short m_dVLOrderA;
int m_dMsgLengthA;
unsigned char m_ucSeqNoA;
unsigned short m_usUDPSrcPortA;
unsigned short m_usUDPDstPortA;
struct timeval m_tsA;

// return parameters for B
unsigned short m_dVLOrderB;
int m_dMsgLengthB;
unsigned char m_ucSeqNoB;
unsigned short m_usUDPSrcPortB;
unsigned short m_usUDPDstPortB;
struct timeval m_tsB;
};

struct ArgForThread
{
    CAFDXReceiverAPI* xThis;    // this
    int dNwIndex;    // Network Index 0:A, 1:B
}ARGFORTHREAD;

```