ARINC664 / AFDX

C/C++ based Application Programming Interface

Avionics Databus Solutions

Programmer‘s Guide

V19.0.x Rev. A
April 2017

www.aim-online.com

ARINC664 / AFDX

C/C++ based Application Programming Interface

Programmer's Guide

V19.0.x Rev. A
April 2017

AIM NO.
60-15900-37-19.0.X

AIM – Gesellschaft für angewandte Informatik und Mikroelektronik mbH

AIM GmbH
Sasbacher Str. 2
D-79111 Freiburg / Germany
Phone +49 (0)761 4 52 29-0
Fax +49 (0)761 4 52 29-33
sales@aim-online.com

AIM GmbH – Munich Sales Office
Terofalstr. 23a
D-80689 München / Germany
Phone +49 (0)89 70 92 92-92
Fax +49 (0)89 70 92 92-94
salesgermany@aim-online.com

AIM UK Office
Cressex Enterprise Centre, Lincoln Rd.
High Wycombe, Bucks. HP12 3RB / UK
Phone +44 (0)1494-446844
Fax +44 (0)1494-449324
salesuk@aim-online.com

AIM USA LLC
Seven Neshaminy Interplex
Suite 211 Trevose, PA 19053
Phone 267-982-2600
Fax 215-645-1580
salesusa@aim-online.com

DOCUMENT HISTORY

The following table defines the history of this document. The description of changes/enhancements made to each version is defined in general terms.

Version	Cover Date	Created by	Description
V10.0x	June 2004	J. Furgerson	Creation of document.
V12.0xA	January 2005	J. Furgerson	Updated to support V12.0 BSP – See API/ACI/AMC-FDX Application Interface Library Reference Manual V12.0x for a list of changes to the Application Programming Interface
V13.0xA	November 2005	J. Furgerson	Updated to support V13.0 BSP – See Reference Manual AyI-FDX and AMC-FDX Application Interface Library V13.0x for a list of changes to the Application Programming Interface
5.3x Revision B	20.06.2005	J. Furgerson	See Appendix A for details
5.3x Revision C	July 2007	S. Riedinger	Unify names of products
5.3x Revision D	August 2007	S. Riedinger	Update images
15.0x Revision A	October 2007	R. Heitzmann	Update to BSP V15.00
15.0x Revision B	December 2008	M. Riedinger	Bugfix in Description
16.0x Revision A	July 2010	M. Riedinger	Update to BSP V16.00
16.0x Revision B	August 2010	R. Heitzmann	Update to BSP V16.00 VL-Limitation for Endsystem deleted Address vor AIM-USA corrected
16.0x Revision C	September 2010	M. Riedinger	Update to BSP V16.00, bugfix in header line
17.1x Revision A	December 2011	M. Riedinger	Update to BSP V17.00
19.0.x Revision A	April 2017	R. Heitzmann	Update to BSP 19.0.0

THIS PAGE IS INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

Section **Title**
Page

LIST OF FIGURES

LIST OF TABLES

Table **Title** **Page**

1 INTRODUCTION

1.1 General

Welcome to the **Programmer's Guide AFDX/ ARINC-664**. This programmer's guide, in conjunction with the **Reference Manual AFDX/ ARINC-664**, is intended to provide the software (s/w) programmer with the information needed to develop a host computer application interface to the API-FDX-2, AMC-FDX-2, APM-FDX-2, ACC-FDX-3U/6U-2/4, APU-FDX-2, APE-FDX-2 and the APX-GNET-2/4 Bus Interface Cards and the fdXTap™ including high level PCI-FDX system design information, board support package (BSP) contents, user application system design concepts, function call guidelines and sample programs. The **Reference Manual AFDX/ ARINC-664** provides the detailed API s/w library functions.

1.2 How This Programmer's Guide is Organized

The **Programmer's Guide AFDX/ ARINC-664** is divided into 6 sections. These sections include the following:

Section 1
Introduction

Provides an introduction to the contents of the programmer's guide documentation conventions and applicable documents.

Section 2
AFDX
Network
Overview

Provides a high level overview of the Avionics Full Duplex Switched Ethernet (AFDX) Network structure, protocols and frame formats.

Section 3
PCI-FDX
Overview

Provides a high level overview of the hardware and software design. Included in the software section is information concerning the compilers supported, a description of the Board Support Package and how to create an application program.

Section 4
Programming Using the API Library

Provides the programming guidelines for the Library Administration and Board-level functions as well as the two main functional systems on the AFDX devices including:

- Transmitter

- Receiver

4.1
Library Admin & System

4.2
Transmitter Programming

4.3
Receiver Programming

Provides an explanation of two complete sample programs, and references for function calls used in the sample programs.

Section 5
Program Samples

Provides expansion for all acronyms and definitions for terms used frequently in this document.

Section 6
Notes

1.3 Conventions Used

1.3.1 General Documentation Conventions

We use a number of different styles of text and layout in this document to help differentiate between the different kinds of information. Here are some examples of the styles we use and an explanation of what they mean:

Italics - used as a placeholder for the actual name, filename, or version of the software in use

Bold text - a function, or parameter, or used to highlight important information

Bold Blue - will be used to show reference documentation

Bold italics - caution, warning or note

`Font` - font used to show paths, directories and filenames within the body of text will be shown in blue. For example:

`C:\Windows\System32\Drivers\Aim_fdx.sys`

```
A smaller version of this font will be used to list
software code.
```

| - an action delineator that will lead you through nested menu items and dialog box options to a final action, for example, the **File | Open** ..

In addition to text and layout convention, there are a couple of naming conventions used to simplify the information herein. The PCI-FDX s/w library, is also called the Application Programming Interface (API). For ease of documentation flow, the PCI -FDX s/w library will be referred to from this point on as the **API S/W Library**. In addition, the software and firmware contained on the PCI-FDX bus interface board will be referred to as the **API Target S/W**.

1.3.2 Parameter Naming Conventions

In order to understand the sample programs and individual programming examples contained in this guide, we should review some of the parameter naming conventions used throughout the API S/W Library. Naming conventions have been used for naming constants, structures, functions calls and data types.

> ***Note:*** ***All constants, structures and functions used in the API S/W Library are defined in the*** `AiFdx_def.h` ***header file. Data types used in the API S/W Library are defined in*** `Ai_cdef.h`**.**

Naming conventions used include the following

- **Constants** - For every function call, a list of constants have been defined to better describe the numerical value of the function input or output. (located in `AiFdx_def.h`). These constants will be used throughout this document.
- **Structures** - Named as **ty_fdx_*name*** where *name* is unique to the structure. (located in AiFdx_def.h)
- **Functions** - Named as either **Fdx*name*** or **FdxCmd*name*** where *name* is unique to the function (located in AiFdx_def.h)
  - **Fdx*name*** functions do not involve driver commands to the bus interface unit (BIU)
  - **FdxCmd*name*** functions involve driver commands to the BIU
- **Data Types** - all variables are assigned an AIM equated data type as shown in Table 1-1 below (defined in Ai_cdef.h)

Table 1-1 API S/W Library Data Type Naming Conventions

API S/W Library	Data Type	Size (in bytes)
AiInt	integer	4
AiUInt	unsigned integer	4
AiInt8	character	1
AiInt16	short integer	2
AiInt32	long integer	4
AiUInt32	unsigned long integer	4
AiUnt16	unsigned short integer	2
AiUInt8	unsigned character	1
AiChar	character	1
AiUChar	unsigned character	1
AiDouble	double floating point	8
AiFloat	single floating point	4

1.4 AIM Document Family

AIM has developed several documents that may be used to aid the developer with other aspects involving the use of the PCI-FDX bus interface card. These documents and a summary of their contents are listed below:

Reference Manual AFDX/ ARINC-664 - provides the AFDX application developer with the detailed PCI-FDX library function calls. This guide is to be used in conjunction with the Programmer's Guide AFDX/ ARINC-664.

Getting Started Manual AFDX/ ARINC-664 - assists first time users of the AIM API-FDX, and AMC-FDX boards, and the fdXTap with software installation, hardware setup and starting a sample project.

API-FDX Hardware Manual - provides the hardware user’s manual for the API-FDX PCI-Bus modules. The document covers the hardware installation, the board connections, the technical data and a general description of the hardware architecture.

AMC-FDX-2 Hardware Manual - provides the hardware user’s manual for the AMC-FDX-2 PCI Mezzanine Card modules. The document covers the hardware installation, the board connections, the technical data and a general description of the hardware architecture.

fdXTap™ Hardware Manual - provides the hardware user’s manual for the fdXTap™ Network Tap device. The document covers the hardware installation, the board connections, the technical data and a general description of the fdXTap™ hardware architecture.

APX-GNET-2/4 Hardware Manual - provides the hardware user’s manual for the APX-GNET-2-4 PCI(-X) Card. The document covers the hardware installation, the board connections, the technical data and a general description of the hardware architecture.

APU-FDX-2 Hardware Manual - provides the hardware user’s manual for the APU-FDX-2 USB 2.0 Device. The document covers the hardware installation, the board connections, the technical data and a general description of the hardware architecture.

PBA.pro Bus Analyzer Getting Started – introduces the PBA.pro Bus Analyzer and contains links to further documentation.

AIM Network Server (ANS) Users Manual - assists users with installation and initial setup of the AIM Network Server software. Client and Server configuration and software/hardware requirements are outlined with complete step-by-step instructions for software installation.

THIS PAGE INTENTIONALLY LEFT BLANK

2 AFDX NETWORK OVERVIEW

The Avionics Full-Duplex Switched Ethernet (AFDX) Network is built around commercial standards including: IEEE802.3 Ethernet Medium Access Controller (MAC) addressing, Internet Protocol (IP) and User Datagram Protocol (UDP). Provisions have been added to ensure guaranteed deterministic timing and redundancy required for Avionics applications. The network data rates include 10Mbps, 100Mbps and 1000 Mbps for GNET.

This section will provide an overview of the following:

a. AFDX Network Structure

b. AFDX Protocol Stack

c. AFDX Frame Format.

Detailed information regarding the AFDX End System requirements can be found in the AFDX End System Detailed Functional Specification.

2.1 AFDX Network Structure

As shown in Figure 2-1, there are three types of AFDX Network elements including:

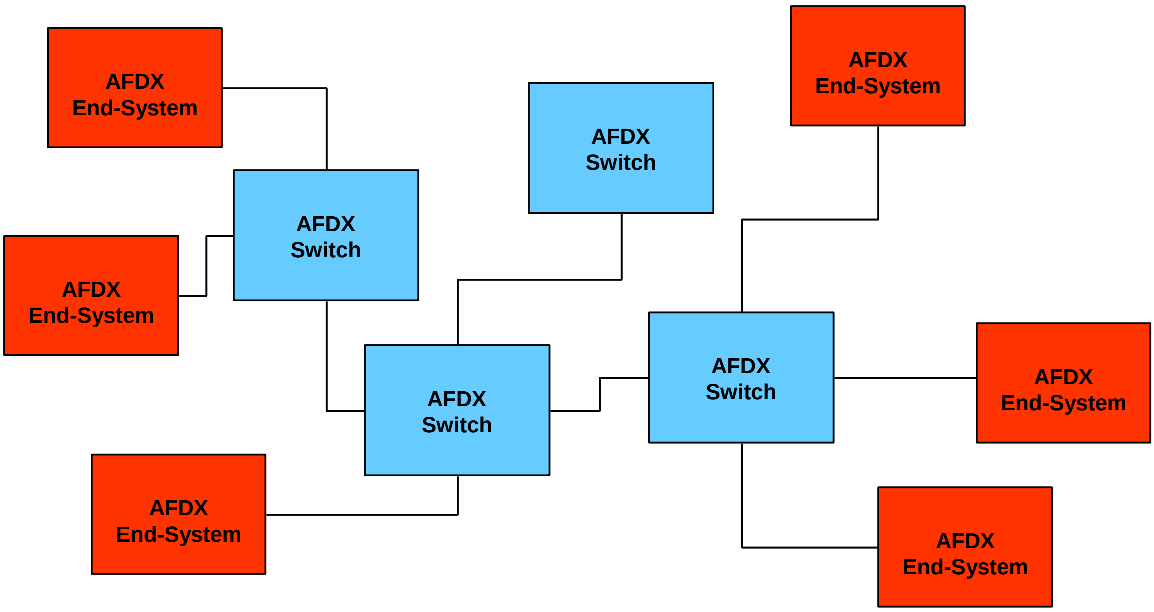
a. **End System(s)** - A device whose applications access the network components to send or receive data from the network. End-Systems perform traffic shaping which is enforced by Switches.

b. **Switch(es)** - A device which performs traffic policing and filtering, and forwards packets towards their destination End-Systems.

c. **Link(s)** - All links/connections are copper or fiber optic, full duplex, 100Mbits/sec (no dedicated backbone bus for Inter-switch communications).

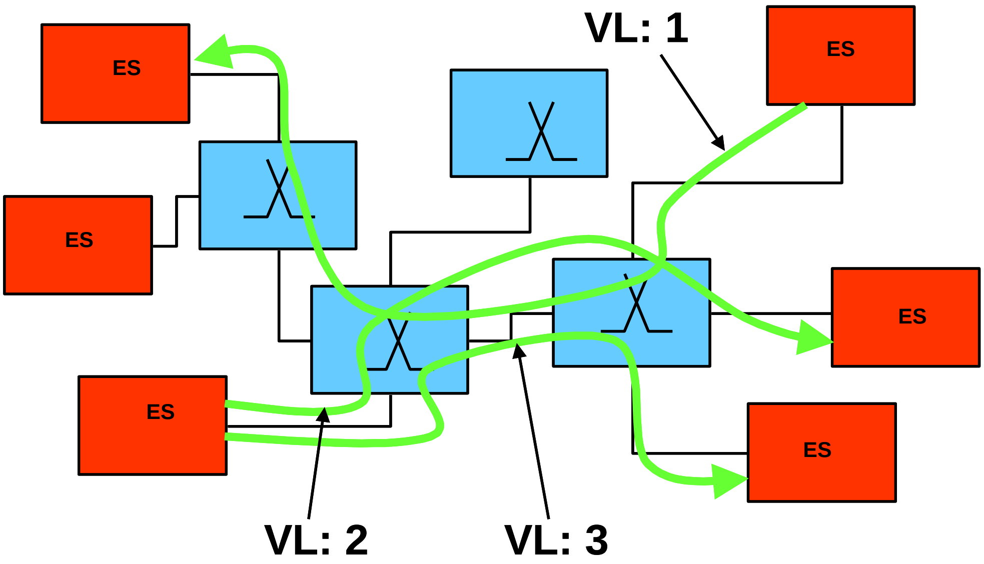
Redundancy is achieved by duplication of the connections (wires) and the Switches.

Figure 2-1 AFDX Network Topology



End-Systems communicate/exchange Frames through **Virtual Links (VLs)** as depicted in Figure 2-1. A **VL** defines a unidirectional connection from one source End-System to one or more destination End-Systems.

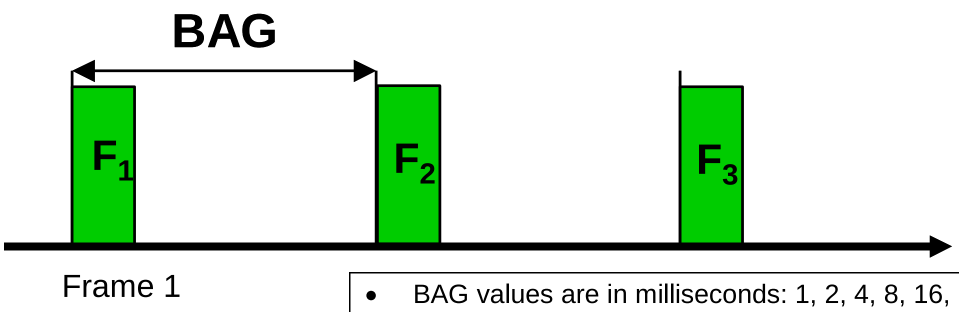
Figure 2-2 Virtual Link Scenario



- An AFDX Network can contain up to 64K VLs

End-Systems perform traffic shaping and Integrity checking on each VL. The End-System controls the flow for each VL in accordance with the **Bandwidth Allocation Gap (BAG)** which is depicted in Figure 2-3.

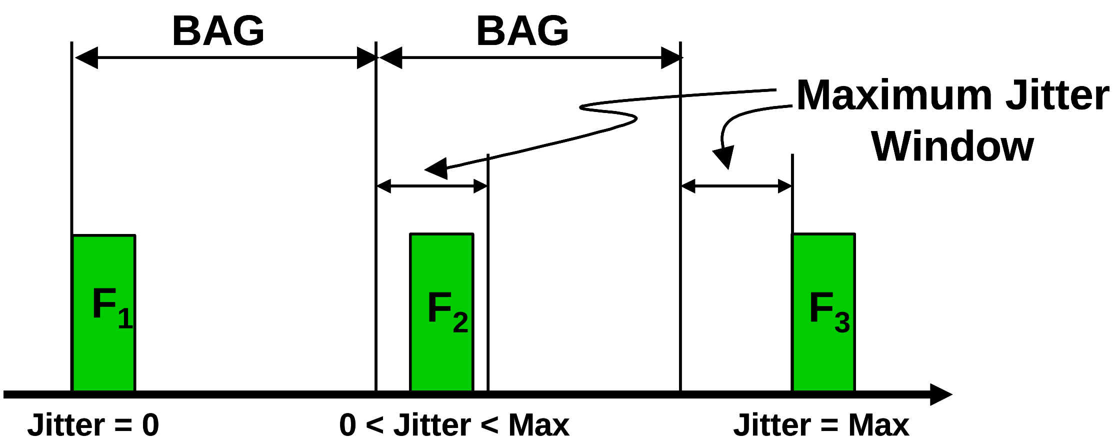
Figure 2-3 Bandwidth Allocation Gap (BAG)



- BAG values are in milliseconds: 1, 2, 4, 8, 16, 32, 64, 128
- Per VL, a maximum of 1000 Frames per second can be transmitted.

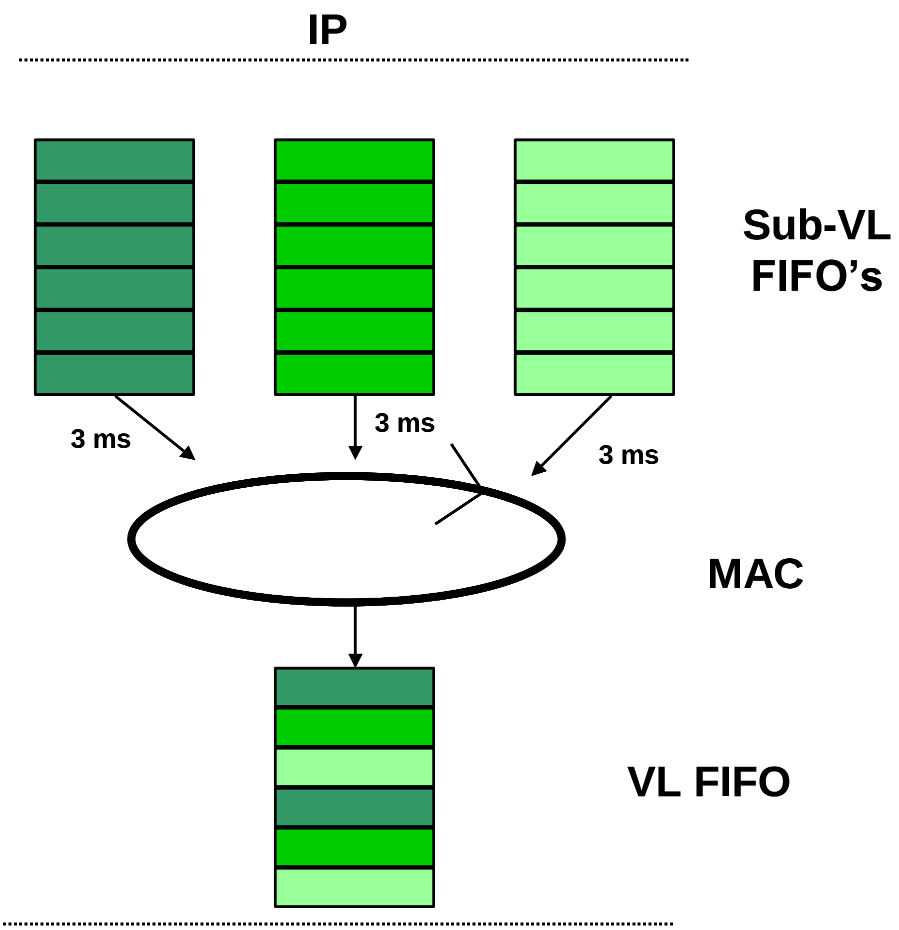
For a VL, frames can appear on the link in a given time interval (Window) which is sized by the BAG and the maximum allowed jitter as shown in Figure 2-4. **Jitter** is the difference between the minimum and maximum time from when a source node sends a message to when the sink node receives the message. Jitter is generally a function of the network design and multiplexing multiple VLs on one port.

Figure 2-4 Maximum Jitter Window



Each VL may consist of up to 4 sub-VLs. Each Sub VL is designated its own FIFO queue. Scheduling of frames is based upon a Round-Robin transmission scheme as shown in Figure 2-5.

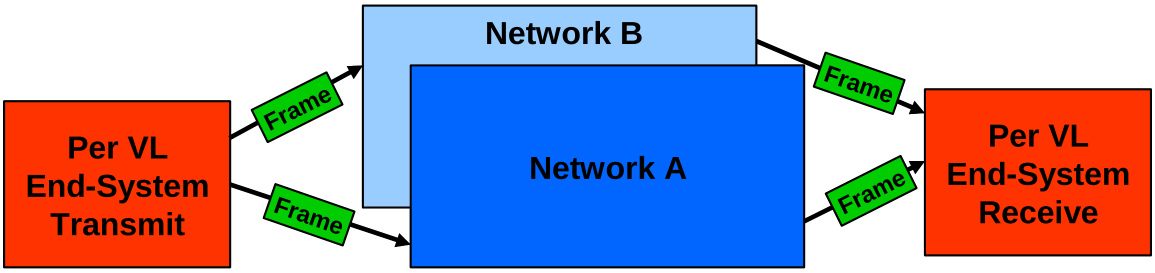
Figure 2-5 Sub-VL Round Robin Scheduling



- The Sub VL FIFO queues are read on a round-robin basis by the VL FIFO Queue to optimize the bandwidth of the VL

End-system ports, links and switches are duplicated for redundancy as shown in Figure 2-6. Frames are concurrently transmitted over both networks.

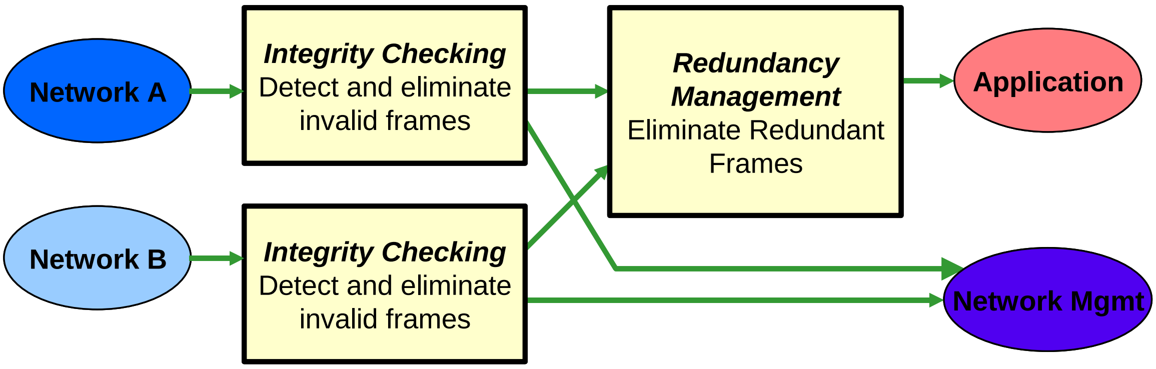
Figure 2-6 Redundancy Management



- Frames are transmitted simultaneously over both networks
- On the Receiving End-System, “First Valid Frame wins”

Integrity checking is done per VL and per Network as shown in Figure 2-7.

Figure 2-7 Redundancy Management & Integrity Checking on a Receive End-System

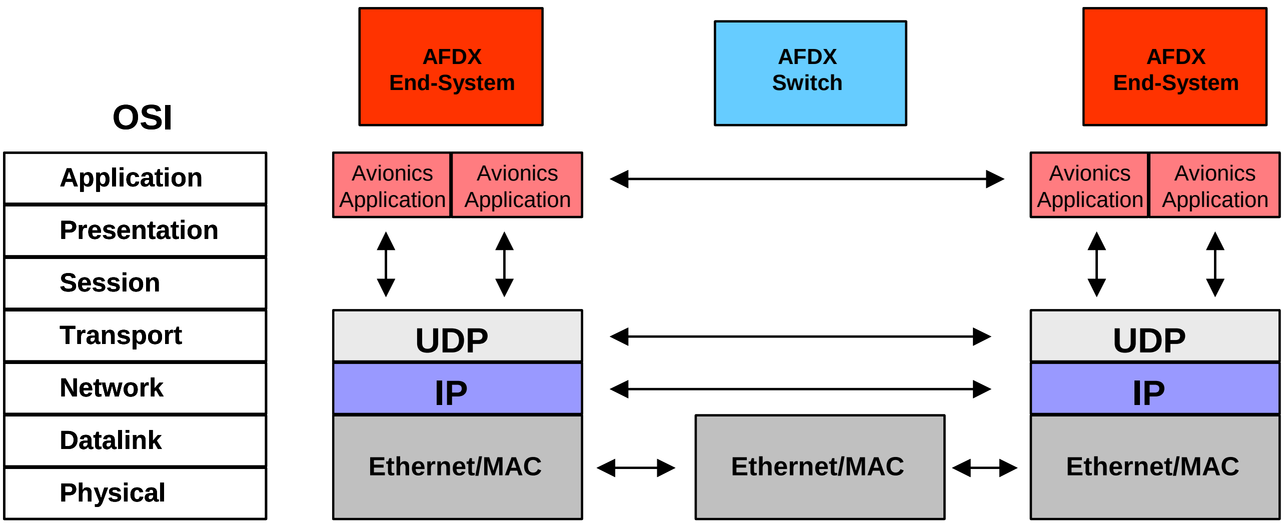


- Integrity Checking is based on Sequence Number and MCFL (Maximum Consecutive Frames Lost).
- All Invalid Frames are discarded

2.2 AFDX Protocol Stack

As shown in Figure 2-8, Avionics applications residing at End-Systems exchange messages via the services of the User Datagram Protocol (UDP) Layer.

Figure 2-8 AFDX Protocol Stack



- AFDX switches switch AFDX frames based on the MAC Destination Address

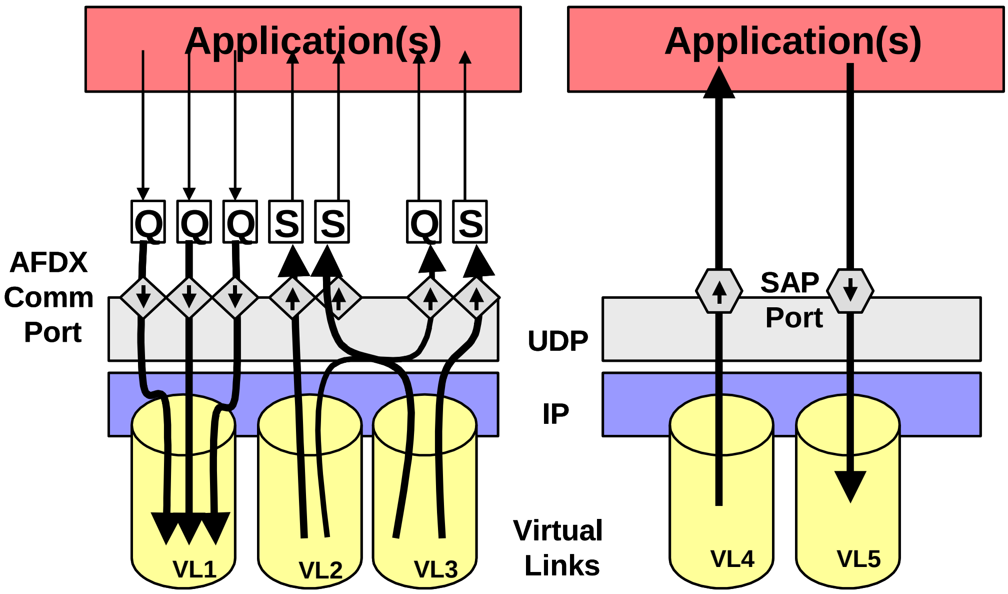
Applications send/receive messages through two types of UDP ports as shown in Figure 2-9: AFDX Communication Ports or Service Access Point (SAP) Ports. AFDX Comm Ports communicate via a static "connection" i.e., the IP/UDP Source/Destination addresses are contained in the AFDX frame header are static. SAP ports, however, are "connectionless" i.e., the E/S application can dynamically determine the destination address (IP address and UDP port number) for messages transmitted, and messages can be received from multiple sources.

AFDX Comm ports provide two different types of services as defined by ARINC 653:

Queuing services - AFDX messages may be sent over several AFDX frames (fragmentation by IP layer), no data is lost or overwritten

Sampling services - AFDX messages are sent in 1 Frame, data may be lost or overwritten.

Figure 2-9 AFDXComm and SAP UDP Ports



- AFDX Comm Ports are associated with an address “Quintuplet” consisting of:
  - UDP Src/Dest Port
  - IP Src/Dest Address
  - MAC Dest Address (VL)
- SAP ports dynamical define their destination (IP address and UDP port #) for each transmission
- Each AFDX Comm Port and each SAP port is associated with a UDP Port
- Each UDP port is associated with a Virtual Link over which all messages sent/received via the port travel. The VL used by the port is identified by a VL field within the IP layer’s destination address
- The IP layer handles the fragmentation/reassembly functions required by Queuing and SAP ports
- The Ethernet/MAC layer handles the Physical and Data Link functions in the AFDX network
- No routing tables are required to map the IP destination address to MAC destination address mapping
- Up to 100Mbps Ethernet is supported

2.3 AFDX Frame Format

The AFDX Frame Structure is shown in Figure 2-10. This section will provide further definition of the main components of the AFDX Frame Structure including:

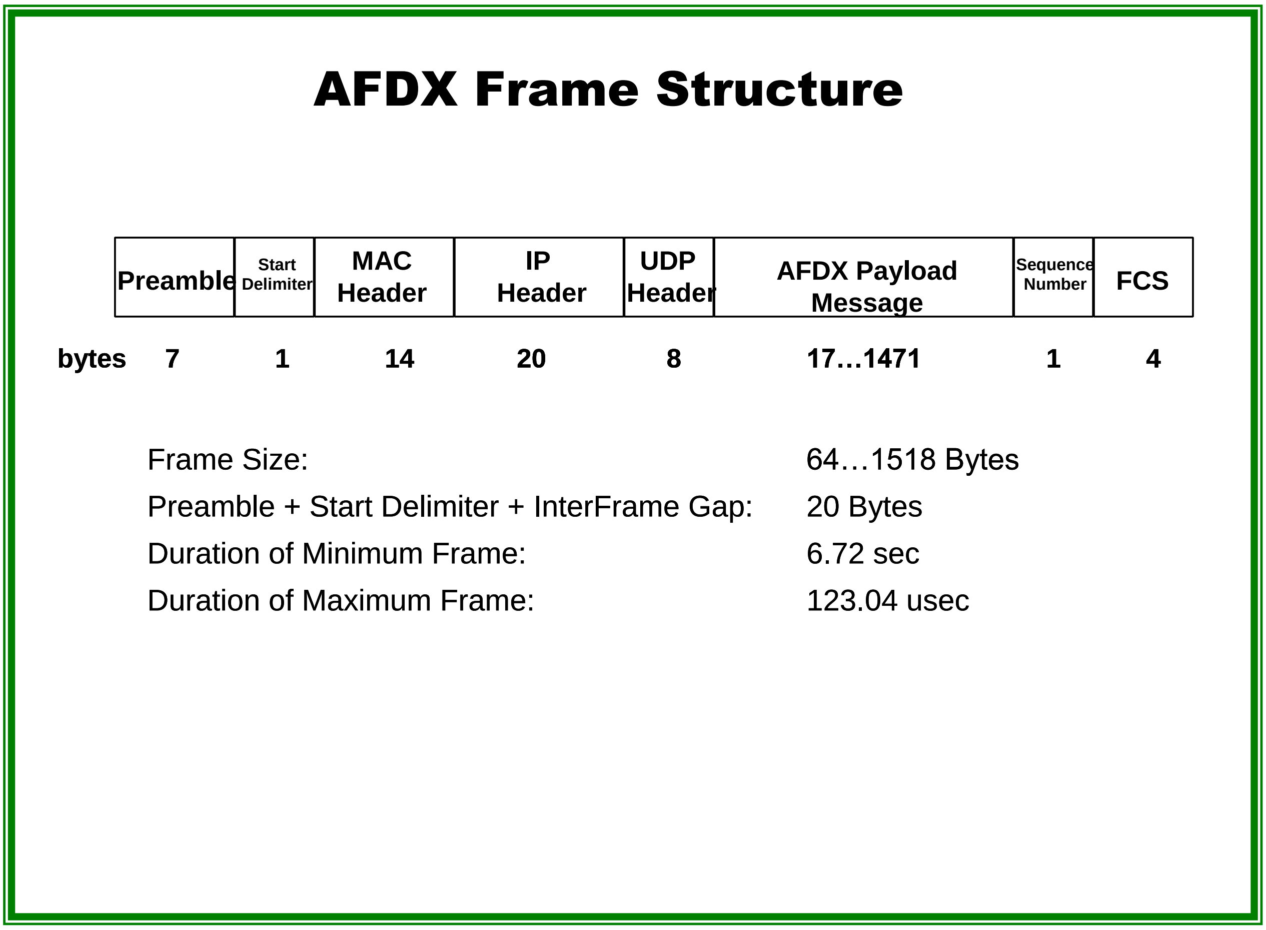
a. MAC Header

b. IP Header

c. UDP Header

d. AFDX Payload.

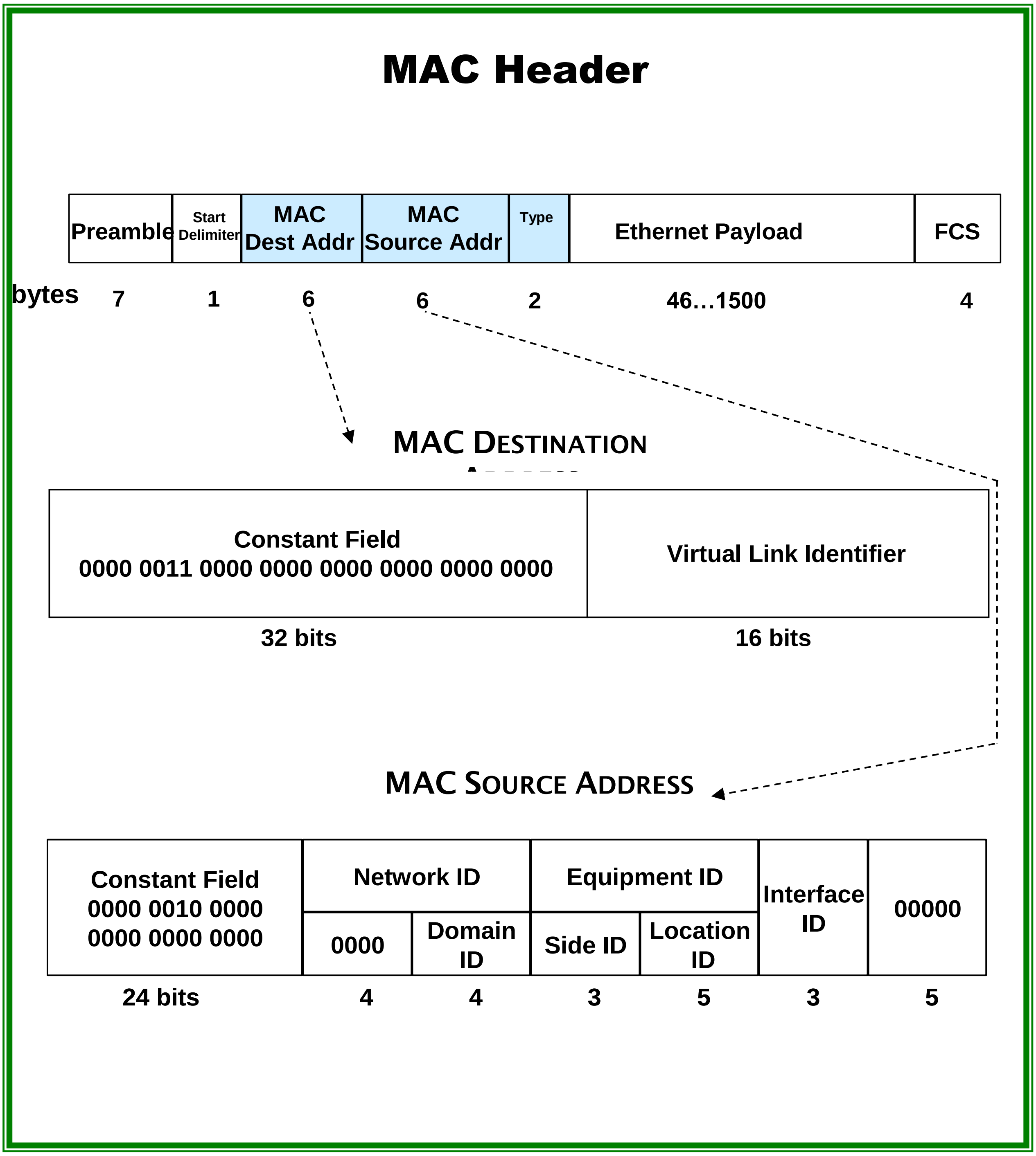
Figure 2-10 AFDX Frame Structure



AFDX - MAC LAYER

The MAC header is comprised of a Source and Destination Address, and a Type Field. Each address is 48 bits wide. The Destination Address identifies the **virtual link**. The Source Address is a Unicast Address. The Destination Address is a Multicast Address.

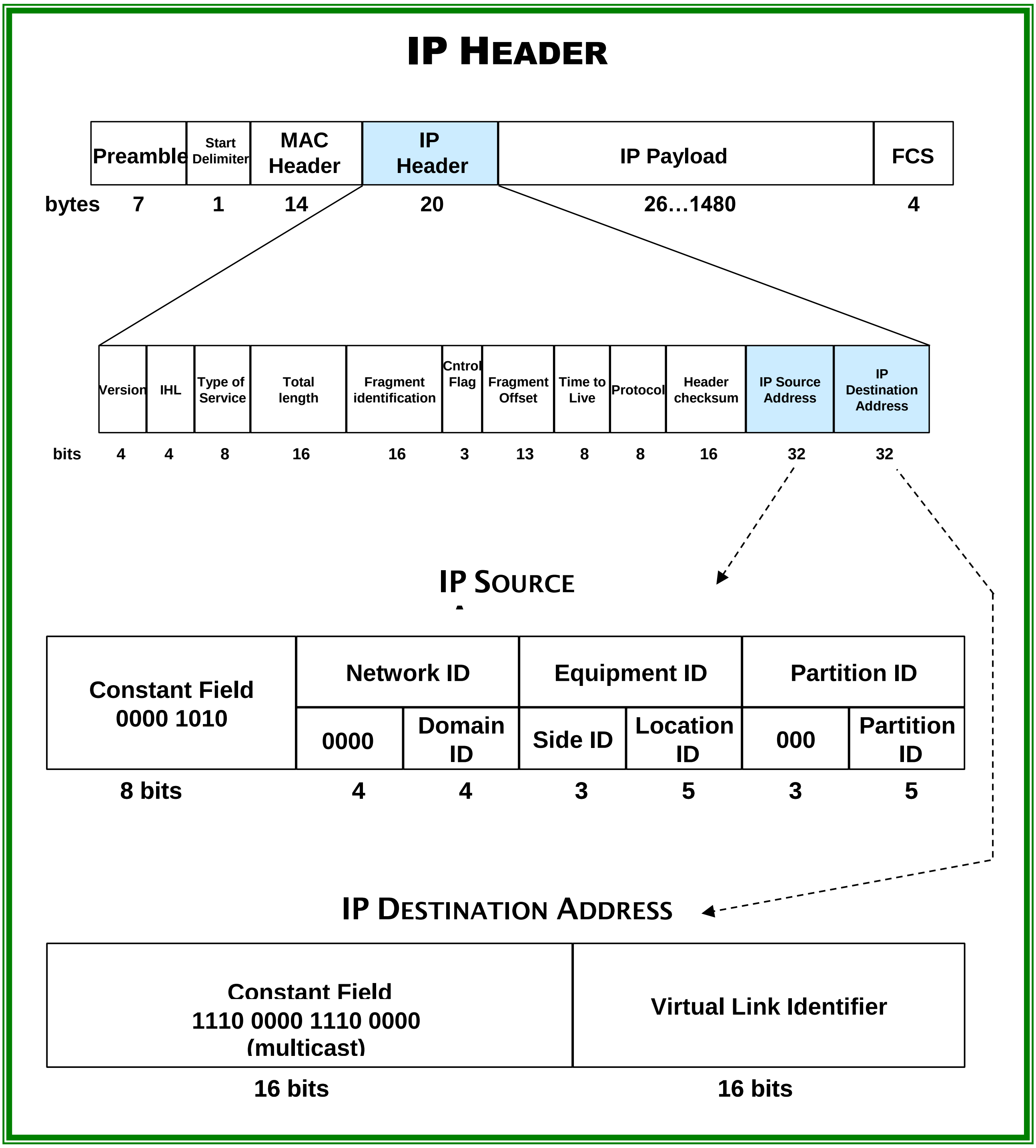
Figure 2-11 MAC Header



AFDX - IP (INTERNET PROTOCOL) LAYER

Figure 2-12 shows the IPv4 header and expands the Source and Destination addresses. IP Source Address is Unicast to identify the transmitter. IP Destination Address is Unicast to identify the target subscriber or is Multicast.

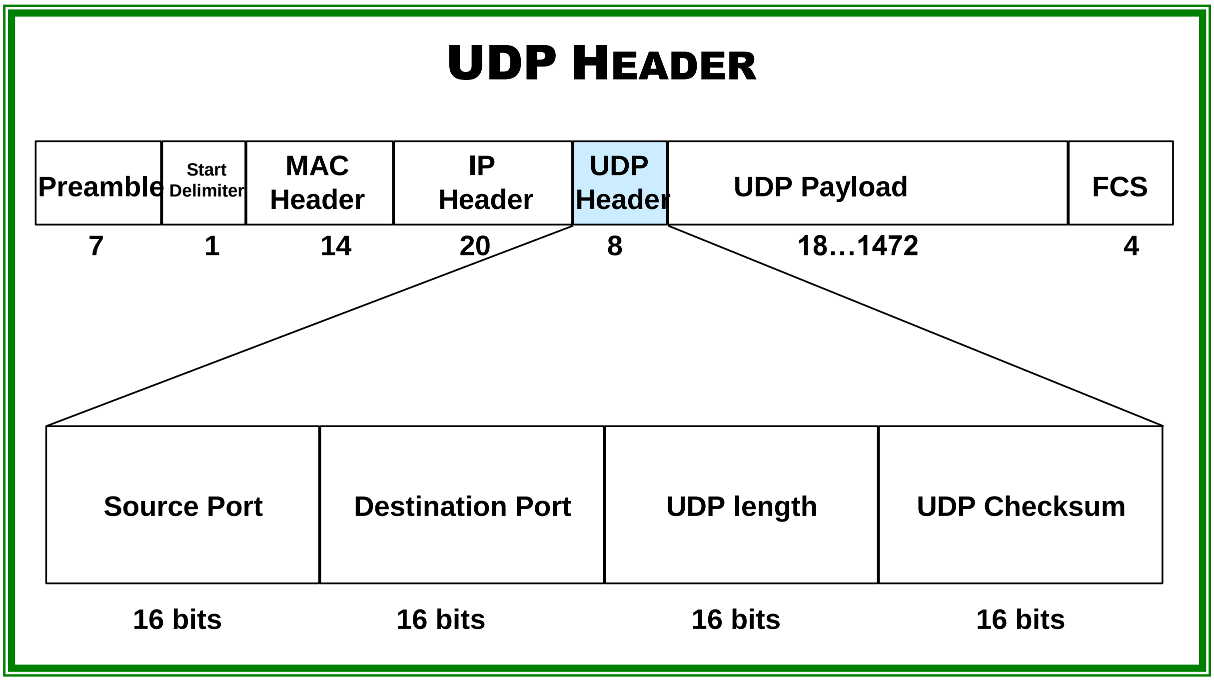
Figure 2-12 IP Header



AFDX - UDP (User Datagram Protocol) Layer

Figure 2-13 shows the UDP layer. The UDP layer takes messages from the application process, attaches source and destination port number fields for the multiplexing/demultiplexing service, adds the UDP length and Checksum, and passes the resulting "segment" to the IP layer.

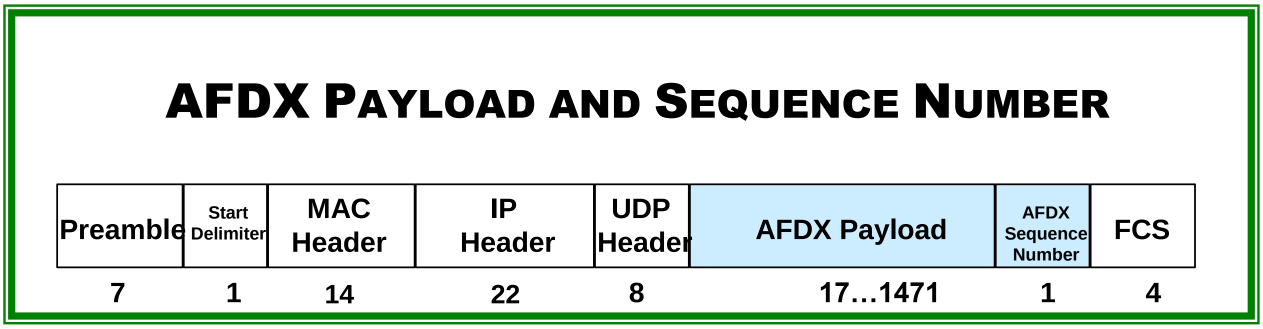
Figure 2-13 UDP Header



AFDX Payload and Sequence Number

Figure 2-14 shows the AFDX payload contents which include a sequence number (0-255). The sequence number is added by the transmitting End-System for each transmitted consecutive frame of the same VL on the AFDX Network. It starts with 0, then wraps around to 1 when it exceeds 255.

Figure 2-14 AFDX Payload and Sequence Number



3 AFDX OVERVIEW

The AIM's family of AFDX modules providing full function test, simulation, monitoring and databus analyzer functions for AFDX applications. The key technical features of the AFDX Bus Interface modules are:

- Each AFDX port can act as Traffic Generator/ Simulator and Receiver/ Monitor
- Supports AFDX- port related Frame Statistics
- Provides configurable Redundancy Management for AFDX Receive and Transmit ports
- Supports operational speeds of either 100Mbps (default) or 10Mbps, for GNET 1Gbps
- IRIG-B Time Tagging with a resolution of 1µsec

Figure 3-1shows two typical application scenarios for any of AIM's AFDX Bus Interface Cards.

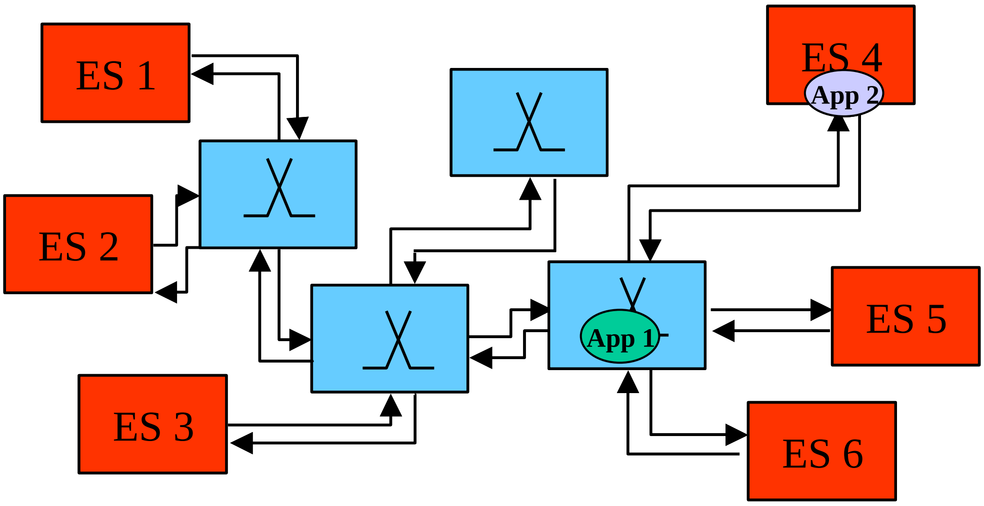
This section will provide an overview of the following AFDX-Module characteristics:

a. Functional overview of
   - Traffic Generation
   - Traffic Receive/Monitor Operation

b. Hardware Overview

c. Software Overview
   - Software Architecture
   - Board Support Package (BSP) Contents
   - Creating Your Own Host Application.

Figure 3-1 FDX Bus Interface Card Application Scenarios



App 1 — **Application Scenario 1**
Switch Development/Testing

Tx1/2 — Configure one AFDX Transmit Port for **Generic Transmit Mode** to generate data as if from multiple End Systems with varying PGWT & IFGs.

Rx1 — Configure one AFDX Receive Port for **Chronologic Receive (Monitor)** mode to record and monitor data transmitted at one switch output.

Rx2 — Configure one AFDX Receive Port for **VL-Oriented Receive** mode to verify proper switching at another switch output.

App 2 — **Application Scenario 2**
ES4 Development/Test

Tx1/2 — Configure the AFDX Transmit Ports for **UDP Port-Oriented Simulation** (Redundant) to simulate the generation of VLs transmitted by ES4.

Rx1/2 — Configure the AFDX Receive Ports for **VL-Oriented Receive Operation** (Redundant) to receive the generation of VLs switched to ES4.

3.1 AFDX Functional Overview

The functionality of the AFDX module can be divided into the following:

a. AFDX Traffic Generation

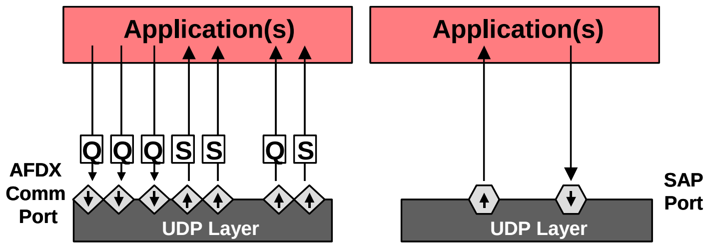
b. AFDX Receive/Monitor Operation.

These functions are defined in the following sections.

3.1.1 AFDX Traffic Generation

The three **AFDX Traffic Generation** modes of data transmission are listed below:

a. **UDP Port-Oriented Simulation** - This mode simulates the AFDX Comm ports (defined by ARINC-653) and SAP ports. AFDX Comm Ports communicate via a static "connection" i.e., the IP/UDP Source/Destination addresses are contained in the AFDX frame header are fixed. SAP ports, however, are "connectionless" i.e., the E/S application can dynamically determine the destination address (IP address and UDP port number) for messages transmitted, and messages can be received from multiple sources.



An AFDX Comm port provide two different types of services:

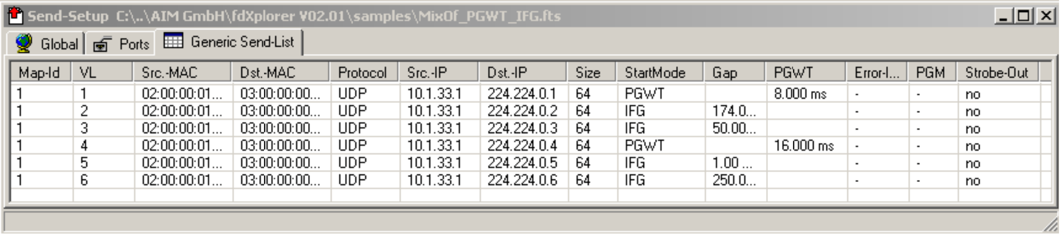
- **Queuing service** - AFDX messages are sent over several AFDX frames (fragmentation by IP layer), no data is lost or overwritten.

- **Sampling service**- AFDX messages are sent in 1 frame, data may be lost or overwritten.

The end-systems, VLs, and partitions are represented by the IP-Addresses and communication end points are described by the AFDX Comm UDP-Port.

SAP ports can also transmit and receive AFDX messages that are sent over one or more AFDX frames, however, the protocol for that communication is not determined by ARINC 653.

b. **Generic Transmit Operation** - This mode provides maximum flexibility and consists of a frame based transmission sequence. Each frame can be associated with attributes defining information about the relative timing between the frames, error injection, payload-generation mode, transmission skew in redundant operation mode and/or special events like a digital output strobe-signal. For high-throughput, special payload-generation modes can be used, so the hardware takes parts of the frame-data from static send-fields. Because all frames must be pre-buffered on the hardware, the number of frames is limited to the board-resources.



Map-Id	VL	Src.-MAC	Dst.-MAC	Protocol	Src.-IP	Dst.-IP	Size	StartMode	Gap	PGWT	Error-I...	PGM	Strobe-Out
1	1	02:00:00:01...	03:00:00:00...	UDP	10.1.33.1	224.224.0.1	64	PGWT		8.000 ms	-	-	no
1	2	02:00:00:01...	03:00:00:00...	UDP	10.1.33.1	224.224.0.2	64	IFG	174.0...		-	-	no
1	3	02:00:00:01...	03:00:00:00...	UDP	10.1.33.1	224.224.0.3	64	IFG	50.00...		-	-	no
1	4	02:00:00:01...	03:00:00:00...	UDP	10.1.33.1	224.224.0.4	64	PGWT		16.000 ms	-	-	no
1	5	02:00:00:01...	03:00:00:00...	UDP	10.1.33.1	224.224.0.5	64	IFG	1.00 ...		-	-	no
1	6	02:00:00:01...	03:00:00:00...	UDP	10.1.33.1	224.224.0.6	64	IFG	250.0...		-	-	no

c. **Replay Operation**

- Physical Re-Transmission of pre-recorded network traffic

Table 3-1 defines the key features and differences between the UDP-Port Oriented transmission mode and the Generic Transmission mode.

Table 3-1 Transmission Mode Key Features

UDP Port-Oriented Simulation	Generic Transmit Operation
Simulation of network traffic in accordance with AFDX End System Detailed Functional Specification	Autonomous operation including sequencing of the outgoing data packets with programmable interframe gaps, without Host interaction.
Support Sampling & Queuing service implementation for multiple VL's	Sampling and Queuing services can be specified for the frames defined within the generic frame sequence.
Programmable Packet Intermessage gap	Flexible packet scheduling by using various packet timing modes
VL simulation with Traffic Shaping and Sequence Numbering	Programmable Sequence Numbering operation within the transmission list
Synchronization of traffic between multiple AFDX- ports	Synchronization of traffic between multiple AFDX- ports
Start of traffic generation on external strobe	Start of traffic generation on external strobe and strobe generation on packet transmission
Start of traffic on absolute time, which allows the synchronization of multiple streams/modules.	Start of traffic on absolute time, which allows the synchronization of multiple streams/modules.
	Various Payload Generation modes, which reduces the data exchange at high performance transmission
	Transmission of the 'Packet Start Time stamp' within the payload of the outgoing frame
Enable/Disable specific VL's	Enable/Disable specific VL's
Error injection capabilities: - Physical Error Injection on Frame Level (CRC, Interframe Gap, Frame Size, Byte Alignment, ..) - Logical Error Injection on MAC- /IP Layer - Wrong Sequence Numbering - Timing Error Injection (BAG- violation)	Error injection capabilities: - Physical Error Injection on Frame Level (CRC, Interframe Gap, Frame Size, Byte Alignment, ..) - Logical Error Injection on MAC- /IP Layer - Wrong Sequence Numbering - Timing Error Injection (BAG- violation)
Redundant or non- redundant Operation available	Redundant or non- redundant Operation available

3.1.2 AFDX Receive / Monitor Operation

The **Receive AFDX Receive / Monitor Operation Modes** define how captured frames are stored on the board and how data is filtered. The two **Receive Modes** are defined as follows:

a. **VL-Oriented Receive Operation -** In this Receive Mode the UDP Port can receive and store messages for either "connection" oriented (AFDX Comm Ports) or "connectionless" oriented (SAP) ports. The Receive AFDX Comm ports are characterized by the address-quintuplet, (VL, Src.-IP, Dst.-IP, Src.-UDP, Dst.-UDP), each with its own message storage area. In this mode, the user must specify the exact address quintuplet in order for the VL frames to be captured. SAP receive ports, however, may receive AFDX messages from multiple sources. Therefore, the user only specifies the VL and UDP/IP destination adress in order for the VL frames to be captured. The source of the AFDX frame is only determined after the message has been received. (Trigger capability is not provided in this receive mode.)

b. **Chronological Receive Operation (Monitor Mode)-** In this Receive mode all captured frames are stored in one single memory area. In this mode, all VL data streams are captured, in addition, the user can specify additional VL filters/checking to be performed if desired. This mode provides for recording/saving the captured data for replay. Four **Capture modes** are available.

- **SingleShot-Standard**
  In this mode, each port uses a pre-defined onboard memory area (**singleshot memory**) for capturing frames. After this memory is filled with frames no more frames will be stored. The size of singleshot-memory depends on your board type and RAM-Size. **Trigger Control Blocks** (TCBs) can be used in this mode to define the trigger condition that will start data capture (default capture start is when a frame is received) and how much "pre-trigger data" is to be stored in the Monitor Buffer.

- **SingleShot-Selective**
  This mode is very similar to SingleShot-Standard mode, but **Trigger-Control-Blocks** are used for **filtering,** i.e. **what** data will be captured. Before a frame is saved in the SingleShot-memory, it will be evaluated using the active TCB. Only those frames which meet the TCB condition will be saved.

- **Continuous**
  In this mode, the SingleShot-memory is used as a **ring-buffer**. As soon as the memory is full, old frames will be overwritten with new frames (**wrap-around**). **Trigger-Control Blocks** can be used in this mode to

define the trigger condition that will start data capture (default capture start is when a frame is received).

- **Record**
  In this mode, the Monitor buffer is organized in the same way as in Continuous mode. However, the **frames will be written directly to a user-specified file**. **Trigger-Control Blocks** can be used in this mode to define the trigger condition that will start data capture (default capture start is when a frame is received).

Table 3-2 defines the key features and differences between the Chronologic and VL-Oriented Receive modes.

Table 3-2 Reception Mode Key Features

VL-Oriented Receive Operation	Chronological Receive Operation (Monitor Mode)
VL- oriented multi buffering and Time Stamping of received data packets	Full chronological traffic monitoring and analyzing with relative gap time measurement and absolute Time Stamping, concurrently with any other mode of operation
VL-Oriented Filtering with optional Second Level Filtering on Generic packet parameters	VL oriented Filtering with optional Second Level Filtering on Generic packet parameters
	Comprehensive trigger capabilities for traffic capturing (VL, header info, error, data, receive time)
	Programmable Data Capture modes providing Continuous, Record, and Selective capture capability
Independent, programmable Buffer Size for each VL	Programmable Monitor Buffer Size, which allows a massive on-board data buffering
Interrupt generation on dedicated Buffer Event	Interrupt generation on Buffer Events
VL-Oriented receive Counters and Error Accumulators are provided	VL-Oriented receive Counters and Error Accumulators are provided
Redundancy Management at redundant port configuration	Redundancy Management at redundant port configuration
Physical Error Detection on Frame Level - CRC Error detection - Frame Size Violation - Interframe Gap violation - Wrong Byte Alignment - Undefined Symbol received	Physical Error Detection on Frame Level - CRC Error detection - Frame Size Violation - Interframe Gap violation - Wrong Byte Alignment - Undefined Symbol received
AFDX- specific Error Detection - Traffic Shaping verification - Verification of static header fields (MAC, IP) - Integrity Checking of VL related packets	AFDX- specific Error Detection - Traffic Shaping verification - Verification of static header fields (MAC, IP) - Integrity Checking of VL related packets
	Strobe generation on dedicated Trigger Event

3.2 Hardware Overview

This programmer's guide is applicable to AIM's API-FDX-2, AMC-FDX, APM-FDX-2, fdxTap and APX-GNET-2/4 Bus Interface Cards. An h/w overview is available in the H/W manual of the board which is part of the deliverey.

3.3 AFDX Software Overview

This section will provide an overview of the AFDX software including:

a. Software Architecture

b. Board Support Package (BSP) Contents

c. Creating a New Microsoft Visual C/C++ Application Program.

The instructions for using the API function calls are defined in Section 4.

3.3.1 AFDX Software Architecture

The AIM "Common Core" design, as shown in the previous section, provides for the utilization of a common application s/w library of function calls to support host application interfaces to the AFDX device(s). Figure 3-2 shows the high-level software architecture of the PCI-FDX module and it's interface to a host computer application.

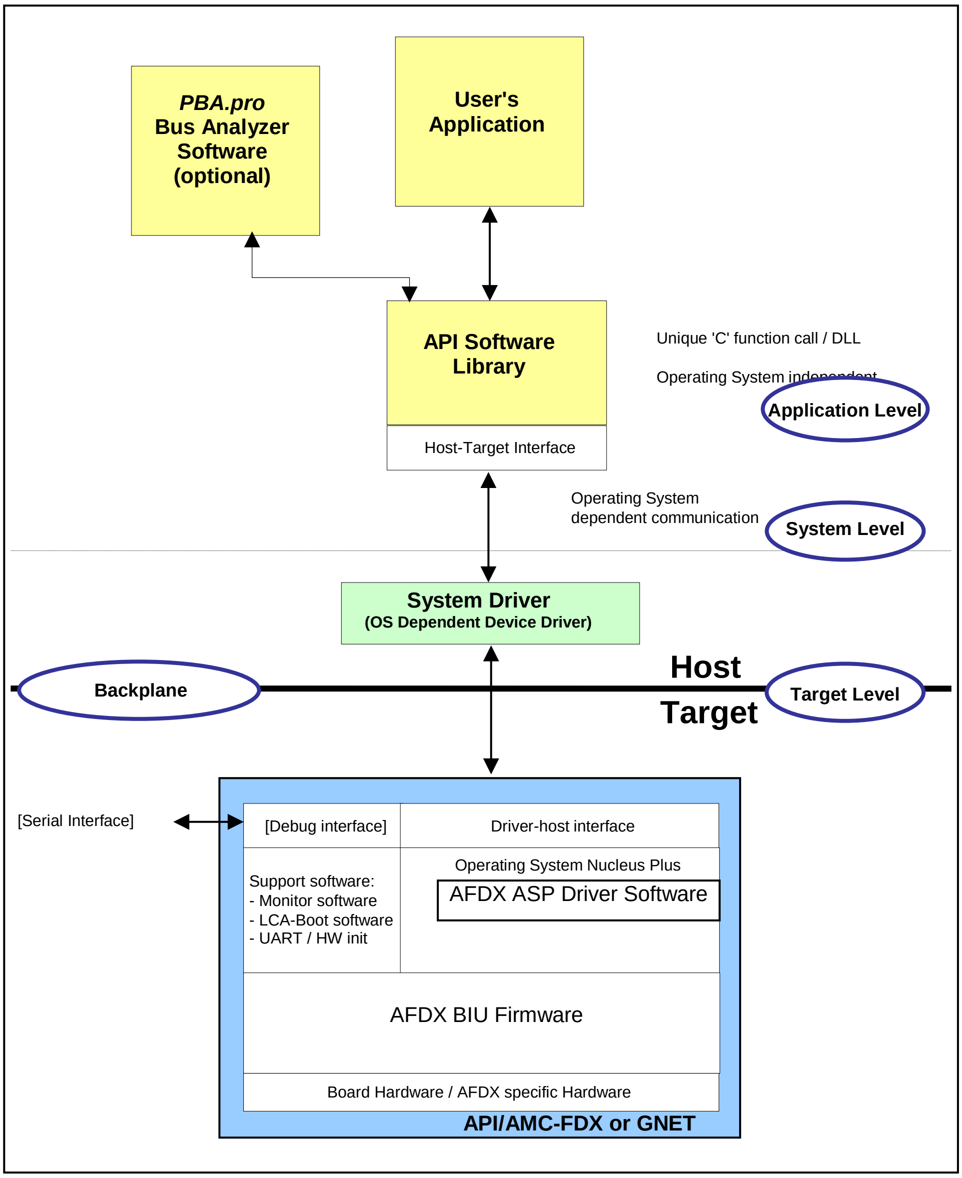
As shown in Figure 3-2, the API S/W Library is utilized by the User's Application program to control the AFDX target module. (As an option, the application developer can utilize the AIM ***PBA.pro*** Bus Analyzer Software Bus Monitor function to monitor bus traffic setup by the User's Application.) Both ***PBA.pro*** and the User's Application program utilize the same API S/W Library.

The API S/W Library encapsulates operating system specific handling of Host-to-Target communication in order to support multiple platforms with one set of library functions. Operating systems and compilers supported by the API S/W Library are defined in Table 3-3.

Table 3-3 Compatible Operating Systems / Compilers

Operating Systems	Compilers
Windows 7/8/10 (32 bit, 64 bit) Linux (32 bit, 64 bit) VxWorks LynxOS	Microsoft Visual Studio (2013 or higher)

Figure 3-2 Host/Target Software Interface Diagram



As shown in Figure 3-2, the API S/W Library consists of "C" functions which can be called within your application program to setup and control the PCI-FDX module(s).

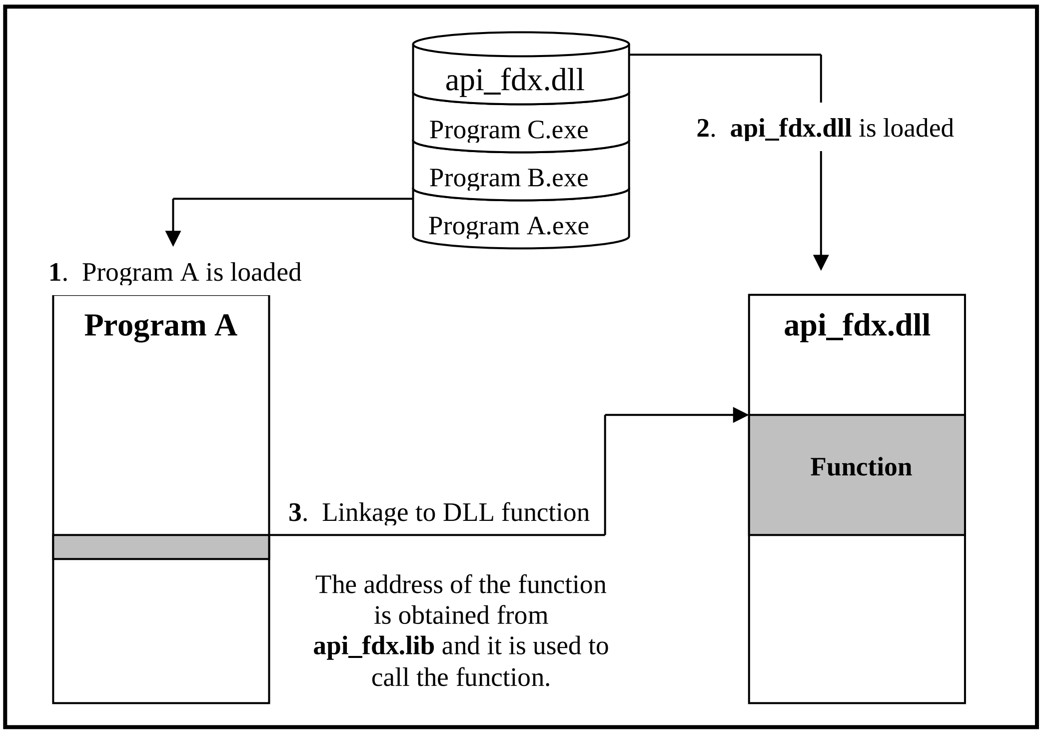
The AIM API S/W Library is supplied as a dynamic link library (DLL) containing the collection of functions used to setup and command the PCI-FDX modules. A function in a DLL is only connected to a program that uses it when the application is run. This is done on each occasion the program is executed as shown in Figure 3-3. Two binary files are utilized by the application program including:

a. `api_fdx.dll` - contains the executable code for the DLL.

b. `api_fdx.lib` – defines the items exported by an AIM API S/W Library DLL in a form which enables the linker to deal with references to exported items when linking a program that uses the AIM API S/W Library DLL function.

> ***Note:** In order to utilize the API S/W Library, `api_fdx.lib` must be linked to the application program. Section 3.3.3provides further detail.*

Figure 3-3 DLL and Program Interfaces



The `api_fdx.lib` and `api_fdx.dll` files are provided in two forms for 32-bit and 64-bit OS use with Microsoft Visual C/C++.

3.3.2 AFDX Board Support Package

The BSP is downloaded to your computer upon s/w installation for your device. (Please see the corresponding **Getting Started Manual** for further information regarding s/w installation.)

3.3.3 Creating a New Microsoft Visual C/C++ Application Program

This section will review the following:

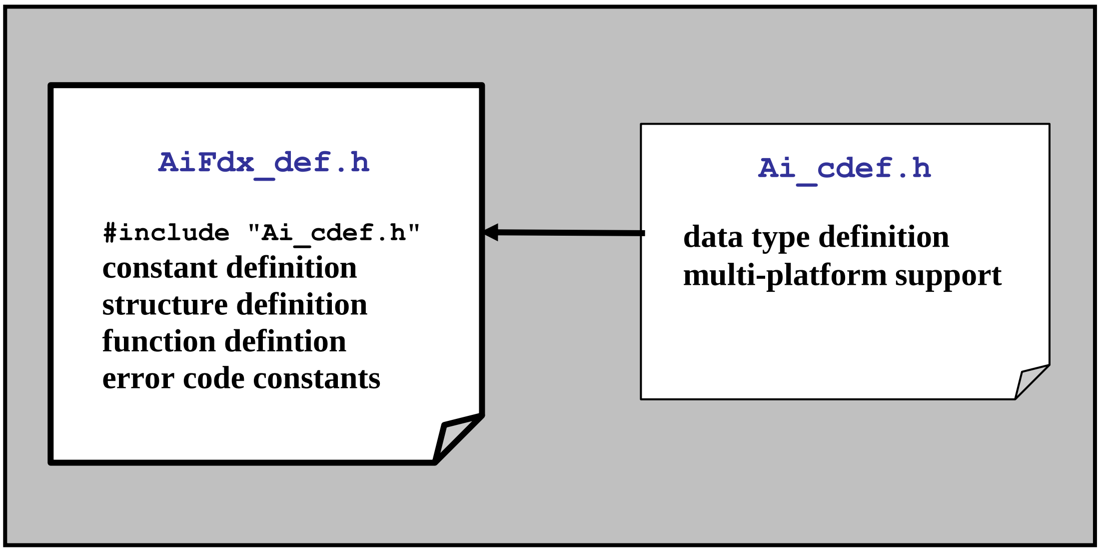
a. API S/W Library header files that need to be included in your application program

b. Windows C/C++ steps to create and compile a new application program.

3.3.3.1 Header File Defines for New Application Programs

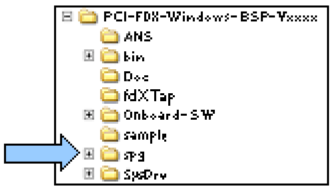
For all platforms, the two C-syntax header files shown in Figure 3-4 are provided. Only the **`AiFdx_def.h`** header file needs to be included in your application program. (This header file provides for the inclusion of the **`Ai_cdef.h`** header file.)

Figure 3-4 API S/W Library Header Files



These header files are located in:

`x:\Program Files\AIM GmbH\PCI-FDX-Windows-BSP-Vxxxx\spg`



All header files need to be included in the search path when compiling your new program as described in the following section.

3.3.3.2 Creating and Compiling Your Application Program

Your new Console Win32 Application program can be created by using a sample program (See Section 1, Program Samples) as a basis and modifying it as needed. Once your new application has been created, there are three additional steps to configuring the Microsoft Visual C/C++ application before compiling to insure your program executes without error including:

a. **Adding proper search paths** for the API S/W Library include files

b. **Adding the preprocessor definition** required for the PCI-FDX device

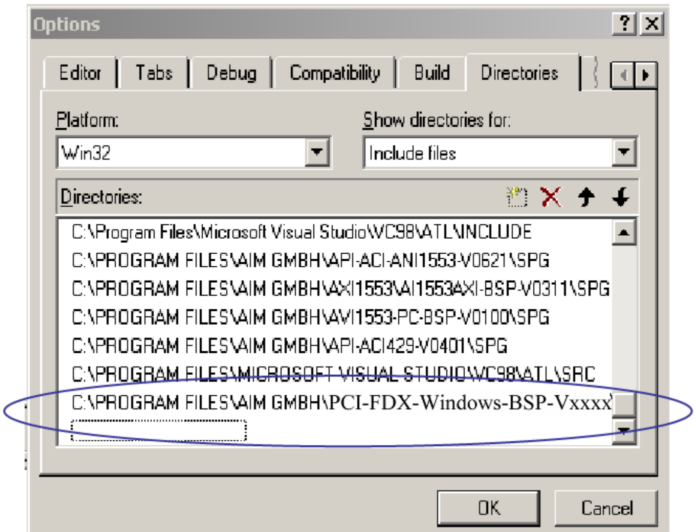
c. **Linking the application program** to the **`api_fdx.dll`** via connection to **`api_fdx.lib`** and compiling your program

> ***Note:*** ***`api_fdx.dll`*** ***must be located in the same directory as the User's Application executable(s).***

Please review the following steps to accomplish the items above.

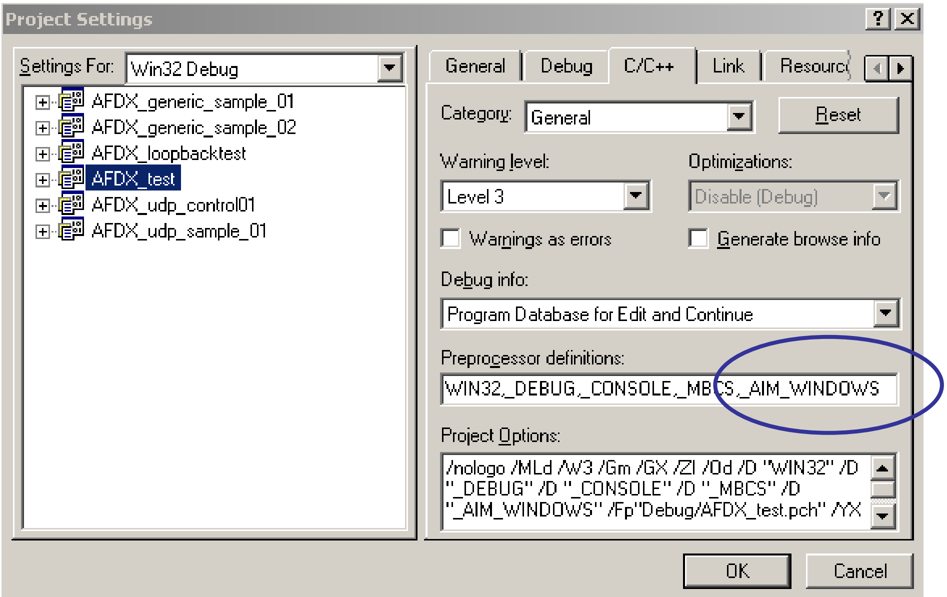
▶ **To add the proper search paths for the API S/W Library Header files perform the following steps:**

1. Select **Tools | Options**
   The Options window will pop up.
2. Select the **Directories Tab**
3. For **Show directories for:**, select **Include Files**
4. Add the Directory with the include files:
   `x:\Program Files\AIM GmbH\PCI-FDX-Windows-BSP-Vxxxx\spg`
5. Select **OK**



▶ **To add the preprocessor definition required for the API/AMC-FDX device:**

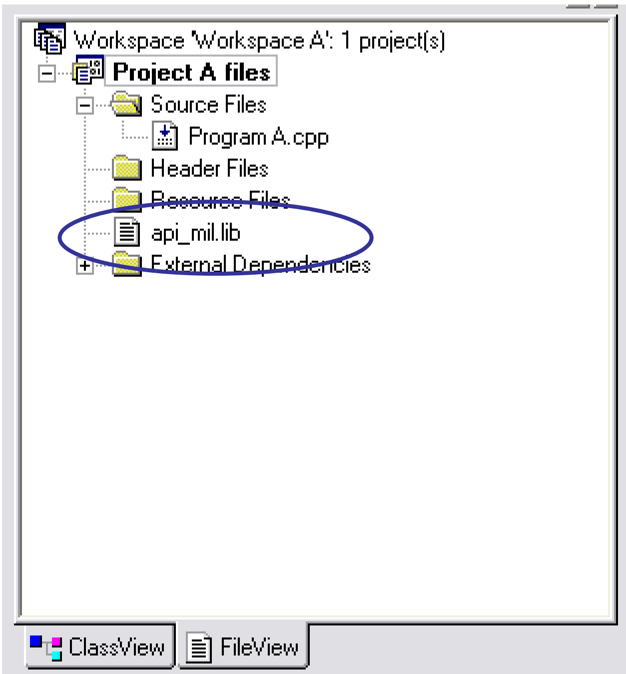
1. Select **Project | Settings**
   The project settings window will pop up.
2. Select the **C/C++** tab
3. Under Preprocessor Definitions enter **_AIM_WINDOWS** and **_AIM_FDX**



4. Select **OK**

▶ **To link the `api_fdx.lib` and `api_fdx.dll` to the application program and compile your program perform the following steps**

1. Select your project file (in example, Project A Files)
2. Select **Project | Add to Project | Files...**
   An "Insert Files into Project" window will pop up.
3. For **Files of Type:** entry, select **Library Files (.lib)**
4. For **File Name**: Look in
   `x:\Program Files\AIM GmbH\PCI-FDX-Windows-BSP-Vxxxx\bin\`
   `release`
5. Select **`api_fdx.lib`**
   `api_fdx.lib` will be added to your project.



6. Now, build the project by selecting **Build | Build *your program name*.exe**
7. Copy
   `x:\Program Files\AIM GmbH\PCI-FDX-Windows-BSP-Vxxxx\bin\release\`**`aim_Fdx.dll`**
   to
   `x:\your project location\debug\`
8. The project can now be run by selecting **Build | Start Debug | Go**

THIS PAGE INTENTIONALLY LEFT BLANK

4 PROGRAMMING USING THE API LIBRARY

Let's now begin to focus on the concepts of writing application programs to setup and control the PCI-FDX module from the host. First, we can look at the complete list of API Library function calls available for the host application developer.

The API S/W Library function calls are divided into the following subgroups and listed in the tables which follow:

a. Library Administration Functions (

Table 4-1) - used to gain general access to the physical resources provided on the FDX-2/4 board. There are also functions to observe the resources. The resources are divided into board- and port-resources.

b. **Target Independent Administration Functions** (Table 4-2) - utility functions to help with IRIG time conversions, error translation, and Monitor Buffer decoding.

c. **System Configuration** (Table 4-3)- Board level functions to reset the board, setup IRIG time, status the version number of the board software and perform resource tests.

d. **Transmitter Functions** (Table 4-4)) - are divided into three categories:

- **Global Transmitter Functions** - used for when you are in either the UDP Port-Oriented or Generic Transmit modes. These functions provide transmitter mode control and status, trigger line I/O setup, and VL enable/disable.

- **Generic or Replay Transmitter Functions** - used to define the AFDX Generic AFDX Frame content including error injection, whether the frames are transmitted cyclically or a certain number of times, and the attributes of the transmission protocol, i.e. IFG, PGWT and skew.

- **UDP Port-Oriented Transmitter Functions** - used to define the VL and UDP port (AFDX Comm port or SAP port), AFDX Frame content including error injection, and attributes of the transmission protocol, i.e. BAG, and skew.

e. **Receiver Functions** (Table 4-5) - are divided into three categories:

- **Global Receiver Functions** - used for when you are in either the Chronologic Receive Operation (Monitor mode) or VL-Oriented Receive modes. These functions provide the receiver mode control, reception control, status, VL control and status, and Receiver Trigger line configuration.

- **VL-Oriented Receiver Functions** - used to simulate a Receive UDP "connection" (AFDX Comm port) or "connectionless" (SAP port) for received AFDX message storage.

- **Chronologic Receive Operation (Monitor) Functions** - used to setup the Monitor capture mode, trigger(s) defining when and what to capture, and to obtain status

Figure 4-1 shows the structure of a basic application program and the Function Call categories associated with each major part of the program. The following sections will guide you in the use of the API S/W Library functions. For detailed information regarding each function call please refer to the **Reference Manual AFDX/ ARINC-664**.

Table 4-1 Library Administration Functions

Function	Description
FdxInit	Initializes the Interface Library. Returns a list of servers.
FdxQueryServerConfig	Returns a list of resources of one server. Connects additional server (additional to local available resources)
FdxQueryResource	Gets detailed information about a resource
FdxInstallServerConfigCallback	Provides a mechanism to notify PnP device changes
FdxLogin	Login for one resource
FdxLogout	Logout from a resource
FdxInstIntHandler	Installs a user-defined interrupt handler function
FdxDelIntHandler	Deletes the user-defined interrupt handler function
FdxExit	Cleanup the Library internal used memory structures.

Table 4-2 Target Independent Administration Functions

Function	Description
FdxCmdFreeMemory	Frees memory, allocated by the Library, in the proper manner
FdxFwIrig2StructIrig	Converts an IRIG time in the format used by the Firmware to a structured format.
FdxStructIrig2FwIrig	Converts an IRIG time in the structured format to the format used by the Firmware.
FdxAddIrigStructIrig	Adds two IRIG time structures
FdxSubIrigStructIrig	Subtracts two IRIG time structures
FdxTranslateErrorWord	Translates a firmware encoded Error Word for Error Information on Receiver Side
GNetTranslateErrorWord	Translates a firmware encoded Error Word for GNET Error Information on Receiver Side
FdxInitTxFrameHeader	Supports a default initialization of a Transmit Header Structure, needed in Generic Transmit Mode
FdxProcessMonQueue	Processes data read via FdxCmdMonQueueRead.

Table 4-3 System Functions

Function	Description
FdxCmdBoardControl	Controls and resets the board operation mode.
FdxCmdIrigTimeControl	Reads and writes the onboard IRIG Time
FdxCmdStrobeTriggerLine	Provides a trigger output strobe on system command.
FdxReadBSPVersion	Reads version numbers of board software package components.
FdxCmdBITETransfer	Performs transfer tests using available port resources of one FDX board.

Table 4-4 Transmitter Functions

Function		Description
Global Transmitter Functions		
	FdxCmdTxPortInit	Initializes the transmitter
	FdxCmdTxModeControl	Defines the Mode of the transmitter
	FdxCmdTxControl	Starts and stops the transmitter
	FdxCmdTxStatus	Obtains global status information about the transmitter
	FdxCmdTxTrgLineCtrl	Controls Transmitter Associated Strobe Input/Output Lines
	FdxCmdTxVLControl	Controls VL (Enable / Disable)
	FdxCmdTxStaticRegsCtrl	Controls Static Transmit Registers
Generic or Replay Transmitter Functions		
	FdxCmdTxQueueCreate	Creates a Transmit Queue for AFDX generic frames
	FdxCmdTxQueueStatus	Retrieves Status of an AFDX Frame Transmit Queue
	FdxCmdTxQueueWrite	Writes AFDX Frames to the Queue
	FdxCmdTxQueueUpdate	Updates AFDX Frames of a generic Queue on the fly
UDP Port-Oriented Transmitter Functions		
	FdxCmdTxCreateVL	Creates a Virtual Link, which can be used for transmission.
	FdxCmdTxCreateHiResVL	Creates a Virtual Link, which can be used for transmission with a high resolution BAG.
	FdxCmdTxUDPCreatePort	Creates a AFDX Comm UDP port for transmission.
	FdxCmdTxUDPChgSrcPort	Changes the source of a UDP port.
	FdxCmdTxUDPDestroyPort	Destroys a configured AFDX Comm UDP port.
	FdxCmdTxUDPWrite	Writes one complete AFDX Payload message to a Tx AFDX Comm UDP port
	FdxCmdTxUDPBlockWrite	Writes one or more complete AFDX Payload Messages to multiple AFDX Comm UDP ports
	FdxCmdTxSAPCreatePort	Creates a SAP UDP port for transmission.
	FdxCmdTxSAPWrite	Writes an AFDX Payload message to a Tx SAP UDP port.
	FdxCmdTxSAPBlockWrite	Writes one or more AFDX Payload Messages to multiple SAP UDP ports
	FdxCmdTxUDPGetStatus	Retrieves the status of a transmission AFDX Comm or SAP UDP port
	FdxCmdTxUDPControl	Controls UDP Port operation (Enable / Disable and error injection)
	FdxCmdTxVLWrite	Writes raw entire Frames to the VL-Buffer (UDP functions above are N/A when using this writing method)
	FdxCmdTxVLWriteEx	Writes Frames to the VL-Buffer with extended frame control possibilities

Table 4-5 Receiver Functions

Function		Description
Global Receiver Functions		
	FdxCmdRxPortInit	Initializes receiver on this port
	FdxCmdRxModeControl	Defines the Mode of the receiver
	FdxCmdRxControl	Starts and stops the receiver
	FdxCmdRxStatus	Obtains status information about the receiver
	FdxCmdRxGlobalStatistics	Obtains global statistics about the bus load
	FdxCmdRxVLControl	Controls settings for each Virtual Link
	FdxCmdRxVLControlEx	Controls extended settings for each Virtual Link
	FdxCmdRxVLGetActivity	Obtains Activity information of one Virtual Link
	FdxCmdRxTrgLineControl	Controls Receiver associated Strobe Input/Output Lines
VL-Oriented Receiver Functions		
	FdxCmdRxUDPCreatePort	Creates an AFDX Comm UDP port
	FdxCmdRxUDPChgDestPort	Changes destination of a UDP port
	FdxCmdRxUDPDestroyPort	Destroys an AFDX Comm UDP port
	FdxCmdRxUDPRead	Reads one complete AFDX Payload message from a Rx AFDX Comm UDP port
	FdxCmdRxUDPBlockRead	Reads one or more AFDX Payload Messages from multiple AFDX Comm UDP ports
	FdxCmdRxSAPCreatePort	Creates a SAP UDP port for reception.
	FdxCmdRxSAPRead	Reads a complete AFDX Payload message from a Rx SAP UDP port.
	FdxCmdRxSAPBlockRead	Reads one or more complete AFDX Payload Messages from multiple SAP UDP ports
	FdxCmdRxUDPControl	Allows a host interrupt on UDP frame reception
	FdxCmdRXUDPGetStatus	Obtains the Status of a SAP or AFDX Comm UDP port
Chronologic Receiver Operation (Monitor) Functions		
	FdxCmdMonCaptureControl	Defines the capture mode
	FdxCmdMonTCBSetup	Defines a Trigger Control Block
	FdxCmdMonTrgWordIni	Initializes the Monitor Trigger Word
	FdxCmdMonTrgIndexWordIni	Initializes the Monitor Trigger Index Word
	FdxCmdMonTrgIndexWordIniVL	Initializes the VL specific Monitor Trigger Index Word
	FdxCmdMonGetStatus	Obtains the Status of a Monitor port
	FdxCmdMonQueueControl	Creates a Queue, associated with the Monitor
	FdxCmdMonQueueRead	Reads data from a Monitor Data Queue
	FdxCmdMonQueueSeek	Sets the internal Read index to a Monitor Data Queue
	FdxCmdMonQueueTell	Gets the internal Read index to a Monitor Data Queue
	FdxCmdMonQueueStatus	Shows the status for a monitor capture queue of a receiver port

Figure 4-1 Basic Application Program Structure

Initialization--->Board setup--->Tx Port Setup---> Rx Port Setup---> Start Tx/Rx ---> Retrieve Status ---> Shutdown

Decide how you want to utilize the full-duplex ports on your board.
Determine the Transmit/Receive modes your application requires.

1

Initialization
(1) API Library
(2) Board Login
(3) Port Login

Library Administration Functions
&
Target Independent Administration Functions

Board Handle
Port Handles

2

Board Setup
(1) single or redundant mode, bit rate & MAC/IPheader verification register
(2) IRIG time

System Functions

3

Port Tx Setup
(1) Assign Portmap ID to each Tx port
(2) Define Transmitter mode (UDP Port-Oriented, Generic Transmit or Replay)

Global Transmitter Functions

UDP Port-Oriented Transmitter Functions

UDP Port-Oriented Setup
(1) Define the VL & Sub VL characteristics
(2) Write UDP port messages created to Tx port

Generic/Replay Transmit Setup
(1) Allocate queue for the storage of the frames to be transmitted.
(2.1) Define the attributes (non-data) of the generic Tx frame
(2.2) Insert the data into the generic Tx Frame
(2.3) Write the Frame attributes and the Frame data to the Tx Queue

Generic & Replay Transmitter Functions

4

Port Rx Setup
(1) *Assign Portmap ID to each Rx port*
(2) *Define Receiver mode (VL-Oriented or Chronological Receive)*

Global Receiver Functions

VL-Oriented Receiver Functions

VL-Oriented Setup
(1) *Define VL characteristics to look for (VL ID and range) and type of verification required*
(2) *Setup Rx UDP port (AFDX Comm or SAP port)*

Chronologic (Monitor) Setup
(1) *Define capture mode*
(2) *Create Monitor queue to hold captured data.*

Chronologic Receiver Functions

5

Start Tx
(1) *Send the AFDX frame cyclically or a certain number of times*
(2) *Setup to start immediately or wait for start time/strobe input*

Global Tranmsitter Functions

6

Start Rx
(1) *Start receiving the AFDX frames and reset/no reset status counters*

Global Receiver Functions

7

Global Transmitter Functions
UDP Port-Oriented Transmitter Functions
Generic & Replay Transmitter Functions

Global Receiver Functions
VL-Oriented Receiver Functions
Chronologic Receiver Functions

Retrieve Status
(1) *Tx Status*
(2) *Rx Status*
(3) *Retrieve Captured data*

8

Global Transmitter Functions
Global Receiver Functions
Library Administration Functions
VL-Oriented Receiver Functions
Chronologic Receiver Functions

Stop Tx/Rx - Shutdown
(1) *Stop Tx/Rx*
(2) *Free Resources*
(3) *Destroy Port (for VL-Oriented mode) or Delete Monitor Queue (For Chronologic Receive mode)*
(4) *Logout of each resource (Board and Port(s))*

4.1 Library Administration and System Programming

This section will discuss some of the typical scenarios a programmer would encounter that would require the use of the Library Administration, Target Independent Administration, and System, as listed in

Table 4-1, Table 4-2, and Table 4-3 respectively. These scenarios include:

a. Initialization, Login and Board Setup

b. Getting AIM Board Status and Configuration Information

c. Utilizing IRIG-B.

4.1.1 Initialization, Login, and Board Setup

This section will discuss the function calls required to support initialization and shutdown of the Application interface to your AIM board/module. Reference Section 1, for additional examples of the function calls described in this section.

The basic Library Administrative and System functions supporting initialization & shutdown include the following:

a. **FdxInit** - initializes API S/W Library.

1

Initialization
(1) API Library
(2) Board Login
(3) Port Login

FdxInit is the first function call to be issued. It returns the names of available servers at `px_ServerNames`. If `px_ServerNames = "local"`, the AFDX board is located where the API is running. If `px_ServerNames = "NULL"`, the end of the list has been reached.

Note:	*This version will only return "local" indicating that the AFDX board is located where the API is running. To connect to an AIM Network Server containing AFDX boards, use **FdxQueryServerConfig**.*

b. **FdxQueryServerConfig -** obtains the number of boards and their configuration. (Also allows for a connection to an AIM Network Server (ANS) containing AFDX boards.)

Once the Application interface has been initialized, the **FdxQueryServerConfig** function should be used to **obtain the configuration of the AFDX boards on the computer/server**.

FdxQueryServerConfig returns the list of resources using the structure TY-RESOURCE_LIST_ELEMENT. The resource information returned includes a resource ID, whether the resource is a board or port, board name. This information is used to login to the board and port(s) using the function **FdxLogin**.

> ***Note***: *This function can be used to connect to a remote ANS PC. If an ac_SrvName other than "local" is specified this function checks that PC to determine if a valid ANS is found. If a valid ANS is found on the specified PC this function connects to that server and returns a list of available resources of that PC.*

c. **FdxLogin -** establishes target communication for a specific resource.

For each resource in the system, the resource (**board and port**) must then be **logged into** using the server name "local" or the name of remote ANS, and the Resource ID returned by **FdxQueryServerConfig**. **FdxLogin** also requires as input information about the client, using the TY_FDX_CLIENT_INFO structure. MSWindows functions, `GetComputerName` and `GetUserName` can be used to obtain this information

> ***Note***: *For login to ports to be configured as redundant, only the first resource ID of the two physical ports can be used.*

d. **FdxCmdBoardControl -** used to control the global setting of the board.

> **2**
>
> **Board Setup**
> ***(1) single or redundant mod, bit rate & MAC/IP header verification register***
> *(2) IRIG time*

The purpose of this function is to setup the port configuration (single or redundant) and the port speed (10 Mbps or 100 Mbps (default) or auto negotiation). In addition, this function sets up the board to verify the MAC and/or IP header (1st 32 bytes of a frame) against either a customer defined value, an AFDX specific register, or Boeing specific register or the default register which is defined by the program-specific board type. The physical ports of the board are configurable in two different ways.

- **Single** - The port works as a single port. This means that the accessible port is represented by one physical port. In this mode it is possible to do traffic policing for this port.

- **Redundant** - The port works as a redundant port. This means that the accessible port is represented by two physical ports which are redundant. In this mode, the AFDX Redundancy Management Algorithm (RMA) is active and only the RMA passed frame is transmitted to the application. For login to that port only the first resource ID of the two physical ports can be used. A login to the second resource will cause an error. If the first of two ports is set to redundant mode, the second port will also be set to redundant mode. For the redundant port mode both physical ports are managed by one BIU.

...prior to termination:

e. **FdxLogout** - This function **closes the application interface** for the specified (**board and port**) resource and must be **called last** in an application program for all opened resources. After calling this function the handle is invalid and it is not possible to use it for further function calls.

8

Stop Tx/Rx - Shutdown

(1) Stop Tx/Rx
(2) Free Resources
(3) Destroy Port (for VL-Oriented mode) or Delete Monitor Queue (For Chronologic Receive mode)
(4) Logout of each resource (Board and Port(s))

The following code demonstrates the **Initialization**, **Board Setup**, **Login** and **Logout** Functions.

The code first searches through the list of servers found using **FdxInit**, and when "NULL" is found, the end of the list has been reached. If "local" was in the list, an AFDX board has been found on the local computer/server. This name, "local", is then used in the **FdxQueryServerConfig** function.

The code continues to search in the Resource List returned by **FdxQueryServerConfig** until "NULL" is found indicating the end of the list has been reached. If the resource is a board, the board is logged into, and the board handle is obtained. If the resource is a port, then the port is logged into and the port handle is returned. These handles will be used for all future API function calls. .

The code configures the ports on the board to **single** (i.e., not redundant).

```
AiChar                      ServerName[128] = "local";
bool                        bRetSuccess = false;
bool                        bFoundLocalServer = false;

TY_SERVER_LIST *            px_ServerList = NULL;
TY_SERVER_LIST *            px_TmpServer;

TY_RESOURCE_LIST_ELEMENT *  pRLE = NULL;
TY_RESOURCE_LIST_ELEMENT *  pRLEHead = NULL;
TY_FDX_CLIENT_INFO          x_ClientInfo;

AiUInt32 ul_HandleBoard = 0, ul_HandlePort1 = 0, ul_HandlePort2 = 0;

printf("\r\n FdxInit() \r\n");
if(FdxInit(&px_ServerList) != FDX_OK)
{
   printf("FdxInit Failed!!!\n");
```

```
   // free the server-list
   if (px_ServerList != NULL)
   {
      FdxCmdFreeMemory(px_ServerList, px_ServerList->ul_StructId);
   }
   return(bRetSuccess);
}
// search the server-list for local server
px_TmpServer = px_ServerList;
while ((px_TmpServer != NULL) && (!bFoundLocalServer))
{
   if (stricmp(px_TmpServer->auc_ServerName, "local") == 0)
   {
      bFoundLocalServer = true;
   }
   else
   {
      px_TmpServer = px_TmpServer->px_Next;
   }
}

if (bFoundLocalServer)
{  // ok, we found a local server
   // lets query the configuration of this server
   if (FdxQueryServerConfig("local", &pRLEHead) == FDX_OK)
   {
      pRLE = pRLEHead;
      while (pRLE != NULL)
      {  //--- login to resources
         switch(pRLE->ul_ResourceType)
         {
         case RESOURCETYPE_BOARD:
            // Board Login
            if (ul_HandleBoard == 0)
            {
               if (FdxLogin("local", &x_ClientInfo, pRLE->ul_ResourceID, 0,
                                     &ul_HandleBoard) != FDX_OK)
               {
                  ul_HandleBoard = 0;
                  printf("Board Login Failure!!!\n");
               }
            break;

         case RESOURCETYPE_PORT:
            // Port Login
            if (ul_HandlePort1 == 0)
            {
               if (FdxLogin("local", &x_ClientInfo, pRLE->ul_ResourceID, 0,
                                     &ul_HandlePort1) != FDX_OK)
               {
                  ul_HandlePort1 = 0;
                  printf("Port 1 Login Failure!!!\n");
               }
            }
            else
               if (ul_HandlePort2 == 0)
               {
                  if (FdxLogin("local", &x_ClientInfo, pRLE->ul_ResourceID, 0,
                                        &ul_HandlePort2) != FDX_OK)
                  {
                     ul_HandlePort2 = 0;
                     printf("Port 2 Login Failure!!!\n");
                  }
               }
```

```
            break;
         }
         pRLE = pRLE->px_Next;
      }
```
This completes the Initialization and Login of the board.

Now let's perform Board Setup for single (not redundant mode), and Verification mode set to compare agains the AFDX program specific verification register.

```
// perform board setup
int i;
TY_FDX_BOARD_CTRL_IN    x_BoardCtrlIn;
TY_FDX_BOARD_CTRL_OUT   x_BoardCtrlOut;

memset( &x_BoardCtrlIn, 0, sizeof(x_BoardCtrlIn) );
memset( &x_BoardCtrlOut, 0, sizeof(x_BoardCtrlOut) );

if (ul_HandleBoard > 0)
{
   //--- init input structure
   for (i=0; i<FDX_MAX_BOARD_PORTS; i++)
   {
      x_BoardCtrlIn.aul_PortConfig[i] = FDX_SINGLE;
      x_BoardCtrlIn.aul_PortSpeed[i] = FDX_100MBIT; /* not selectable for GNET */
      x_BoardCtrlIn.aul_ExpertMode[i] = FDX_EXPERT_MODE;
   }
   x_BoardCtrlIn.ul_RxVeriMode = FDX_BOARD_VERIFICATION_TYPE_AFDX;
   //--- reset board
   if (FDX_OK != (FdxCmdBoardControl(ul_HandleBoard, FDX_WRITE, &x_BoardCtrlIn,
                                       &x_BoardCtrlOut)))
   {
      printf("Board Reset Failure!!!\n");
   }
   else
   {
      printf("Board Initialized\n");
   }
}
```
This completes Board Setup.

..... and prior to termination, Logout of each resource (board and port).

```
// Close Device
if (ul_HandleBoard != 0)
{
   if (FDX_ERR == FdxLogout(ul_HandleBoard)) {
      printf("FdxLogout Board Error.\n");
   }
   else {
      printf("FdxLogout Board done.\n");
   }
}
if (ul_HandlePort1 != 0)
{
   if (FDX_ERR == FdxLogout(ul_HandlePort1)) {
      printf("FdxLogout Error 1.\n");
   }
   else
   {
      printf("FdxLogout Port1 done.\n");
   }
}
if (ul_HandlePort2 != 0)
```

```
{
   if (FDX_ERR == FdxLogout(ul_HandlePort2))
   {
      printf("FdxLogout Error 2");
   }
   else
   {
      printf("FdxLogout Port2 done.\n");
   }
}
```

Note: ***In addition to Board and Port Logout, Monitor Queue, and or Tx/Rx UDP Port must also be deleted/destroyed if previously created prior to termination (****FdxCmdMonQueueControl****,*** ***FdxCmdTxUDPDestroyPort****,*** ***and*** ***FdxCmdRxUDPDestroyPort****).***

Note: ***Before Programm exit (close of library) call FdxExit() to free resource list.***

```
/* free the resource list, the device list and the server list */
    if (FDX_OK != FdxExit())
        printf("\r\n FdxExit() FAIL");
```

4.1.2 Getting AIM Board Status and Configuration Information

Once you have initialized and opened the connection to the AIM board as described in the previous section, you can obtain the status of the configuration of the board and the software versions contained on your AIM board. The system functions that perform this status are as follows:

a. **FdxQueryResource** - Obtains information about the board or port resource, including board name, serial number, physical ports available, etc., and which clients are using this resource.

b. **FdxReadBSPVersion** - Returns the version number of all AIM board software package components

c. **FdxCmdBITETransfer** - Performs some transfer tests using available port resources of one FDX board. This function will determine the number of ports on the board. If only two ports, it will test them against each other. If four ports are used, Port 1 and Port 2 will be tested against each other and Port 3 and Port 4 will be tested against each other

   Port 1 and Port 2 must be connected with a Loop-Back cable (crossover), if available Port 3 and Port 4 must be connected with a Loop-Back cable (crossover).

   > ***Note**: This test should be performed prior to login. Only "local" operation of the resources supported.*

d. **FdxCmdBoardControl** - this function can also be used to read back the configuration of the board including: port configuration (single or redundant), port speed, connection/link status, and size of free global and shared memory.

4.1.3 Utilizing IRIG-B

The API S/W Library provides one System function call to setup/read/write IRIG-B time, and three Target Independent Administration Functions to convert and calculate IRIG-B time including:

2

Board Setup
(1) single or redundant mode & bit rate
(2) IRIG time

a. **FdxCmdIrigTimeControl**- Sets/writes the IRIG-B time on the on-board IRIG timecode encoder, or allows the IRIG-B input to be received from an external source.

b. **FdxFwIrig2StructIrig** - converts the IRIG-B time from Firmware format to 32-bit values for each hour, minute, second, day, millisecond and microsecond value.

   - This function may be required to reformat the IRIG time received in the Received UDP/Monitor Buffers via functions **FdxCmdRxUDPRead** and **FdxCmdMonQueueRead** (See Section 4.3.1.1 for coding example.)

c. **FdxStructIrig2FwIrig** - converts the IRIG-B time from the structure format (provided with **FdxFwIrig2StructIrig**) to the Firmware format

d. **FdxAddIrigStructIrig** and **FdxSubIrigStructIrig** - adds or subtracts two Structured IRIG values.

The following is an example of the **FdxCmdIrigTimeControl** and **FdxAddIrigStructIrig.** Notice the declaration of the `x_IrigTimeA, B and C` as type `TY_FDX_IRIG_TIME`. `TY_FDX_IRIG_TIME` is defined in the `AiFdx_def.h` header file.

Note: ***To obtain an accurate time stamp value you should delay the immediate reading of the IRIG time.***

Note: IRIG time starts with "DAY one" (First of January) not with "DAY zero".

```
AiInt32 r_RetVal;
TY_FDX_IRIG_TIME x_IrigTimeA, x_IrigTimeB, x_IrigTimeC;
AiUInt32 ul_Mode;
time_t clock;              /* Posix */
struct tm *pxSystemTime;   /* Posix */

/* Read Irig Time (A) */
if (FDX_OK != (r_RetVal = FdxCmdIrigTimeControl( ul_Handle, FDX_IRIG_READ,
&x_IrigTimeA, &ul_Mode)))
{
   printf("\r\nFdxCmdIrigTimeControl() failed.");
}

/* Set Irig Time to Day 001:00:00:00.000 */
x_IrigTimeB.ul_Day      = 1;
```

```
x_IrigTimeB.ul_Hour     = 0;
x_IrigTimeB.ul_Min      = 0;
x_IrigTimeB.ul_Second   = 0;
x_IrigTimeB.ul_MilliSec = 0;
x_IrigTimeB.ul_MicroSec = 0;
x_IrigTimeB.ul_NanoSec  = 0;
x_IrigTimeB.ul_Info     = 0;

if (FDX_OK != FdxCmdIrigTimeControl( ul_Handle, FDX_IRIG_WRITE, &x_IrigTimeB,
&ul_Mode))
   printf("\r\nFdxCmdIrigTimeControl() failed.");

AIM_WAIT(6000);

/* And Read it Back after 6 seconds (B) */
if (FDX_OK != FdxCmdIrigTimeControl( ul_Handle, FDX_IRIG_READ, &x_IrigTimeB,
&ul_Mode))
   printf("\r\nFdxCmdIrigTimeControl() failed.");

/* Add Irig A and Irig B */
x_IrigTimeC = FdxAddIrigStructIrig( &x_IrigTimeA, &x_IrigTimeB);

/* Set Irig Time To Host Time */
clock = time((time_t*)NULL);
pxSystemTime = localtime( &clock );

x_IrigTimeA.ul_Day      = pxSystemTime->tm_yday +1;
x_IrigTimeA.ul_Hour     = pxSystemTime->tm_hour;
x_IrigTimeA.ul_Min      = pxSystemTime->tm_min;
x_IrigTimeA.ul_Second   = pxSystemTime->tm_sec;
x_IrigTimeA.ul_MilliSec = 0;
x_IrigTimeA.ul_MicroSec = 0;
x_IrigTimeA.l_Sign      = 0;

if (FDX_OK != FdxCmdIrigTimeControl( ul_Handle, FDX_IRIG_WRITE, &x_IrigTimeA,
&ul_Mode))
   printf("\r\nFdxCmdIrigTimeControl() failed.");

AIM_WAIT(6000);

/* Set source to external */
if (FDX_OK != FdxCmdIrigTimeControl( ul_Handle, FDX_IRIG_EXTERN, NULL, &ul_Mode))
   printf("\r\nFdxCmdIrigTimeControl() failed.");

/* Set source to intern */
if (FDX_OK != FdxCmdIrigTimeControl( ul_Handle, FDX_IRIG_INTERN, &x_IrigTimeB,
                                                 &ul_Mode))
   printf("\r\nFdxCmdIrigTimeControl() failed.");
```

4.1.4 Interrupt Handling

If setup by the user, interrupts can be generated by the Receiver functions. (Interrupts for Transmit operations are planned for future enhancements.) The type of interrupts available and the associated setup function required to setup the interrupt is defined in Table 4-6. Figure 4-2 shows the basic steps involved in setting up and creating an application utilizing interrupts.

Table 4-6 Available Interrupt Types and Related Function Call

Transmitter	Receiver
FdxCmdTxQueueWrite* Interrupt when BIU is instructed to stop or synchornize. *In Future API	**FdxCmdRxVLControlEx** Interrupt on: - Frame Reception for user-specified VL - Frame Error for a user-specified VL - Buffer Full/ Half Full/Quarter Full for a user-specified VL **FdxCmdMonTCBSetup** Interrupt on: - Trigger Control Block event is true.

The **FdxInstIntHandler** and **FdxDelIntHandler** function calls are used to setup and remove the interrupt setting for a specified BIU. These functions are discussed below and sample code is provided demonstrating Interrupt setup using these functions. The additional software setup required for the BC, RT, BM, and/or Replay function call(s) is discussed in the associated section of this document:

The functions available to setup interrupts and interrupt handler execution include the following Library Administration functions:

a. **FdxInstIntHandler** - Provides a pointer to the interrupt handler function. The following code installs an Interrupt Handler function named `userInterruptFunction` to handle interrupts generated by the Rx Monitor `(FDX_INT_RT)`.

```
//Install Interrupt Handler function to handle Receive monitor interrupts
FdxInstIntHandler(ul_Handle, FDX_INT_RX, userInterruptFunction );
```

- The Interrupt Handler function is a function that you create to perform application specific processing based on the type of interrupt received.

- Only one interrupt handler is required, however, you can also create one interrupt handler for each type of interrupt. (Currently only Receive interrupts are available for processing.)

- Interrupt Handler function input parameters must follow a pre-defined format as defined in the **FdxInstIntHandler** function call in the **Reference Manual AFDX/ ARINC-664**:

```
void userInterruptFunction(AiUInt8 bModule, AiUInt8 uc_Port,
      AiUInt8 uc_Type,TY_FDX_INTR_LOGLIST_ENTRY x_Info )
```

b. **FdxDelIntHandler** - Removes the pointer interface to the interrupt handler function. This function should be called prior to the module close (**FdxLogout**). The following code uninstalls an Interrupt Handler function named `userInterruptFunction` to handle interrupts generated by the Rx Monitor `(FDX_INT_RT).`

//Uninstall the Receive interrupt handler function(s)

```
FdxDelIntHandler( ul_Handle, FDX_INT_RX);
```

Further definition and examples of these interrupt scenarios can be found in the afdx_Sample.exe (included in the BSP with sources).

Figure 4-2 Interrupt Setup Process

Decide which type of interrupt is required for your application

1

Create an **Interrupt Handler** application to process interrupt/data when interrupt occurs.

2

Include function call **FdxInstIntHandler** to intialize the BIU with a pointer to your **Interrupt Handler**.

3

Setup the Transmitter/Receiver function(s) interrupt(s) as required by your application.

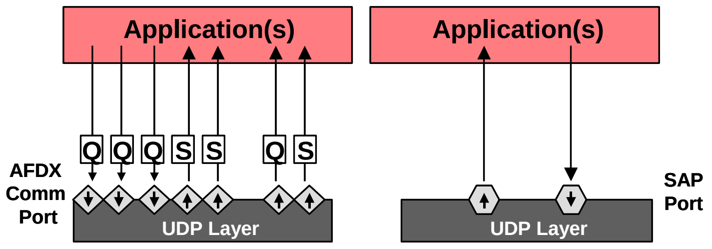
4

Delete the the host-to-AIM board interrupt setup prior to the end of your application using **FdxDelIntHandler**.

4.2 Transmitter Programming

The AFDX transmit port can be configured in one of three **AFDX-Traffic Generation** modes of data transmission as listed below:

a. **UDP Port-Oriented Simulation** - This mode simulates the AFDX Comm ports (defined by ARINC-653) and SAP ports. AFDX Comm Ports communicate via a static "connection" i.e., the IP/UDP Source/Destination addresses are contained in the AFDX frame header are fixed. SAP ports, however, are "connectionless" i.e., the E/S application can dynamically determine the destination address (IP address and UDP port number) for messages transmitted, and messages can be received from multiple sources.



An AFDX Comm port provide two different types of services:

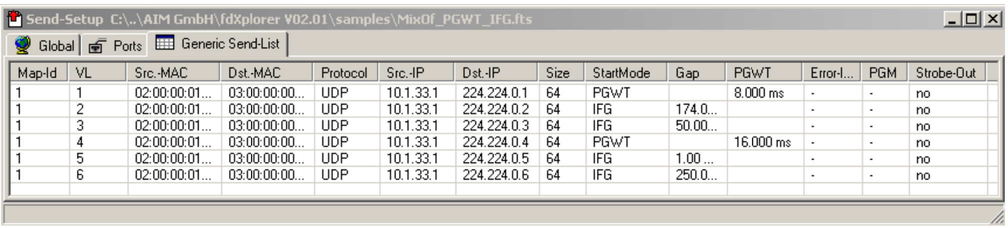
- **Queuing service** - AFDX messages are sent over several AFDX frames (fragmentation by IP layer), no data is lost or overwritten.

- **Sampling service** - AFDX messages are sent in 1 frame, data may be lost or overwritten.

The end-systems, VLs, and partitions are represented by the IP-Addresses and communication-end points are described by the AFDX Comm UDP-Port.

SAP ports can also transmit and receive AFDX messages that are sent over one or more AFDX frames, however, the protocol for that communication is not determined by ARINC 653.

b. **Generic Transmit Operation** - This mode provides maximum flexibility and is based on a frame based transmission sequence. Each frame provides information about the relative timing between the frames, error injection, payload-generation modes, transmission skew in redundant operation mode and/or special events like a digital output strobe-signal. For high-throughput, special payload-generation modes can be used, so the hardware takes parts of the frame-data from static send-fields. Because all frames must be pre-buffered on the hardware, the number of frames is limited to the board-resources.


Send-Setup C:\..\AIM GmbH\fdXplorer V02.01\samples\MixOf_PGWT_IFG.fts

Global | Ports | Generic Send-List

Map-Id	VL	Src.-MAC	Dst.-MAC	Protocol	Src.-IP	Dst.-IP	Size	StartMode	Gap	PGWT	Error-I...	PGM	Strobe-Out
1	1	02:00:00:01...	03:00:00:00...	UDP	10.1.33.1	224.224.0.1	64	PGWT		8.000 ms	-	-	no
1	2	02:00:00:01...	03:00:00:00...	UDP	10.1.33.1	224.224.0.2	64	IFG	174.0...		-	-	no
1	3	02:00:00:01...	03:00:00:00...	UDP	10.1.33.1	224.224.0.3	64	IFG	50.00...		-	-	no
1	4	02:00:00:01...	03:00:00:00...	UDP	10.1.33.1	224.224.0.4	64	PGWT		16.000 ms	-	-	no
1	5	02:00:00:01...	03:00:00:00...	UDP	10.1.33.1	224.224.0.5	64	IFG	1.00 ...		-	-	no
1	6	02:00:00:01...	03:00:00:00...	UDP	10.1.33.1	224.224.0.6	64	IFG	250.0...		-	-	no

c. **Replay Operation**

- Physical Re-Transmission of pre-recorded network traffic (or real-time playback of data captured while in Record mode.)

Table 1-1 defines the key features and differences between the UDP-Port Oriented transmission mode and the Generic Transmission mode.

If your application requires the generation of AFDX traffic, this section will provide you with the understanding of the Transmitter programming functions required for use within your application. Regardless of the transmission mode you select, Global Transmitter functions must be performed first. This section will describe Transmitter Functions as follows:

a. Global Transmitter Functions

- Port Initialization & Tx Mode Setup
- Transmission Control
- Strobe Input/Output Usage (optional)
- Global Transmit Status

b. UDP Port-Oriented Simulation

- Creating the Virtual Link and Sub VL UDP Port
- Writing messages to the UDP Port
- Individual UDP Port Error Injection, Skew
- Individual UDP Port Status and Enable/Disable

c. Generic Transmit

- Allocating a Transmit Queue
- Defining Frames and Writing to the Transmit Queue
- Generic Transmit Queue Status

d. Replay

- Allocating a Transmit Queue
- Writing a Replay file to the Transmit Queue

- Replay Transmit Queue Status.

4.2.1 Global Transmitter Functions

Global Transmitter functions are the functions that apply to all modes of operation (UDP Port-Oriented, Generic Transmit or Replay). The major functions of the Global Transmitter functions include:

a. Port Initialization & Tx Mode Setup

b. Transmission Control

c. Strobe Input/Output Usage (optional)

d. Global Transmit Status

These Global Transmitter function are described in the following sections.

4.2.1.1 Port Initialization and Tx Mode Setup

After the Board setup is complete, as defined in Section 4.1, an individual Transmit Port can be initialized and the mode can be configured for either UDP Port-Oriented, Generic Transmit or Replay Mode using the functions listed below. Once the mode has been configured, the user can then program the port transmission protocol and data characteristics as defined in the section applicable to the mode (Section 4.2.1- 1.1.1).

3

Port Tx Setup
(1) Assign Portmap ID to each Tx port
(2) Define Transmitter mode (UDP Port-Oriented or Generic Transmit)

a. **FdxCmdTxPortInit -** will perform **initialization** and reset of the Transmit port global characteristics. This must be the first Transmitter function call issued. **The user must assign a Portmap ID to each Tx port**. This Portmap ID is a virtual ID assigned to the physical Port. State after initialization includes:

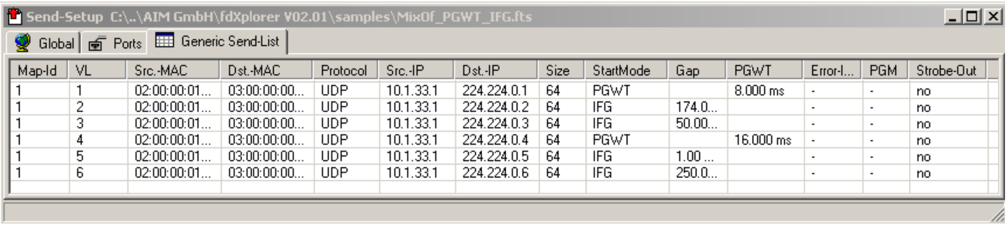
- **No Transmit Queues defined**- this is referring to the queue assigned to a port with the **FdxCmdTxQueueCreate** function call (for Generic & Replay Transmit mode).

- **No VL created, no UDP port created** - this is referring to the VL and port created with the **FdxCmdTxCreateVL** and **FdxCmdTxUDPCreatePort** functions (for UDP Port-Oriented Simulation mode)

b. **FdxCmdTxModeControl** - provides configuration for the three transmit modes including:

- **UDP Port-Oriented Simulation** - This mode simulates the AFDX Comm and SAP UDP ports as defined by the AFDX End System Detailed Functional Specification.

- **Generic Transmit Operation** - This mode provides maximum flexibility and is based on a frame based transmission sequence. Each frame provides information about the relative timing between the frames, error injection, payload-generation modes, transmission skew in redundant operation mode and/or special events like a digital output strobe-signal. For high-throughput, special payload-generation modes can be used, so the hardware takes parts of the frame-data from static send-fields. Because all frames must be pre-buffered on the hardware, the number of frames is limited to the board-resources.



- **Replay Operation -** Physical Re- Transmission of recorded network traffic or (or real-time playback of data captured while in Record mode.)

The following code example uses API S/W Library constants to initialize one port using a Portmap equal to 1 and configures the mode to Generic Transmit. The port handle was previously obtained using **FdxLogin**.

```
    TY_FDX_PORT_INIT_IN       x_PortInitIn;
    TY_FDX_PORT_INIT_OUT      x_PortInitOut;
    TY_FDX_TX_MODE_CTRL       x_TxModeCtrl;

   //--- Initialization
    x_PortInitIn.ul_PortMap = 1;
    if (FDX_OK != (FdxCmdTxPortInit(ul_HandlePort, &x_PortInitIn, &x_PortInitOut)))
   {
     printf("Port Reset failure!!!\n");
   }
   else
   {
     printf("Port Transmitter Initialized\n");
   }

   //--- mode control -> Set TX port to Generic mode
   x_TxModeCtrl.ul_TransmitMode = FDX_TX_GENERIC;
   if (FDX_OK != (FdxCmdTxModeControl(ul_HandlePort, &x_TxModeCtrl)))
   {
      printf("Port Mode Control Failure!!!\n");
   }
   else
   {
      printf("Port set to Generic Transmit mode\n");
   }
```

4.2.1.2 Transmission Control

After all protocol and data characteristics of the port have been programmed (as defined in Section 4.2.2 - 4.2.4), the method used to start/stop transmission of the data should be defined. There are two global functions providing Tx Control:

5

Start Tx

(1) Send the AFDX frame cyclically or a certain number of times

(2) Setup to start immediately or wait for start time/strobe input

a. **FdxCmdTxControl** - defines how and when to start transmission for a **Physical port** and the length of transmission including:

- **Starting/Stopping** the transmitter - provides for:

  1. Starting/stopping the transmitter on command
  2. Starting the transmitter based on an external strobe input
  3. Starting the transmitter based on a specified start time

- **Cyclic or user-specified number of transmissions** for the frames defined in the Transmit Queue when in Generic Transmit mode. (In Replay the number of frames transmitted depends on the size of the Replay file. In UDP Port-Oriented Simulation, the transmission rate for an AFDX Comm port is defined by **FdxCmdTxUDPCreatePort** (for Sampling port), and the transmission is initiated when data is written to the port using **FdxCmdTxUDPWrite** or **FdxCmdTxUDPBlockWrite** (for a Queuing port). For a SAP port, transmission is initiated with **FdxCmdTxSAPWrite** or **FdxCmdTxSAPBlockWrite**.)

b. **FdxCmdTxVLControl** - provides the user with the option to enable/disable an individual VL as defined in either the UDP Port-Oriented Transmit mode or Generic Transmit mode. The default condition of a VL is enabled, therefore, this function is not required unless you choose to disable individual VLs.

> ***Note:*** *Lower level Sub VL (UDP Port) enable/disable control (**FdxCmdTxUDPControl**) is available when in UDP Port-Oriented Transmit mode as discussed in Section 4.2.2.3.*

The following code configures a port to transmit AFDX frames cyclically (`x_TxControl.ul_count = 0` (Generic Transmit mode only)) and to start immediately upon this command.

```
TY_FDX_TX_CTRL x_TxControl;

x_TxControl.ul_Count = 0;  //0 value indicates cyclic transmission
x_TxControl.e_StartMode = FDX_START;

if (ul_HandlePort != NULL)
{
      if (FDX_OK != (FdxCmdTxControl( ul_HandlePort, &x_TxControl))) {
```

```
              printf("Failure to start transmitter\n");
        }
        else {
              printf("Transmitter started\n");
        }
}
```

4.2.1.3 Strobe Input/Output Usage

As a programming option, the strobe input/output signals for the Transmitter ports can be used in various ways to control transmission of AFDX frames or to indicate the occurrence of a specific frame transmission as shown in Table 4-7. This table indicates the functions required to utilize the Strobe external input signals to control transmission of AFDX frames and in which Transmit modes these functions are applicable.

Every two AFDX ports on the AIM AFDX board utilize the 15 pin D-Sub connector for Strobe Input and Outputs. The layout for the pin is shown in Figure 4-3. Each physical port can be assigned one Strobe input line and one Strobe output line.

For GNET-2 the Strobe and IRIG Connecteor Pinout is as described in Figure 4-3. For GNET-4 the Strobe and IRIG Connecteor is available on a separate break out module. The pinout of the HDSUB 15 is identical to Figure 4-3 except the IRIG pins. On the APX-GNET-BOP pin 1 and 2 are internally connected. The IRIG Input function on pin 1 is always active. If the module shall be configured to act as IRIG- Time source (IRIGout activated), then a jumper has to be installed on the APX-GNET-4 module. Please refer also to according hardware manuals.

Table 4-7 Strobe Input/Output Transmitter Functions

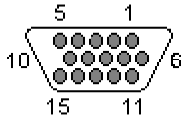
Function Type	Applicable mode		API Function	Strobe- In Function	Strobe-Out Function
	UDP Port-Oriented	Generic Transmit			
Global Transmitter	X	X	**FdxCmdTxTrgLineControl**	Defines the strobe Input/Output lines to be used for the receive ports	
Global Transmitter	X	X	**FdxCmdTxControl**	Strobe-in to Start transmitter*	
Generic Transmit Operation		X	**FdxCmdTxQueueWrite**	Strobe-in to Start transmission of this frame *	Strobe-out on transmission of this frame.

* *In redundant operation mode, the strobe input for port A shall be the same strobe input resource for Port B.*

Figure 4-3 Strobe and IRIG Connector Pinout

Pin No.	Type	Signal
1	I	IRIG Input
2	O	IRIG Output
3		Reserved.
4		Reserved.
5	Power	GND
6		Reserved.
7	I	Strobe Input1
8	O	Strobe Output1
9	I	Strobe Input2
10	O	Strobe Output2
11	I	Strobe Input3
12	O	Strobe Output3
13	I	Strobe Input4
14	O	Strobe Output4
15		Reserved.

15 Pin D-Sub:



4.2.1.4 Global Transmit Status

7

Retrieve Status
(1) Tx Status
(2) Rx Status
(3) Retrieve Captured data

The Global Transmitter function group includes one function call, **FdxCmdTxStatus**, that provides you with the capability to obtain the following status:

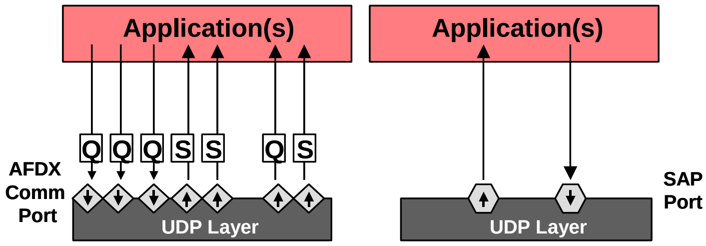
a. **Transmitter status** - Stopped/Running/Error

b. **Frames Transmitted** - for primary and redundant port (if programmed in Redundant mode using the **FdxCmdBoardControl** Board Setup function)

c. **Mode** - Generic Transmit, UDP Port-Oriented, or Replay.

Additional lower level status can be obtained when in UDP Port-Oriented Transmit mode by using the function **FdxCmdTxUDPGetStatus**, and, when in the Generic or Replay Transmit mode, by using **FdxCmdTxQueueStatus** as described in Sections 4.2.2.4, 4.2.3.3 and 4.2.4.3 respectively.

4.2.2 UDP Port-Oriented Simulation Mode



When operating in UDP Port-Oriented Simulation mode AFDX Comm ports (connection oriented) and SAP ports (connectionless) are simulated.

The functions described in this section should be used after the port has been configured for UDP Port-Oriented Simulation using the function **FdxCmdTxModeControl,** as described in Section 4.2.1.1. Setting up the port when in UDP Port-Oriented Simulation mode consists of the following main functions:

> **UDP Port-Oriented Setup**
> *(1) Define the VL & Sub VL characteristics*
> *(2) Write UDP port messages created to Tx port*

a. Creating the Virtual Link and Sub VL (UDP Port)

b. Writing messages to UDP Port

c. Individual UDP Port Error Injection, Skew and Enable/Disable (optional)

d. Individual UDP Port Status

e. Changing the source of a UDP while transmit is enabled.

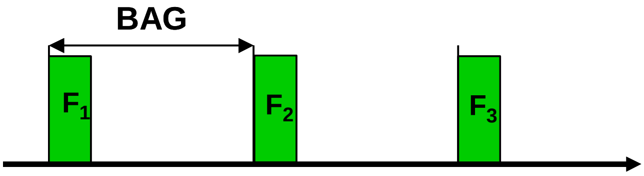
4.2.2.1 Creating the Virtual Link and Sub VL

The following functions are associated with VL and Sub VL (UDP Port) creation for a defined port.

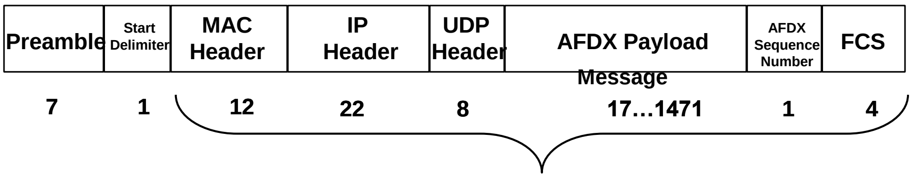
a. **FdxCmdTxCreateVL** or **FdxCmdTxCreateHiResVL** - these functions will define the characteristics associated with the **VL** as listed in the example below. Only one function is needed. The **FdxCmdTxCreateHiResVL** provides a higher resolutionBAG as defined below. These characteristics will apply to all SubVLs associated with the VL.

FdxCmdTxCreateVL					
VL ID	BAG	Max Frame Length	Frame Buffer Size	MAC Source Address	Network Control (applicable to Redundant mode only)
0	1 ms	147 bytes	0	02:00:00:01:21:20	A+B at once

- For function **FdxCmdTxCreateVL**, **BAG** values are in milliseconds: 1, 2, 4, 8, 16, 32, 64, 128. For function **FdxCmdTxCreateHiResVL**, higher resolution BAG values can be specified anywhere within the range of 800 µsec to 128000 µsec.



- **Max Frame Length** includes all fields shown below (except the Preamble and Start Delimiter):



Frame-Size

- **Frame Buffer Size** - Allows the user to specify the size of the VL Buffer (needed when using function **FdxCmdTxVLWrite** or **FdxCmdTxVLWriteEx.** If set to 0, the target software computes the length of this buffer.)

- **Network Control** - Allows the user to specify how frames are transmitted when in **redundant mode**. i.e., whether Port A/B frames are transmitted skewed, in-sync or only Port A or only Port B.

b. There are two sets of functions associated with creation of the UDP Tx port: AFDX Comm port functions and SAP functions as described below:

For AFDX Comm Port (Sampling or Queuing Connection-oriented port):

- **FdxCmdTxUDPCreatePort** - this function will define the characteristics associated with the **Sub VL** AFDX Comm Sampling/Queuing port associated with a VL as listed in the example below. These characteristics will apply to all SubVLs associated with the VL. Up to four Sub VLs can be defined for one VL. A **UDP Handle** to this UDP Port Sub VL is returned when this command is issued successfully. The handle is used for all further communication/control of the Sub VL (AFDX Comm UDP Port). An AFDX Comm port is connection-oriented, therefore, the entire address quintuplet is specified for the create port function to define the point-to-point connection.

FdxCmdTxUDPCreatePort								
Sub VL ID	Type (Sampling or Queuing)	Sampling Rate	Source IP	Destination IP	Source UDP	Destination UDP	Maximum Message Size	Number of buffered messages
1	Sampling	10 ms	10.1.33.1	224.224.0.0	1	1	100 bytes	1
2	Sampling	10 ms	10.1.33.1	224.224.0.0	2	2	90 bytes	1
3	Sampling	10 ms	10.1.33.1	224.224.0.0	3	3	80 bytes	1
4	Queuing		10.1.33.1	224.224.0.0	4	4	max. 8000 bytes	3

Sampling Rate - values start with 1 milliseconds in multiples of 1. This is the rate at which the AFDX frame will be transmitted cyclically at a Sampling Port.

Maximum Message Size - the size of the AFDX Payload Message. The size is fixed for Sampling ports. For queuing ports, it is the maximum size of the complete message to be fragmented and transmitted out the queuing port.

Number of Buffered Messages - for a Sampling Port this is always 1. For a Queuing Port, this indicates the number of complete messages (with size = to max Message size) that may be buffered for transmission. The default value is 2.

The following code creates one VL (VL 33) and two Sub VLs (UDP port 1 and UDP port 2) both configured as Sampling ports.

```
//--- create VL, define communication parameters for VL 33 on Port
x_TxCreateVL.ul_VlId             = 33;          /* VL */
x_TxCreateVL.ul_SubVls           = 1;           /* Number of Sub VLs */
x_TxCreateVL.ul_Bag              = 32;          /* BAG [ms] */
x_TxCreateVL.ul_MaxFrameLength   = 1000;        /* Maximum Frame Length [bytes] */
x_TxCreateVL.ul_FrameBufferSize  = 0;           /* Default Frame Buffer Size */
x_TxCreateVL.ul_MACSourceLSLW    = 0x00089aC0;  /* MAC Source */
x_TxCreateVL.ul_MACSourceMSLW    = 0x00000200;  /* MAC Source */
x_TxCreateVL.ul_NetSelect        = FDX_TX_FRAME_BOTH;
x_TxCreateVL.ul_Skew             = 0;
if (FDX_OK != ( FdxCmdTxCreateVL (ul_Handle,&x_TxCreateVL))) {
   printf("VL Creation on Port failed!!!\n");
}
else {
   printf("VL Created 33 \n");
}

//--- create udp-port 1 for write on Port
x_TxUDPDescription.ul_PortType = FDX_UDP_SAMPLING;   /* Sampling Port */
x_TxUDPDescription.x_Quint.ul_UdpSrc     = 23;
x_TxUDPDescription.x_Quint.ul_UdpDst     = 24;
x_TxUDPDescription.x_Quint.ul_VlId       = 33;
x_TxUDPDescription.x_Quint.ul_IpSrc      = ul_GenerateIp("10.1.33.1");
x_TxUDPDescription.x_Quint.ul_IpDst      = ul_GenerateIp("224.224.0.33");
x_TxUDPDescription.ul_SubVlId            = 1;
x_TxUDPDescription.ul_UdpMaxMessageSize = 200;
x_TxUDPDescription.ul_UdpNumBufMessages = 1;        /* 0=default */
x_TxUDPDescription.ul_UdpSamplingRate   = 100;      /* [ms] */
```

```
if (FDX_OK != ( FdxCmdTxUDPCreatePort (ul_Handle,&x_TxUDPDescription,
                                        &ul_Udp1Handle)))
 {
   printf("UDP Port Creation Failure on Port!!!\n");
 }
 else
 {
   printf("Tx UDP Port 1 Created.);
 }

//--- create udp-port 2 for write on Port
x_TxUDPDescription.ul_PortType = FDX_UDP_SAMPLING;  /* Sampling Port */
x_TxUDPDescription.x_Quint.ul_UdpSrc    = 34;
x_TxUDPDescription.x_Quint.ul_UdpDst    = 42;
x_TxUDPDescription.x_Quint.ul_VlId      = 33;
x_TxUDPDescription.x_Quint.ul_IpSrc     = ul_GenerateIp("10.1.33.1");
x_TxUDPDescription.x_Quint.ul_IpDst     = ul_GenerateIp("224.224.0.33");
x_TxUDPDescription.ul_SubVlId           = 1;
x_TxUDPDescription.ul_UdpMaxMessageSize = 300;
x_TxUDPDescription.ul_UdpNumBufMessages = 1;        /* 0=default */
x_TxUDPDescription.ul_UdpSamplingRate   = 50;       /* [ms] */


if (FDX_OK != ( FdxCmdTxUDPCreatePort (ul_Handle,&x_TxUDPDescription,
                                        &ul_Udp2Handle)))
 {
   printf("UDP Port Creation Failure on Port!!!\n");
 }
 else
 {
   printf("Tx UDP Port 2 Created.);
 }
```

For SAP Port (Connectionless-oriented port):

- **FdxCmdTxSAPCreatePort** - this function will define the characteristics associated with the **Sub VL** SAP port. Up to four Sub VLs can be defined for one VL. A **UDP Handle** to this UDP Port Sub VL is returned when this command is issued successfully. The handle is used for all further communication/control of the Sub VL (UDP Port). The characteristics for the SAP Tx port include only those listed below. Remember, for a SAP port - since the port is "connectionless" it can transmit to multiple E/S's and the destination is determined at the time of transmission, therefore, only the Source IP/UDP addresses are required.

FdxCmdTxSAPCreatePort				
Sub VL ID	Source IP	Source UDP	Maximum Message Size	Number of buffered messages
1	10.1.33.1	1	max. 8000 bytes	3
2	10.1.33.1	2	max. 8000 bytes	2
3	10.1.33.1	3	max. 8000 bytes	4
4	10.1.33.1	4	max. 8000 bytes	3

Maximum Message Size - the size of the AFDX Payload Message. For SAP ports, it is the maximum size of the complete message to be fragmented and transmitted out the port.

Number of Buffered Messages - For a SAP Port, this indicates the number of messages (with size = to max Message size) that may be buffered for transmission at any time. The default value is 2.

4.2.2.2 Writing Messages to the Port

Now that the MAC Header, IP Header and UDP Header have been defined, it is time to define the data to be inserted into the AFDX Payload message portion of the AFDX frame.

	Preamble	Start Delimiter	MAC Header	IP Header	UDP Header	AFDX Payload Message	Sequence Number	FCS
bytes	7	1	14	20	8	17…1471	1	4

There are two sets of functions for writing messages to the port - one for AFDX Comm ports and one for SAP ports as defined in the following two sections.

4.2.2.3 Writing Messages to the AFDX Comm Port

For AFDX Comm ports, writing AFDX Payload messages to the AFDX frame is accomplished using the **FdxCmdTxUDPWrite** or **FdxCmdTxUDPBlockWrite** functions. These functions **can be executed while the transmitter is enabled** (or disabled), but should be performed after **FdxCmdTxUDPCreatePort** has been executed and a UDP Port handle obtained from that function. The size of the message written to the port must be less than or equal to the Maximum Message Size defined when creating the UDP port using **FdxCmdTxUDPCreatePort**. **FdxCmdTxUDPBlockWrite** performs in the same manner as **FdxCmdTxUDPWrite**, however, it allows the user to write to multiple ports with one function call. The transmission of the message written to the port varies for a Sampling and Queuing Port as follows:

a. **Sampling Port**

- **FdxCmdTxUDPWrite** or **FdxCmdTxUDPBlockWrite** should be called before the port is started (using **FdxCmdTxControl**) in order to initialize the data contents of the UDP message buffer.

- If the message written to the port(s) is smaller than the Maximum Message Size defined with **FdxCmdTxCreatePort,** the remaining bytes at the end of the UDP Buffer (with size equal to Maximum Message Size) will not be overwritten. (Remember - only one message can be written to a sampling port at a time.)

- AFDX Frame will be transmitted at the Sampling Rate once the port is started (using **FdxCmdTxControl**)

- If the message is to be updated each time the AFDX data frame is transmitted, the **FdxCmdTxUDPWrite** or **FdxCmdTxUDPBlockWrite** should be performed at the same sampling rate defined using **FdxCmdTxUDPCreatePort**.

b. **Queuing Port**

- **FdxCmdTxUDPWrite / FdxCmdTxUDPBlockWrite** initiates the transmission of the message(s).

- More than one message can be written to the queuing port using the **FdxCmdTxUDPBlockWrite** function. Care should be taken to insure that the number of messages written to the port does not exceed the Message Buffer size defined with **FdxCmdTxUDPCreatePort.**

- The transmission of the entire message may require multiple AFDX frame transmissions (fragmentation will be by IP layer).

The following code inserts a byte pattern of “050505…” into the AFDX Payload message portion of the AFDX frame. The UDP Port Handle is obtained from the **FdxCmdTxCreatePort** function.

```
//Write message to UDP Tx Port
/* create Data */
ul_ByteCount = 100;
for (i=0; i<ul_ByteCount; i++)
  uc_Data[i]=(AiUInt8)5;
if (FDX_OK != ( FdxCmdTxUDPWrite (ul_Handle, aul_UDPHandles[ul_HandleCnt],
         ul_ByteCount, uc_Data, &ul_BytesWritten)))
{
   printf("FdxCmdTxUDPWrite failed!!!\n");
   return(FDX_ERR);
}
else
{
   printf("FdxCmdTxUDPWrite() ul_BytesWritten:%ld", ul_BytesWritten);
```

```
}
```

Note: ***An alternative method of writing data to the port involves the use of the function*** *FdxCmdTxVLWrite or FdxCmdTxVLWriteEx****. These functions allow the user to write entire AFDX frames to the VL Buffer, therefore, providing maximum flexibility as to the content of the port's transmit output. When using this function, the UDP functions used to Create (****FdxCmdTxUDPCreatePort****), Destroy (****FdxCmdTxUDPDestroyPort****), Write (****FdxCmdTxUDPWrite****), Control (****FdxCmdTxUDPControl****) and Get Status (****FdxCmdTxUDPGetStatus****) are not applicable.***

4.2.2.4 Writing Messages to the SAP Port

For SAP ports, writing AFDX Payload messages to the AFDX frame is accomplished using the **FdxCmdTxSAPWrite** or **FdxCmdTxSAPBlockWrite** functions. These functions **can be executed while the transmitter is enabled** (or disabled), but should be performed after **FdxCmdTxSAPCreatePort** has been executed and a UDP Port handle obtained from that function. The size of the data written to the port must be less than or equal to the Maximum Message Size (maximum is 8Kbytes) defined when creating the UDP port using **FdxCmdTxSAPCreatePort**. **FdxCmdTxSAPBlockWrite** performs in the same manner as **FdxCmdTxSAPWrite**, however it allows the user to write AFDX message(s) (can be unique for each port) to multiple ports with one function call. Transmission considerations for a SAP port are as follows:

a. **FdxCmdTxSAPWrite / FdxCmdTxSAPBlockWrite** initiates the transmission of the message(s).

b. More than one message can be written to the SAP port using the **FdxCmdTxSAPBlockWrite** function. Care should be taken to insure that the number of messages written to the port does not exceed the Message Buffer size defined with **FdxCmdTxSAPCreatePort.**

c. The transmission of the entire message may require multiple AFDX frame transmissions (fragmentation will be by IP layer).

4.2.2.5 Individual UDP Port Error Injection, Skew and Enable/Disable

The function **FdxCmdTxUDPControl** provides the lowest level UDP port control available to manipulate the individual Sub VLs (UDP ports) in the following manner:

*Note**:*** ***This function requires that the Transmit Port has been enabled via*** *FdxCmdTxControl* ***function and the VL has not been disabled via the*** *FdxCmdTxVLControl* ***function. The*** *FdxCmdTxUDPControl* ***is the lowest level port control function available, therefore, all higher level functions (****FdxCmdTxControl* ***&*** *FdxCmdTxVLControl****) will supersede the Sub VL UDP port level control imposed with*** *FdxCmdTxUDPControl***.**

a. **Inject errors** for a certain number of AFDX frames, or cyclically, as defined in Table 4-8.

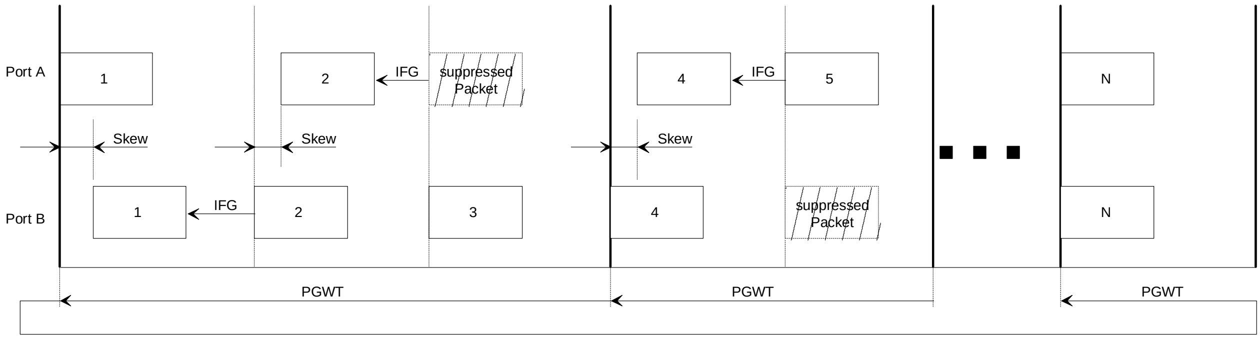
Table 4-8 Physical Error Injection

Error Type	Description:
CRC Error	CRC Error transmitted with this frame
Byte Alignment Error	Wrong Byte alignment in transmit frame, which means that an odd number of nibbles will be transmitted. Therefore, this error will also cause a CRC error condition.
Preamble Error	Wrong Preamble Sequence transmitted. If this type is selected, the Encoder device substitutes the first nibble of the Start Frame Delimiter with the value '1000' instead of '1001'
Physical Symbol ('HALT') Error	Physical Symbol Error. During Frame Transmission, the MAC-Encoder device asserts the Tx-Error signal, which forces the physical transceiver to transmit 'HALT' symbols.

b. For ports setup in **redundant** mode, **skew** by a user specified value or enable/disable the primary/redundant port's frame transmission as defined below and shown in Figure 4-4:

- Packet on Network A is delayed by the Skew value, related to Network B
- Packet on Network B is delayed by the Skew value, related to Network A
- Packet transmitted on both Networks (Skew=0)
- Packet only transmitted on Network A
- Packet only transmitted on Network B

Figure 4-4 Redundant Network Frame Transmission Options



Note: *The Skew Value between two redundant frames is defined with a resolution of 1 microsecond. Therefore, if the following frame pair is scheduled by an Interframe Gap, the resolution of the Interframe Gap timer is decreased from 1 GTU up to 1 microsecond.*

c. **Enable/Disable** individual UDP ports.

*Note**:** **Disabling/enabling the UDP port via** FdxCmdTxControl**,** FdxCmdTxVLControl**,** FdxCmdTxUDPControl **will not reset the error or skewing conditions previously configured. To disable error injection or skew at an individual UDP port, the user must issue the** FdxCmdTxUDPControl **function with parameters set as required.***

4.2.2.6 Individual UDP Port Status

The function **FdxCmdTxUDPGetStatus** provides the lowest level UDP port status information available including:

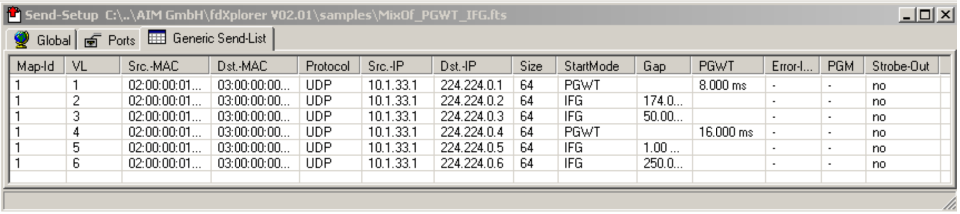
a. **Message Count** - Count of messages sent through this UDP port since the transmitter was started.

4.2.2.7 Changing the Source ID of a UDP Port

The function **FdxCmdTxUDPChgSrcPort** provides the user with the ability to change the port's source ID while transmitting data. This functionality might be used if the application included simulation of a server.

4.2.3 Generic Transmit Mode

When operating in Generic Transmit Mode, the user is provided with the capability to define each individual frame and the sequence of the frame to be transmitted by the port. Each frame defined provides information about the relative timing between the frames, error injection, payload-generation modes, transmission skew (in redundant operation mode) and/or special events like a digital output strobe-signal. For high-throughput, special payload-generation modes can be used, so the hardware takes parts of the frame-data from static send-fields. Because all frames must be pre-buffered on the hardware, the number of frames is limited to the board-resources.



Map-Id	VL	Src.-MAC	Dst.-MAC	Protocol	Src.-IP	Dst.-IP	Size	StartMode	Gap	PGWT	Error-I...	PGM	Strobe-Out
1	1	02:00:00:01...	03:00:00:00...	UDP	10.1.33.1	224.224.0.1	64	PGWT		8.000 ms	-	-	no
1	2	02:00:00:01...	03:00:00:00...	UDP	10.1.33.1	224.224.0.2	64	IFG	174.0...		-	-	no
1	3	02:00:00:01...	03:00:00:00...	UDP	10.1.33.1	224.224.0.3	64	IFG	50.00...		-	-	no
1	4	02:00:00:01...	03:00:00:00...	UDP	10.1.33.1	224.224.0.4	64	PGWT		16.000 ms	-	-	no
1	5	02:00:00:01...	03:00:00:00...	UDP	10.1.33.1	224.224.0.5	64	IFG	1.00 ...		-	-	no
1	6	02:00:00:01...	03:00:00:00...	UDP	10.1.33.1	224.224.0.6	64	IFG	250.0...		-	-	no

The functions described in this section should be used after the port has been configured for Generic Transmit mode using the function **FdxCmdTxModeControl,** as described in Section 4.2.1.1. Setting up the port when in Generic Transmit mode is basically a two step process with a function provided for statusing as described in the following sections:

a. Allocating a Transmit Queue

b. Defining the Frames / Writing to the Transmit Queue

...and once the transmit port is enable via **FdxCmdTxControl** as defined in Section 4.2.1.2.....

c. Transmit Queue Status

Generic Transmit Setup

(1) Allocate queue for the storage of the frames to be transmitted.

(2.1) Define the attributes (non-data) of the generic Tx frame

(2.2) Insert the data into the generic Tx Frame

(2.3) Write the Frame attributes and the Frame data to the Tx Queue

4.2.3.1 Allocating a Transmit Queue

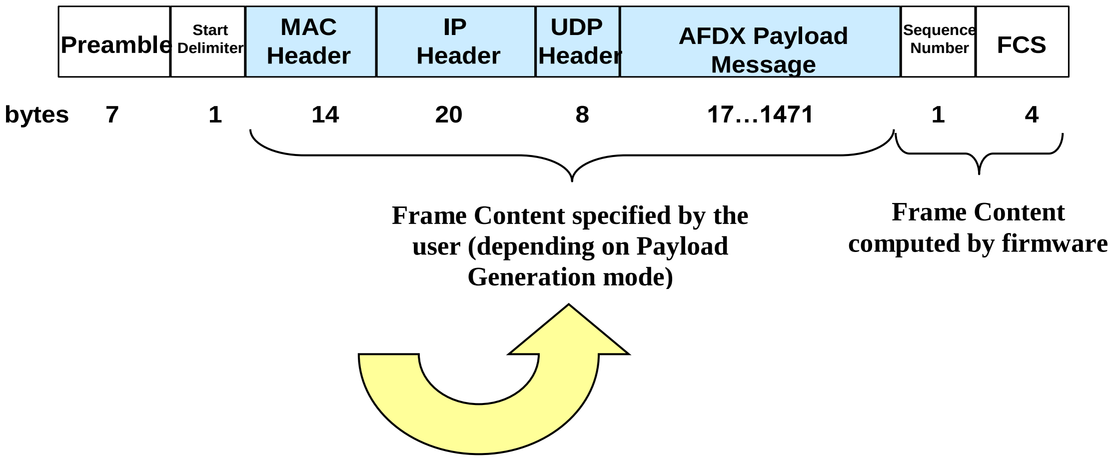
Allocation of the transmit queue involves the function **FdxCmdTxQueueCreate**. For Generic Transmit mode, this function basically defines a queue and the size of the queue to be used to transmit the frames that will be defined by the function **FdxCmdTxQueueWrite**. The size of the queue will depend on the number of generic transmit frames required for transmission. Remember, these frames can be transmitted cyclically or a user-specified number of times as defined with the Global Transmit function, **FdxCmdTxControl** (Section 4.2.1.2).

4.2.3.2 Defining the Frames / Writing to the Transmit Queue

The next steps are to define one or more AFDX frames then write the entire list of AFDX frames to the Transmit Queue using the function **FdxCmdTxQueueWrite**. Each frame entry to the queue will include the following:

a. **Frame Attributes** - define the manner in which the frame should be transmitted on the network. User-specifiable variables include those listed in Table 4-9.

b. **Frame Content** - 64 - 1518 bytes of frame data beginning with the MAC Header and ending with the FCS. MAC Header through the AFDX Payload message is defined by the user. The frame content required for definition depends upon the Payload Generation mode (See Table 4-10) selected for the frame attributes.



As you can see, there are many different combinations allowed for the definition of each AFDX Frame. Multiple AFDX frames can be defined in the Transmit Queue (taking care that the Transmit Queue size defined in **FdxCmdTxQueueCreate** can hold all frames defined) using one **FdxCmdTxQueueWrite** function call. Additional frames can be added to the Transmit Queue with additional **FdxCmdTxQueueWrite** function calls, however, the **port transmission must be disabled**. The new entries will always be queued at the end of the transmit queue. To purge the Transmit Queue of all frames, the port must be reinitialized using the **FdxCmdTxPortInit** function.

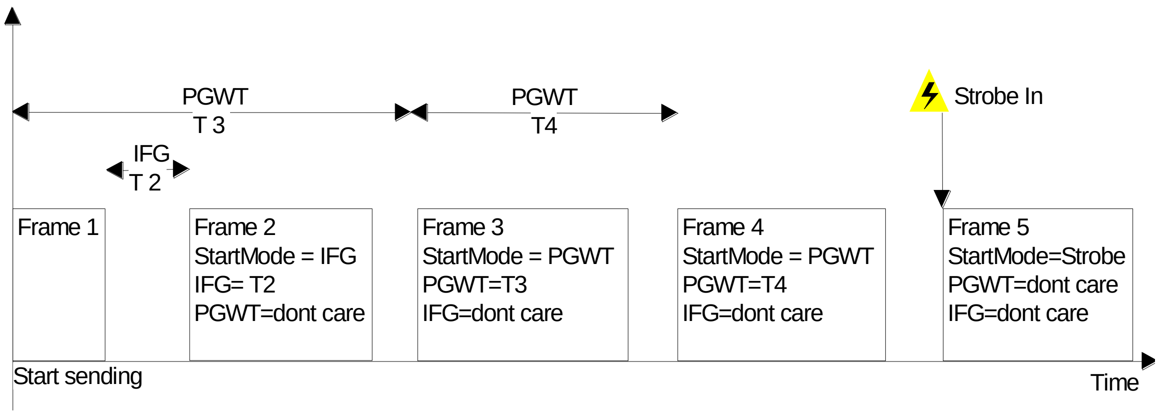
Table 4-9 Frame Attributes for Generic Transmit Frames

Frame Attribute	Description
Frame Size	Total size of the associated frame in Bytes (including FCS). (64-1518 bytes)
Payload Generation Mode	Defines AFDX frame fields that will be inserted by the MAC-Hardware from the static Tx data registers (setup using ***FdxCmdTxStaticRegsCtrl***). See Table 4-10 for Payload Generation Mode options.
Frame Start Mode	Starts transmission of this frame on one of three conditions (See Figure 4-5): - when user-specified InterFrame Gap (IFG)* time has expired - when user-specified Packet Group Wait Time (PGWT)** has expired - on external Trigger Strobe (See Section 4.2.1.3for Strobe Setup) *Gap between the end of the preceding frame and the current frame (resolution of 40ns). Range = 120 ns to approx. 655µsec. ** The time from the transmission start point of the last frame where the PGWT value is processed to the start point of the current frame with a resolution of 1us.
External Strobe	Enables/Disables output of external Strobe on transmission of this frame (See Section 4.2.1.3 for Strobe Setup)
Preamble Count	Varies the number of preamble Bytes (default 7 bytes)
Physical Error Injection	Same Physical Error Injection capabilities as defined for UDP Port-Oriented Simulation mode as shown in Table 4-8. (CRC Error, Byte Alignment Error, Preamble Error, Physical Symbol ('HALT') Error)
Sequence Number Control	Starting Sequence Number can be defined by the user. The offset added to the sequence number for each frame can also be specified.
Redundant Mode Network Select / Skew	For Redundant mode: Packet on Network A is delayed by the user-defined Skew* value, related to Network B Packet on Network B is delayed by the user-defined Skew* value, related to Network A Packet transmitted on both Networks (Skew=0) Packet only transmitted on Network A Packet only transmitted on Network B *Skew can be programmed with a resolution of 1µsec. Range is 0...65535 µsec.

Table 4-10 Payload Generation Mode Frame Content Source

Payload Generation Mode	Frame Content		
	From User	From Static Registers	
		Frame Data	Bytes (0-n)
No Payload Generation	User must provide complete frame	None	
IP Partial	MAC/IP/UDP Header (minus fields provided by Static Registers)	MAC Destination MAC Source MAC Type/Length IP Version IP Protocol Field UDP Checksum UDP Payload	2-5 3-5 complete complete complete complete complete
IP Full	MAC Header (minus fields provided by Static Registers)	MAC Destination MAC Source MAC Type/Length IP Header UDP Header UDP Payload	2-5 3-5 complete complete complete complete
IP Partial + Timetag	same as IP Partial	same as IP Partial + UDP Payload Timetag	
IP Full + Timetag	same as IP Full	same as IP Full + UDP Payload Timetag	

Figure 4-5 Packet Group Wait Time & Interframe Gap



Please note that the minimal granted packet group wait time of a frame (i.e Frame 3) depends on the actual size of the preceding frame (i.e Frame 2).

When you define packet group wait time and interframe-gap time you should pay attention to these frame-durations.

Frame-Size	10 Mbps	100 Mbps
19	31.20 µSec	3.12µSec
64	67.20 µSec	6.72 µSec
1518	1230.00 µSec	123.00 µSec
2000	1616.00 µSec	161.60 µSec
x	(x + 20)*0.80 µSec	(x + 20)*0.08 µSec

20 bytes must be added to the pure frame-size, because every frame begins with a preamble (7 bytes) and a start-delimiter (1 byte) and should have a minimal interframe-gap (12 bytes).

Additional Bytes	10 Mbps	100 Mbps
Preamble (7)	5.60 µSec	0.56 µSec
Start Delimiter (1)	0.80 µSec	0.08 µSec
min. Interframe gap (12)	9.60 µSec	0.96 µSec

The following code creates one Transmit Queue, and, for one AFDX frame, defines the frame attributes and content and writes the frame into the Transmit Queue. The AFDX Payload message is initialized to ASCII characters A-Q.

```
TY_FDX_TX_MODE_CTRL x_TxModeControl;
TY_FDX_TX_QUEUE_SETUP x_TxQueueCreate;
TY_FDX_TX_QUEUE_INFO x_TxQueueInfo;
struct my_Frame_tag {
   TY_FDX_TX_FRAME_HEADER x_Frame;
   AiUInt8 uc_Data[1000];
} My_Frame;
AiUInt8  Dt[100];

printf("\n FdxCmdTxQueueCreate...");
x_TxQueueCreate.ul_QueueSize = 0; //When using size 0, the internal default
                                  //queue size will be used.
if (FDX_OK!=(FdxCmdTxQueueCreate(ul_Handle,&x_TxQueueCreate,&x_TxQueueInfo)))
{
   printf("FdxCmdTxQueueCreate failed!!!\n");
}
else
{
   printf("FdxCmdTxQueueCreate done.\n");
}

//--- Create Frame for the Tx Queue
My_Frame.x_Frame.uc_FrameType                          = FDX_TX_FRAME_STD;
My_Frame.x_Frame.x_FrameAttrib.uc_NetSelect            = FDX_TX_FRAME_BOTH;
My_Frame.x_Frame.x_FrameAttrib.uc_ExternalStrobe       = FDX_DIS;
My_Frame.x_Frame.x_FrameAttrib.uc_FrameStartMode       = FDX_TX_FRAME_START_PGWT;
My_Frame.x_Frame.x_FrameAttrib.uc_PayloadBufferMode    = FDX_TX_FRAME_PBM_STD;

My_Frame.x_Frame.x_FrameAttrib.uc_PayloadGenerationMode = FDX_TX_FRAME_PGM_USER;
//no payload generation - all frame data defined by the user in this frame
My_Frame.x_Frame.x_FrameAttrib.uc_PreambleCount        = FDX_TX_FRAME_PRE_DEF;
My_Frame.x_Frame.x_FrameAttrib.ul_BufferQueueHandle    = 0;//used when payload buffer
                                                           //mode is not standard
My_Frame.x_Frame.x_FrameAttrib.ul_InterFrameGap        = 25; // 25=1usec;*/
My_Frame.x_Frame.x_FrameAttrib.ul_PacketGroupWaitTime  = 1000; // 1000=1msec*/
My_Frame.x_Frame.x_FrameAttrib.ul_PhysErrorInjection   = FDX_TX_FRAME_ERR_OFF;
My_Frame.x_Frame.x_FrameAttrib.ul_Skew                 = 0;  // redundant mode only
My_Frame.x_Frame.x_FrameAttrib.uw_FrameSize            = 64;  //bytes (includes CRC)
My_Frame.x_Frame.x_FrameAttrib.uw_SequenceNumberInit   = FDX_TX_FRAME_SEQ_INIT_AUTO;
My_Frame.x_Frame.x_FrameAttrib.uw_SequenceNumberOffset  = FDX_TX_FRAME_SEQ_OFFS_AUTO;

/* --- Frame 1 --- VL 60 */
us_FrameCount = 64;

//---MAC Dst= 0x03000000003c (VL 60)
Dt[ 0]=0x03;Dt[ 1]=0x00;Dt[ 2]=0x00;Dt[ 3]=0x00;Dt[ 4]=0x00;Dt[ 5]=0x3c;
//---MAC Src= 0x020000012120
Dt[ 6]=0x02;Dt[ 7]=0x00;Dt[ 8]=0x00;Dt[ 9]=0x01;Dt[10]=0x21;Dt[11]=0x20;
//---MAC Type/Length
Dt[12]=0x08;Dt[13]=0x00;

//---IP Header (Version/IHL, Type of service, Total length, Fragment ID,
// Time to live, Protocol, Header Checksum)
Dt[14]=0x45;Dt[15]=0x00;Dt[16]=0x00;Dt[17]=0x2d;Dt[18]=0x00;Dt[19]=0x00;
Dt[20]=0x40;Dt[21]=0x00;Dt[22]=0x01;Dt[23]=0x11;Dt[24]=0x6d;Dt[25]=0xa2;
//---IP Source Address 10.001.33.1
Dt[26]=0x0a;Dt[27]=0x01;Dt[28]=0x21;Dt[29]=0x01;
//---IP Destination Address 224.224.0.60 (VL 60)
Dt[30]=0xe0;Dt[31]=0xe0;Dt[32]=0x00;Dt[33]=0x3c;
```

```
//---UDP Source Port = 24
Dt[34]=0x00;Dt[35]=0x18;
//---UDP Dest Port = 23
Dt[36]=0x00;Dt[37]=0x17;
//---UDP Length = 25
Dt[38]=0x00;Dt[39]=0x19;
//---UDP Checksum
Dt[40]=0x00;Dt[41]=0x00;

//---AFDX Payload
Dt[42]=0x41;Dt[43]=0x42;Dt[44]=0x43;Dt[45]=0x44;Dt[46]=0x45;
Dt[47]=0x46;Dt[48]=0x47;Dt[49]=0x48;Dt[50]=0x49;Dt[51]=0x4a;
Dt[52]=0x4b;Dt[53]=0x4c;Dt[54]=0x4d;Dt[55]=0x4e;Dt[56]=0x4f;
Dt[57]=0x50;Dt[58]=0x51;

for ( i = 0 ; i< 59; i++)
   My_Frame.uc_Data[i] = (unsigned char) Dt[i];

if (FDX_OK!=(FdxCmdTxQueueWrite(ul_Handle,FDX_TX_FRAME_HEADER_GENERIC
                                  ,1,sizeof(My_Frame),&My_Frame)))
{
   printf("FdxCmdTxQueueWrite failed!!!\n");
}
else
{
   printf("FdxCmdTxQueueWrite done.\n");
}
```

4.2.3.3 Generic Transmit Queue Status

One status function is provided, **FdxCmdTxQueueStatus**, which will indicate the following while in Generic Transmit mode:

a. **Run Status** - indicates that frames have been written to the queue and the transmitter is up and running.

b. **Frames Sent** - the number of frames transmitted

c. **Frames in the Transmit Queue** - the number of frames written to the queue.

4.2.4 Replay Transmit Mode

When operating in Replay Transmit Mode, the user is provided with the capability to replay previously recorded data or playback data being captured real-time while in Chronologic Receive mode. (See Section 4.3for instructions on how to setup for Record Capture mode.) Replay mode does not reproduce any physical error conditions detected when the data was recorded, but protocol errors as well as size violations are replayable as listed in Table 4-11. A packet will be discarded by the firmware if any of the Non-replayable error conditions are detected in the replay data. (**Note:** Replay mode not yet available for GNET.)

Table 4-11 Errors Replayable/Not Replayable

	Error Definition	Symbol
Non- Replayable*	Wrong physical Symbol during frame reception.	PHY
	Wrong Preamble/Start Frame Delimiter received.	PRE
	Unaligned Frame length received	TRI
	MAC CRC Error.	CRC
	Short Interframe Gap Error (<960ns)	IFG
	Frame without valid Start Frame Delimiter received	SFD
Replayable	AFDX IP Framing Error (AFDX-IP frame specific settings violated).	IPE
	AFDX MAC Framing Error (AFDX-MAC frame specific settings violated).	MAE
	Long Frame Received (> 1518 Bytes up to 2000 bytes)	LNG
	Short Frame Received (40 to < 64 Bytes)	SHR
	VL specific Frame size Violation	VLS
	Sequence No. Mismatch	SNE
	Traffic Shaping Violation	TRS

*Note: The packet will be discarded by the firmware if any of the Non-replayable error conditions are detected in the replay data

The functions described in this section should be used after the port has been configured for Replay Transmit mode using the function **FdxCmdTxModeControl,** as described in Section 4.2.1.1. Setting up the port for Replay Transmit mode is basically a two step process with a function provided for statusing as described in the following sections:

a. Allocating a Transmit Queue

b. Writing the Replay data to the Transmit Queue

c. Transmit Queue Status

Replay Transmit Setup

(1) Allocate queue for the storage of the frames to be transmitted.

(2) Write the Replay file address to the Tx Queue

4.2.4.1 Allocating a Transmit Queue

Allocation of the transmit queue involves the function **FdxCmdTxQueueCreate**. For Generic Replay mode, this function basically defines a reloadable queue and the size of the queue. The size of the queue will depend on:

a. the Replay file size, when transmitting a pre-recorded file. Remember, this replay file can be setup to be transmitted at a specific start time, on an external strobe input or immediately as defined with the Global Transmit function, **FdxCmdTxControl** (Section 4.2.1.2).

b. the amount of data read from the Monitor Queue to be written to the Transmit Queue, for play-back of real-time data being captured.

4.2.4.2 Writing a Replay File to the Transmit Queue

The next step is to write the replay data entries to the Transmit Queue. If writing a replay file, the address and size of the replay file will be indicated, using the function **FdxCmdTxQueueWrite**. If writing real-time data from the Monitor Queue, the address and size of the Monitor Queue entries will be indicated. The Transmit Queue can be reloaded with Replay entries by issuing additional **FdxCmdTxQueueWrite** function calls, **even while the port is enabled** and transmitting data. The new Replay file entries will always be queued at the end of the transmit queue.

4.2.4.3 Replay Transmit Queue Status

One status function is provided, **FdxCmdTxQueueStatus**, which will indicate the following while in Replay Transmit mode:

a. **Empty Transmit Queue** - indicates transmit queue is created, but no Replay frame entries have been entered.

b. **Partially Full Transmit Queue** - the transmit queue is partially filled with Replay frame entries.

c. **Full Transmit Queue** - the transmit queue is full.

d. **All Frames Sent** - All replay frame entries sent to the transmit queue have been sent.

e. **Run Status** - indicates that frames have been written to the queue and the transmitter is up and running.

f. **Frames Sent** - the number of frames transmitted

4.3 Receiver Programming

The AFDX receive port can be configured in one of two **Receive** modes as listed below:

a. **VL-Oriented Receive Operation -** In this Receive Mode the UDP Port can receive and store messages for either "connection" oriented (AFDX Comm Ports) or "connectionless" oriented (SAP) ports. The Receive AFDX Comm ports are characterized by the address-quintuplet, (VL, Src.-IP, Dst.-IP, Src.-UDP, Dst.-UDP), each with its own message storage area. For an AFDX Comm port in this mode, the user must specify the exact address quintuplet in order for the VL frames to be captured. SAP receive ports, however, may receive AFDX messages from multiple sources. Therefore, the user only specifies the VL and UDP/IP destination adress in order for the VL frames to be captured. The source of the AFDX frame is only determined after the message has been received. (Trigger capability is not provided in this receive mode.)

b. **Chronological Receive Operation (Monitor Mode) -** In this Receive mode all captured frames are stored in one single memory area. In this mode, all VL data streams are captured, in addition, the user can specify additional VL filters/checking to be performed if desired. This mode provides for recording/saving the captured data for replay. Four **Capture modes**, which define what data is captured and when data capture begins, are available:

- **SingleShot-Standard**
  In this mode, each port uses a pre-defined onboard-memory-area (**singleshot-memory**) for capturing frames. After this memory is filled with frames no more frames will be stored. The size of singleshot-memory depends on your board type and RAM-Size. **Trigger Control Blocks** (TCBs) can be used in this mode to define the trigger condition that will start data capture (default capture start is when a frame is received) and how much "pre-trigger data" is to be stored in the Monitor Buffer.

- **SingleShot-Selective**
  This mode is very similar to SingleShot-Standard mode, but **Trigger-Control-Blocks** are used for **filtering,** i.e. **what** data will be captured. Before a frame is saved in the SingleShot-memory, it will be evaluated using the active TCB. Only those frames which meet the TCB condition will be saved.

- **Continuous**
  In this mode, the SingleShot-memory is used as a **ring-buffer**. As soon as the memory is full, old frames will be overwritten with new frames (**wrap-around**). **Trigger Control Blocks** can be used in this mode to define the trigger condition that will start data capture (default capture start is when a frame is received).

- **Record**
  In this mode, the Monitor buffer is organized in the same way as in Continuous mode. However, the **frames will be written directly to a user-specified file or directly to an output port configured for Replay**. **Trigger Control Blocks** can be used in this mode to define the trigger condition that will start data capture (default capture start is when a frame is received).

Table 3-2 defines the key features and differences between the Chronologic and VL-Oriented Receive modes.

If your application requires the reception and processing of AFDX traffic, this section will provide you with the understanding of the Receiver programming functions required for use within your application. Regardless of the Receive mode you select, Global Receiver functions must be performed first. This section will describe Receiver Functions as follows:

a. Global Receiver Functions
   - Port Initialization & Rx Mode Setup
   - Reception Control
   - Strobe Input/Output Usage (optional)
   - Global Receiver Status

b. VL-Oriented Receive Operation
   - Defining the VL and UDP Port to be Monitored/Captured
   - Reading messages from the UDP Port
   - Individual UDP Port Status

c. Chronological Receive Operation (Monitor Mode)
   - Defining the Capture mode
   - Allocating the Monitor Queue
   - Additional VL Filter Capability
   - Creating Trigger Conditions
   - Reading the Captured Data.

4.3.1 Global Receiver Functions

Global Receiver functions are the functions that apply to all modes of operation (VL-Oriented and Chronological (Monitor) modes). The major functions of the Global Receiver functions include:

a. Port Initialization & Rx Mode Setup

b. Reception Control

c. Strobe Input/Output Usage (optional)

d. Global Receiver Status.

These Global Receiver functions are described in the following sections.

4.3.1.1 Port Initialization and Rx Mode Setup

After the Board setup is complete, an individual Receive Port can be initialized and the mode can be configured for either VL-Oriented or Chronological Receive Mode using the functions listed below. Once the mode has been configured, the user can then program the receive processes as defined in the section applicable to the mode (Section 4.3.1.1 - 4.3.1.2).

4

Port Rx Setup
(1) Assign Portmap ID to each Rx port
(2) Define Receiver mode (VL-Oriented or Chronological Receive)

a. **FdxCmdRxPortInit -** will perform **initialization** and reset of the Receive port global characteristics. This must be the first Receiver function call issued. **The user must assign a Portmap ID to each Rx port**. This Portmap ID is a virtual ID assigned to the physical Port. The Receive PortMap ID is contained in the data read from the monitor queue (**FdxCmdMonQueueRead**). The Portmap ID aids in the identification of the physical port from which the data was received, especially for applications using multiple AFDX cards and using receive ports in redundant mode. State after initialization includes:

- **Global Statistics Available** - the global receiver status, output from the **FdxCmdRxGlobalStatistics** function call, are available. (See Section 4.3.1.4 for further information)

- **All VL statistics enabled** - if any VL's were previously disabled using the function call **FdxCmdRxVLControl**, they are now enabled for **statistics** gathering only.

- **Chronological Receive Mode** - this is the default mode of operation for a receive port. However, no VLs are enabled for **capturing**.

- **No Trigger Control Blocks (TCBs)** - no TCBs are enabled. (Trigger Control Blocks define special triggers that will trigger the start of data capture.)

> ***Note**: If in Chronological Mode, TCB setup is not required if the user wants to start capture on the first AFDX frame received since this is the default trigger.*

b. **FdxCmdRxModeControl** - provides configuration for the following:

- **Receive mode -** selecting this mode will define the group of functions used next for receive process setup.

  1. **VL-Oriented Receive Operation -** In this Receive Mode each port is characterized by either the address-quintuplet (VL, Src.-IP, Dst.-IP, Src.-UDP, Dst.-UDP) for AFDX Comm ports, or only the VL, Dst.-IP, and Dst.-UDP for SAP ports. Each UDP port has its own message-memory area. (Trigger capability is not provided in this receive mode.)

  2. **Chronological Receive Operation (Monitor Mode)-** In this Receive mode all captured frames are stored in one single memory area. In this mode, all VL data streams are captured, in addition, the user can specify additional VL filters/checking to be performed if desired. This mode provides for recording/saving the captured data for replay.

Default Payload mode - **the payload mode defines the amount of data from the AFDX frame that will be stored in the receive buffers when in Chronological Monitor mode. (The entire AFDX payload data message is stored when in VL-Oriented Receive mode). Regardless of the payload mode chosen, the frame statistics will always be computed. The frame data defined to be stored for each Payload mode is shown in**

- Table 4-12.

> ***Note**: The payload mode for individual VL's can be modified using the **FdxCmdRxVLControl** function call.*

- **Default Chronological mode -** the default Chronological mode defines what data is captured when in Chronological Monitor mode (When in VL-Oriented Receive mode, only statistics are computed. The user must specify which VL's are to be captured.): The default Chronological modes available are shown in Table 4-13.

> ***Note**: The Chronological mode for individual VL's can be modified using the **FdxCmdRxVLControl** function call.*

Table 4-12 Default Frame Content Captured Based on Payload Mode

Default Payload Mode	Frame Content Captured *(option in Chronological mode only)*
Full Payload	All
IP Extended	MAC/IP Header + 20 bytes of the IP payload
IP	MAC/IP Header
MAC	MAC Header

Table 4-13 Default Chronological Modes

Default Chronological Mode	VLs Captured	Statistics Available
Enable Count	None	VL-Oriented counters and Global statistics
Monitor Enable All	All VLs enabled	VL-Oriented counters and Global statistics
Monitor Enable Good	All VLs enabled - Only error free frames captured	VL-Oriented counters and Global statistics

The following code example uses API S/W Library constants and structures to initialize one port using a Portmap equal to 2 and configures the mode to Chronological Monitor mode. The default setup for Payload mode is to capture the full AFDX frame, and the default Chronological mode is setup to capture and provide statistics for all VLs.

```
TY_FDX_PORT_INIT_IN       x_PortInitIn;
TY_FDX_PORT_INIT_OUT      x_PortInitOut;
TY_FDX_RX_MODE_CTRL_IN    x_ModeCtrlIn;
TY_FDX_RX_MODE_CTRL_OUT   x_ModeCtrlOut;

//--- Initialization
x_PortInitIn.ul_PortMap = 2;
if (FDX_ERR == FdxCmdRxPortInit( ul_Handle, &x_PortInitIn, &x_PortInitOut))
{
   printf("Port Reset failure!!!\n");
}
else
{
   printf("Port Receiver Initialized\n");
}

//--- mode control -> select Chrono Mode
x_ModeCtrlIn.ul_ReceiveMode = FDX_RX_CHRONO;
x_ModeCtrlIn.ul_DefaultPayloadMode = FDX_PAYLOAD_FULL;
x_ModeCtrlIn.ul_DefaultCronoMode = FDX_RX_DEFAULT_MON_ENA_ALL;
x_ModeCtrlIn.ul_GlbMonBufferSize = 0;  // if zero, a default value will be used
if (FDX_OK != (FdxCmdRxModeControl( ul_Handle, &x_ModeCtrlIn, &x_ModeCtrlOut)))
{
   printf("Port 2 Mode Control Failure!!!\n");
}
```

```
else
{
   printf("Port Set to Chrono Monitor Receive Mode\n");
   printf("Port Global Mon Buffer Size: %d bytes\n",
      x_ModeCtrlOut.ul_GlbMonBufferSize);
}
```

Note: ***The Global Monitor Buffer Size provides the user with the capability to specify the Monitor buffer size, however, in most cases, the default Monitor Buffer size is the Global RAM size (4Mbytes)/# of ports on the board.***
For GNET the default Monitor Buffer size is 64 Mbytes.

Note: ***The Global Monitor Buffer Size provides the user with the capability to specify the Monitor buffer size, however, in most cases, the default Monitor Buffer size is the Global RAM size (4Mbytes)/# of ports on the board.***

4.3.1.2 Reception Control

After all filters and frame validation methods of the port have been programmed (as defined in Section 4.3.2 - 4.3.3), the method used to start/stop reception of the data should be defined. There is one global function providing Rx Control:

a. **FdxCmdRxControl** - starts/stops reception for a **Physical port** and resets counters including:

- **Starting/Stopping** the receiver - provides for
  1. Starting/stopping the receiver on command
- **Global Statistics Reset -** defines which global counters to reset including:
  1. Reset nothing
  2. Reset all
  3. Reset only error related counters.

The following code configures port 2 to receive AFDX frames and resets all receive counters for this port upon this command.

```
TY_FDX_RX_CTRL x_RxControl;

x_RxControl.ul_StartMode             = FDX_START;
x_RxControl.ul_GlobalStatisticReset = FDX_RX_GS_RES_ALL_CNT;
if (FDX_OK != (FdxCmdRxControl( ul_Handle, &x_RxControl)))
{
   printf("Failure to start Receiver!!!\n");
}
else
```

```
{
   printf("Receiver Started\n");
}
```

4.3.1.3 Strobe Input/Output Usage

As a programming option, the strobe input/output signals for the ports can be used in various ways to control triggers to start data capture and/or to indicate the occurrence of a specific frame condition or Capture status as shown in Table 4-7. This table indicates the functions required to utilize the Strobe external input/output signals and in which Receive modes these functions are applicable.

Every two AFDX ports on the AIM AFDX board utilize the 15 pin D-Sub connector for Strobe Input and Outputs. The layout for the pin is shown in Figure 4-3. Each physical port can be assigned one Strobe input line and one Strobe output line.

For GNET-2 the Strobe and IRIG Connecteor Pinout is as described in Figure 4-3. For GNET-4 the Strobe and IRIG Connecteor is available on a separate break out module. The pinout of the HDSUB 15 is identical to Figure 4-3 except the IRIG pins. On the APX-GNET-BOP pin 1 and 2 are internally connected. The IRIG Input function on pin 1 is always active. If the module shall be configured to act as IRIG- Time source (IRIGout activated), then a jumper has to be installed on the APX-GNET-4 module. Please refer also to according hardware manuals.

Table 4-14 Strobe Input/Output Receiver Functions

Function Type	Applicable mode: VL-Oriented	Applicable mode: Chronological Monitor	API Function	Strobe- In Capability	Strobe-Out Capability
Global Receiver	X	X	**FdxCmdRxTrgLineControl**	Defines the strobe Input/Output lines to be used for the receive port	
Global Receiver	X	X	**FdxCmdRxVLControlEx**		Strobe out on frame reception for a specific VL*
Global Receiver	X	X	**FdxCmdRxVLControlEx**		Strobe-out on erroneous frame reception for a specific VL*
Chronological Monitor		X	**FdxCmdMonCaptureControl**		Strobe-out on Capture stop or Strobe out on Half Monitor Buffer Full*
Chronological Monitor		X	**FdxCmdMonCaptureControl**		Strobe-out on Capture start/re-start*
Chronological Monitor		X	**FdxCmdMonTCBSetup**	Use Strobe-in to enable Trigger	

* *The trigger output strobe is asserted after trigger condition has been detected by the BIU Processor. Thus, the packet which caused the trigger has to be completely received and processed by the BIU Processor before the strobe is asserted. Therefore, the strobe will appear with a delay on the trigger output, relative to the packet on the network. The delay time is dependent on the current network traffic.*

4.3.1.4 Global Receiver Status

The Global Receiver function group includes three receiver status function calls, **FdxCmdRxStatus**, **FdxCmdRxGlobalStatistics** and **FdxCmdRxVLGetActivity,** as shown in Table 4-15. Additional lower level status can be obtained when in VL-Oriented Receive mode by using the function **FdxCmdRxUDPGetStatus**, and, when in the Chronological Receive mode, by using **FdxCmdMonGetStatus** as described in Section 4.3.2 and 4.3.3 respectively.

7

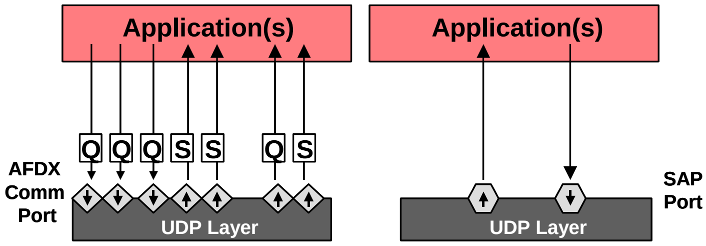
Retrieve Status
(1) Tx Status
(2) Rx Status
(3) Retrieve Captured data

Table 4-15 Global Receiver Status

API Function	Status	Description
FdxCmdRxStatus	Receiver Status	Port is Stopped/Running/Error
FdxCmdRxGlobalStatistics (composite status for all VLs)	Total Byte Count	Count of total Bytes received since start of the last counter reset
	Error Free frames	Count of error free frames since start or the last counter reset
	Erroneous Frames	Count of erroneous frames since start or the last counter reset
	Bytes per second	Bytes received per second
	Frames per second	Frames received per second
	Physical errors (PHY)	Count of frames with wrong physical Symbol during frame reception
	Preamble Errors (PRE)	Count of frames with wrong Preamble/Start Frame Delimiter received
	Unaligned Frame length (TRI)	Count of unaligned Frame length received
	MAC CRC Errors (CRC)	Count of MAC CRC Error
	IFG Errors (IFG)	Count of short IFG Error (<960ns)
	IP Header Errors (IPE)	Count of IP static header field errors
	MAC Header Errors (MAE)	Count of MAC static header field errors
	Start Frame Delimiter Errors (SFD)	Count of frames received without valid Start Frame Delimiter
	Frame Length Errors (VLS)	Count of VL specific Frame size Violation
	Sequence Number Errors (SNE)	Count of Sequence Number integrity errors
	Traffic Shaping Errors (TRS)	Count of Traffic Shaping Violation
	Frames with size = 1-63 bytes (SHR)	
	Frames w/size = 64-127 bytes	
	Frames w/size = 128-255 bytes	
	Frames w/size = 256-511 bytes	
	Frames w/size = 512-1023 bytes	
	Frames w/size = 1024-1518 bytes	
	Frames w/size = >1518 bytes (LNG)	
FdxCmdRxVLGetActivity (all active or specific VLs)	Enable Mode	Enable mode configured with **FdxCmdRxVLControl**
	Payload Mode	Payload mode configured with **FdxCmdRxVLControl**

	Verification Mode	Verification mode configured with **FdxCmdRxVLControl**
	List of Error types detected: PHY, PRE, TRI, CRC, IFG, IPE, MAE, SFD, LNG, SHR, VLS, SNE	(Errors defined above for **FdxCmdRxGlobalStatistics**)
	VL valid frame count	
	VL erroneous frame count	
	Frames per second	Frames received per second
	Redundant Frames	Count of redundant frames discarded

4.3.2 VL-Oriented Receive Mode



In this Receive Mode the UDP Port can receive and store messages for either "connection" oriented (AFDX Comm Ports) or "connectionless" oriented (SAP) ports. The Receive AFDX Comm ports are characterized by the address-quintuplet, (VL, Src.-IP, Dst.-IP, Src.-UDP, Dst.-UDP), each with its own message storage area. For an AFDX Comm port in this mode, the user must specify the exact address quintuplet in order for the VL frames to be captured. SAP receive ports, however, may receive AFDX messages from multiple sources. Therefore, the user only specifies the VL and UDP/IP destination adress in order for the VL frames to be captured. The source of the AFDX frame is only determined after the message has been received. When operating in VL-Oriented Receive mode the AFDX UDP ports can perform redundancy management, integrity checking and traffic shaping.

The functions described in this section should be used after the port has been configured for VL-Oriented Simulation using the function **FdxCmdRxModeControl,** as described in Section 4.3.1.1. Setting up the port when in VL-Oriented Receive mode consists of the following main functions:

VL-Oriented Setup

(1) Define VL characteristics to look for (VL ID and range) and type of verification required
(2) Setup Rx UDP port (AFDX Comm port or SAP port...)

a. Defining Virtual Link and UDP port to be monitored/captured

...and once the Receive port is enabled via **FdxCmdRxControl** as defined in Section 4.3.1.2 .....

b. Reading messages from the UDP Port

c. Individual UDP Port Status

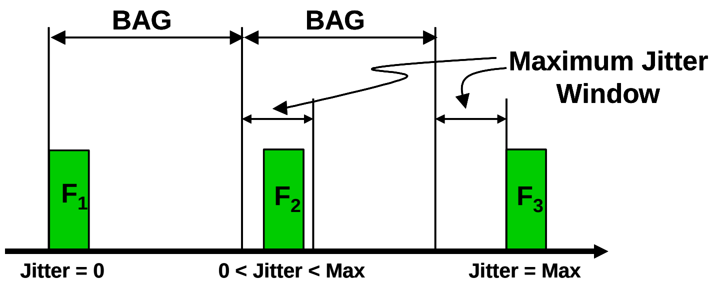
4.3.2.1 Defining the Virtual Link and UDP Port to be Monitored/Captured

The following two functions are associated with definition of the VL and UDP Port streams to be monitored/captured for a defined port.

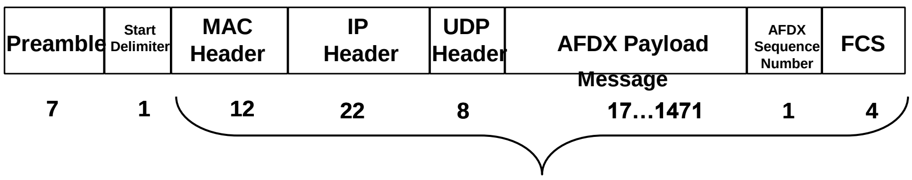
FdxCmdRxVLControl - enables (or disables) an individual **VL** to be monitored/captured. Remember, when in VL-Oriented Receive mode, all VL's are initially disabled (with the **FdxCmdRxPortInit** function), therefore, this function is required if any VL is to be monitored/captured. The VL(s) must be setup for extended operation and configured for the following receive processing options:

- **Verification mode -** allows the user to define the type of verification to be performed on the VL as shown in Table 4-16. Each verification mode requires that parameters be set to establish the range of acceptable receive frame behavior as shown in Table 4-17 and including:

  1. **BAG** values are in milliseconds and include 1, 2, 4, 8, 16, 32, 64, 128 msecs. **Jitter** range is 1 to 65535 microseconds.



  2. **Max/Min Frame Length** includes all fields shown below (except the Preamble and Start Delimiter):



Frame-Size

  3. **Max Skew** - the maximum time difference between the arrival time of the redundant frame with the same sequence number. Values are in microseconds with a range of 0 to 65535µsec.

- **Extended Filter -** allows the user to specify that the VL frames meet an additional filter before being captured. This generic filter **compares up to 4 bytes of the AFDX frame with a user specified value**. The user has the option to store the frame if the values match/don't match.

b. **FdxCmdRxVLControlEx** - (optional) extended VL function to configure output of a **strobe signal** or **interrupt** upon **VL** frame reception or frame reception error or **interrupt** on VL Buffer Full/Half Full/Quarter Full.

Table 4-16 Verification Mode Options and Required Parameters (for VL-Oriented Rx Mode)

Verification Mode	Description	Default Setting		Parameters Required				
		redundant mode	single mode	BAG	Max Jitter	Max Frame Length	Min Frame Length	Max Skew
Redundancy Management	Enable Redundancy Management as described in AFDX End System Detailed Functional Specification. The discard counter is incremented if the current received frame is discarded by the RM facility for either Port A or Port B.	✓				✓		✓
Traffic shaping Verification	Enable Traffic Shaping Verification like described in AFDX Switch Detailed Functional Specification. If during the previous frame check, an error occurs (except if Sequence number error or Invalid Packet Processing is enabled), the frame is not fed to the TS facility.		✓	✓	✓	✓	✓	
VL specific Frame size Check	Maximum frame size for the given VL is checked.	✓	✓			✓		
Sequence Number Integrity check	Sequence numbering of the incoming frames are checked	✓						
Invalid Packet processing	All Packets, also the erroneous, will be passed through to the buffer							

c. There are two sets of functions associated with creation of the UDP Rx port: AFDX Comm port functions and SAP functions as described below:

For AFDX Comm Port (Sampling or Queuing Connection-oriented port):

- **FdxCmdRxUDPCreatePort** - this function will define the characteristics associated with a **UDP** Sampling/Queuing port associated with a VL as listed in the example below (showing four UDP ports). A **UDP Handle** to the UDP Port is returned when this command is issued successfully. The handle is used for all further communication/control of the UDP Port. An AFDX Comm port is connection-oriented, therefore, the entire address quintuplet is specified for the create port function to define the point-to-point connection.

FdxCmdRxUDPCreatePort							
VL ID	Type (Sampling or Queuing)	Source IP	Destination IP	Source UDP	Destination UDP	Maximum Message Size	Number of buffered messages
60	Sampling	10.1.33.1	224.224.0.0	1	1	100 bytes	1
60	Sampling	10.1.33.1	224.224.0.0	2	2	90 bytes	1
60	Sampling	10.1.33.1	224.224.0.0	3	3	80 bytes	1
60	Queuing	10.1.33.1	224.224.0.0	4	4	max. 8000 bytes	3

Maximum Message Size - the size of the AFDX Payload Message. The size is fixed for Sampling ports. For queuing ports, it is the maximum size of the complete message to be reassembled and received at the queuing port.

Number of Messages - for a Sampling Port this is always 1. For a Queuing Port, this indicates the number of AFDX complete (reassembled) messages (with size = to max Message size) to be buffered when received. The default value is 2.

The following code enables one VL (VL 60) one UDP Sampling port (1) for monitoring/capturing. Verification Mode is disabled, therefore, there will be no Traffic Shaping performed on the received data frames for that VL. The whole AFDX frame will be stored in the VL buffer.

```
//--- VL control (per VL which we want to watch)
x_VLControl.ul_VLId            = DEF_VL;
x_VLControl.ul_VLRange         = 1;
x_VLControl.ul_EnableMode      = FDX_RX_VL_ENA_EXT; //required value for VL-Oriented
x_VLControl.ul_PayloadMode     = FDX_PAYLOAD_FULL;
x_VLControl.ul_TCBIndex        = 0;

x_VLDesc.ul_VerificationMode   = FDX_RX_VL_CHECK_DISA;
x_VLDesc.ul_VLBufSize          = 0x8000;
```

```
   if (FDX_OK != (FdxCmdRxVLControl(ul_Handle, &x_VLControl, &x_VLDesc)))
   {
      printf("Receive VL Control Failure!!!\n");
   }
   else
   {
      printf("VL:%d Enabled for Capturing on Port\n", DEF_VL);
   }


//--- create udp-port for read
x_UdpDesc.ul_PortType           = FDX_UDP_SAMPLING;
x_UdpDesc.x_Quint.ul_IpDst      = DEF_DST_IP;
x_UdpDesc.x_Quint.ul_IpSrc      = DEF_SRC_IP;
x_UdpDesc.x_Quint.ul_UdpDst     = DEF_DST_UDP1;
x_UdpDesc.x_Quint.ul_UdpSrc     = DEF_SRC_UDP1;
x_UdpDesc.x_Quint.ul_VlId       = DEF_VL;
x_UdpDesc.ul_UdpNumBufMessages= 1;
x_UdpDesc.ul_UdpMaxMessageSize= DEF_UDP_MAXMSG;

if (FDX_OK != FdxCmdRxUDPCreatePort( ul_Handle, &x_UdpDesc, &g_pUdp1Port2Handle))
{
   printf("Receive UDP Port Creation Failure!!!n");
}
else
{
   printf("Rx UDP Port Created on Port -- VL:%d UDP Port:%d\n",DEF_VL,DEF_DST_UDP1);
}
```

For SAP Port (Connectionless-oriented port):

- **FdxCmdRxSAPCreatePort** - this function will define the characteristics associated with the SAP port. A **UDP Handle** to this UDP SAP Port is returned when this command is issued successfully. The handle is used for all further communication/control of the UDP Port. The characteristics for the SAP Tx port include only those listed below. Remember, for a SAP port - since the port is "connectionless" it can receive from multiple E/S's and the source is determined at the time of reception, therefore, only the Destination IP/UDP addresses are required.

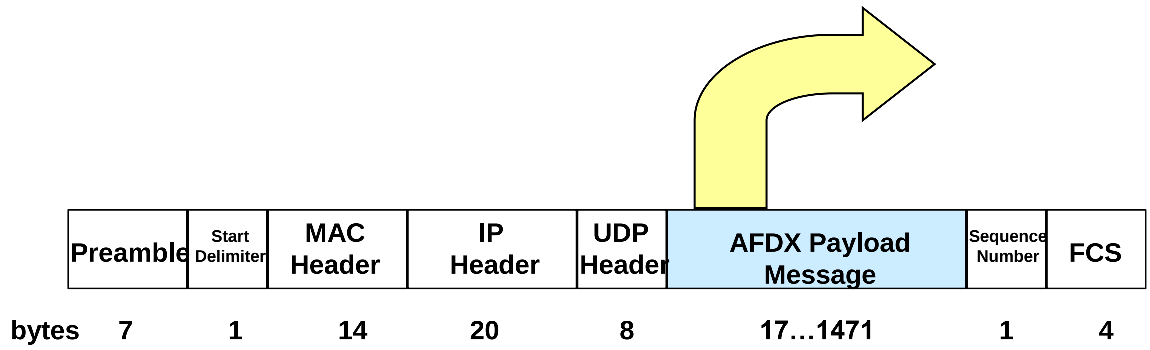
FdxCmdRxSAPCreatePort				
VL ID	Destination IP	Destination UDP	Maximum Message Size	Number of buffered messages
60	224.224.0.0	1	max. 8000 bytes	1
60	224.224.0.0	2	max. 8000 bytes	1
60	224.224.0.0	3	max. 8000 bytes	1
60	224.224.0.0	4	max. 8000 bytes	3

Maximum Message Size - the size of the AFDX Payload Message. For SAP ports, it is the maximum size of the complete message to be reassembled and received at the port.

Number of Buffered Messages - For a SAP Port, this indicates the number of complete reassembled messages (with size = to max Message size) that may be buffered. The default value is 2.

4.3.2.2 Reading Messages from the Port

Now that the MAC Header, IP Header and UDP Header have been defined for the UDP port as described above, the Receive port can be started using the Global Receive function call, **FdxCmdRxControl** as defined in Section 4.3.1.2. The user may then want to read the AFDX Payload Message received at the port.



There are two sets of functions for reading messages received by the port - one for AFDX Comm ports and one for SAP ports as defined in the following two sections.

4.3.2.3 Reading Messages from the AFDX Comm Port

For AFDX Comm ports, reading AFDX Payload messages received by the port is accomplished using the **FdxCmdRxUDPRead** or **FdxCmdRxUDPBlockRead** functions. These functions should be performed after **FdxCmdRxCreatePort** has been executed and a UDP Port handle obtained from that function. The size of the data read from the port cannot exceed the Maximum Message Size defined when creating the UDP port using **FdxCmdRxUDPCreatePort**. **FdxCmdRxUDPBlockRead** performs in the same manner as **FdxCmdRxUDPRead**, however, it allows the user to read to from multiple ports with one function call.

The format of the Payload message received from the port is shown in Figure Figure 4-6.

Figure 4-6 AFDX Comm Port Message Buffer Layout

AFDX Comm Port Message Buffer Layout				
	31 24	23 16	15 8	7 0
Buffer Header	Time Tag High			
	Time Tag Low			
	Message Size			
	Reserved			
AFDX Payload Message	Received UDP Message sampling message: up to UDP payload (1 – 1471 bytes) queuing message: up to 8Kbytes)			

One entry will contain one complete sampling or queuing message and a Buffer Header containing the time tag of the last received message. (For queuing ports, where the messages can be fragmented, it is the time tag of the last received fragment.)

Programming considerations are listed below for each type of AFDX Comm UDP port:

a. **Sampling Port**

- If the message received at the port is larger than the Maximum Message Size defined with **FdxCmdRxCreatePort,** the extra bytes will be discarded.

- **FdxCmdRxUDPRead** or **FdxCmdRxUDPBlockRead** should be performed at the sampling rate expected at the receive port for that UDP. The UDP Buffer used to store a received Sampling port's AFDX message is overwritten when the subsequent AFDX frame is received. (Number of messages read returned by the function call will be 0 or 1.)

b. **Queuing Port**

- The reception of the entire message may require the reception of multiple AFDX frames (reassembly will be by IP layer).

- More than one message can be read from the queuing port using the **FdxCmdRxUDPBlockRead** function.

- Queuing messages are received asynchronously, therefore, the UDP Queuing port should be polled at a rate appropriate for expected Queuing messages. If a message has not been received or is in the process of being received by the UDP port, **FdxCmdRxUDPRead** /

FdxCmdRxUDPBlockRead will return a zero for Number of messages actually read.

The following code checks the status of a UDP port, and if the number of messages received is greater than 0, the UDP message is read and printed.

```
/* Read Udp Port Status */
if (FDX_OK != (r_RetVal = FdxCmdRxUDPGetStatus ( ul_HandlePort, aul_UDPHandles[k],
   &x_UdpRxStatus )))
    printf("\r\n\n FdxCmdRxUDPGetStatus() failed.");
else
{

   printf("\r\n\n FdxCmdRxUDPGetStatus() UDP-HandleNo:%d MsgCount:%10ld
   MsgErrorCount:%10ld ",k, x_UdpRxStatus.ul_MsgCount,
   x_UdpRxStatus.ul_MsgErrorCount);

   /* Read Udp Port Data */

   if (0 < x_UdpRxStatus.ul_MsgCount)
   {
       if (FDX_OK != (r_RetVal = FdxCmdRxUDPRead (  ul_HandlePort, aul_UDPHandles[k],
   3, &ul_MsgRead, auc_Data )))
           printf("\r\n FdxCmdRxUDPRead() failed.");
       else
       {
       if (ul_MsgRead > 0)
       {

       TY_FDX_UDP_HEADER *px_UDPHeader;
       /* Get pointer to first Header (start of Array) */
       px_UDPHeader = (TY_FDX_UDP_HEADER*) auc_Data;

        for (j=0; j<ul_MsgRead; j++)
        {
           TY_FDX_IRIG_TIME x_IrigTime;
           /* Get IRIG Time */
           FdxFwIrig2StructIrig( &px_UDPHeader->x_FwIrigTime, &x_IrigTime);

           /* print size and Time information */
           printf("\r\n FdxCmdRxUDPRead() MsgRead:%10ld MsgSize = %08lx IRIG Day:%ld
        Time:%02ld:%02ld:%02ld:%03ld:%03ld",
           ul_MsgRead, px_UDPHeader->ul_MsgSize,
           x_IrigTime.ul_Day, x_IrigTime.ul_Hour, x_IrigTime.ul_Min,
           x_IrigTime.ul_Second, x_IrigTime.ul_MilliSec, x_IrigTime.ul_MicroSec);

           /* print Buffer Data */
           pData = (AiUInt8 *)px_UDPHeader;
           for (i=0; i < px_UDPHeader->ul_MsgSize; i++)
           {
           if (0 == (i%16))
              printf("\r\n Data %04lx: %02x", i, pData[i+16]);
           else
               printf(" %02x", pData[ i+16]);
           }

           /* Get pointer to next message */
           px_UDPHeader = (TY_FDX_UDP_HEADER*)    ( ((AiUInt32)(px_UDPHeader)) +
      px_UDPHeader->ul_MsgSize + sizeof(TY_FDX_UDP_HEADER) );

           }
        }
        else
```

```
            printf("\r\n FdxCmdRxUDPRead() MsgRead:%10ld ", ul_MsgRead);
         }
     }
}
```

4.3.2.4 Reading Messages from the SAP Port

For SAP ports, reading AFDX Payload messages received by the port is accomplished using the **FdxCmdRxSAPRead** or **FdxCmdRxSAPBlockRead** functions. These functions should be performed after **FdxCmdRxSAPCreatePort** has been executed and a UDP Port handle obtained from that function. The size of the data read from the port cannot exceed the Maximum Message Size defined when creating the UDP port using **FdxCmdRxSAPCreatePort**. **FdxCmdRxSAPBlockRead** performs in the same manner as **FdxCmdRxSAPRead**, however, it allows the user to read to from multiple ports with one function call.

The format of the Payload message received from the port is shown in Figure 4-7. Notice the difference between the data received at a SAP port and the data received at the AFDX Comm Port is that the SAP port identifies the source (IP/UDP source) of the AFDX message. This is performed since the port is "connectionless" i.e. the destination of the message is not determined until time of transmission, therefore, a destination port could receive messages from multiple source ports.

Figure 4-7 SAP Port Message Buffer Layout

SAP Port Message Buffer Layout				
	31 24	23 16	15 8	7 0
Buffer Header	Time Tag High			
	Time Tag Low			
	Message Size			
	IP Source Address			
	UDP Source Port			
	Reserved			
AFDX Payload Data	Received UDP Message message size: up to 8Kbytes			

One entry will contain one complete re-assembled message and a Buffer Header containing the source and the time tag of the last received message/fragment.

Programming considerations are listed below for the SAP port:

a. The reception of the entire message may require the reception of multiple AFDX frames (reassembly will be by IP layer).

b. More than one message can be read from the port using the **FdxCmdRxSAPBlockRead** function.

c. Messages are received asynchronously, therefore, the SAP port should be polled at a rate appropriate for expected messages. If a message has not been received or is in the process of being received by the UDP port, **FdxCmdRxSAPRead** / **FdxCmdRxSAPBlockRead** will return a zero for Number of messages actually read.

4.3.2.5 Individual UDP Port Status

The function **FdxCmdRxUDPGetStatus** provides the lowest level UDP port status information available including:

a. Message Count - Count of messages written to this UDP port since the receiver was started.

b. Error Count - Count of erroneous message written to the UDP port buffer since the receiver was started.

4.3.3 Chronological Monitor Receive Mode

In this Receive mode all captured frames are stored in one single memory area. In this mode, all VL data streams can be captured with options to capture only user-specified VLs or range of VLs. In addition, the user can specify additional VL filters/checking to be performed if desired including redundancy management, integrity checking and traffic shaping. This mode provides for recording/saving the captured data for replay. Four **Capture** modes provide different methods for storage of captured frames in the Monitor buffer and how data is to be captured. In addition, an extensive and flexible trigger function is provided allowing the user to define specific conditions on which to trigger/start data capture. Strobe inputs/outputs can also be used with the trigger functions to signal start/stop of data capture or to enable specific trigger conditions.

The functions described in this section should be used after the port has been configured for Chronological Monitor Receive mode using the function **FdxCmdRxModeControl,** as described in Section 4.3.1.1. Setting up the port when in Chronological Monitor mode consists of the major steps defined below and described in further detail in the following sections:

> **Chronologic (Monitor) Setup**
> *(1) Define capture mode*
> *(2) Create Monitor queue to hold captured data.*

a. Defining the Capture mode

b. Allocating the Monitor Queue

c. Additional VL Filter Capability (optional)

d. Create Trigger conditions (optional)

...and once the Receive port is enable via **FdxCmdRxControl** as defined in Section 4.3.1.2 .....

e. Reading the Captured Data (optional)

f. Retrieving Monitor Status

Note: ***If you set the Default Chronological mode to FDX_RX_DEFAULT_ENA_CNT (using the*** *FdxCmdRxModeControl* ***function)where only the VL-Oriented*** *counters* ***are updated and VLs are disabled for*** *capturing****, the functions described in this section do not apply, with the exception of the VL Filter functions described in Section 4.3.3.3.***

4.3.3.1 Defining the Capture Mode

The function **FdxCmdMonCaptureControl** provides configuration of one of four **Capture** modes and **Strobe** output based on Monitor Buffer Capture conditions as described below:

Capture Modes			
SingleShot-Standard	**SingleShot-Selective**	**Continuous**	**Record**

a. **Capture Modes**

- **SingleShot-Standard**
  In this mode, each port uses a pre-defined onboard-memory-area (**singleshot-memory**) for capturing frames. After this memory is filled with frames no more frames will be stored. The size of singleshot-memory depends on your board type and RAM-Size. **Trigger Control Blocks** (TCBs) can be used in this mode to define the trigger condition that will start data capture (default capture start is when a frame is received) and how much "pre-trigger data" is to be stored in the Monitor Buffer.

  Listed below are some application examples:

  - Capture messages before an erroneous packet is received (using 10% of Monitor Buffer) for a particular VL and all messages after the trigger event until the Monitor buffer is full.

  - Capture messages received for a particular VL before an external strobe input is received (using 25% of Monitor Buffer) and all messages for the VL after the strobe input until the Monitor Buffer is full.

- **SingleShot-Selective**
  This mode is very similar to SingleShot-Standard mode, but **Trigger-Control-Blocks** are used for **filtering,** i.e. **what** data will be captured. Before a frame is saved in the SingleShot-memory, it will be evaluated using the active TCB. Only those frames which meet the TCB condition will be saved.

  Listed below are some application examples:

  - Capture a message only if the value of a specific parameter is equal to a specified value.

  - Capture only packets with a certain error type.

  - Capture packet when an external strobe input occurs.

- **Continuous**
  In this mode, the SingleShot-memory is used as a **ring-buffer**. As soon as the memory is full, old frames will be overwritten with new frames (**wrap-around**). **Trigger-Control Blocks** can be used in this mode to define the trigger condition that will start data capture (default capture start is when a frame is received).

  Listed below are some application examples:

  - Continuously capture all messages received for all/a particular VL/or range of VLs received and read/send the monitor buffer data to a display which provides continuous updates.

  - Start capture when any frame is received for a particular VL, capture only the frames received for the VL, and setup a TCB to indicate a trigger condition when packets are received for the VL with an unaligned frame length error. Send the monitor buffer data to a display which provides continuous updates and shows the packets which create a trigger event.

- **Record**
  In this mode, the Monitor buffer is organized in the same way as in Continuous mode. However, the **frames can be written directly to a user-specified file for later replay, or fed directly to an output port, configured for Replay, for immediate loopback**.

  Trigger-Control Blocks can be used in this mode to define the trigger condition that will start data capture (default capture start is when a frame is received).

  Listed below are some application examples:

  - Continuously capture all messages received for all/a particular VL/or range of VLs received and save the data in a file for subsequent replay.

  - Continuously capture all messages received for a particular VL with a specific MAC/IP/UDP port address that contains a hex value of 'FAF3' in the fifth word of the payload data area for subsequent replay or analysis.

b. **Strobe Output** - as described in Section 4.3.1.3 strobe output signals can be used by the Receiver when in Chronological Receive mode to signal the following conditions:

- Capture has **stopped due to full Monitor** Capture Buffer (SingleShot-Standard or SingleShot-Selective Capture modes)

- Capture has **stopped due to half full Monitor** Capture Buffer (Continuous or Record mode)

- Capture has **started** (for all Capture modes)/**re-started** (for SingleShot Selective Capture mode)

4.3.3.2 Allocating the Monitor Queue

After the Chronological Monitor mode has been configured (**FdxCmdRxModeControl**) and the Capture mode has been defined (**FdxCmdMonCaptureControl**), a Monitor queue should be allocated if data is to be captured. A Monitor Queue is used to store the captured data. When in the Continuous or Recording Capture mode, the frame data will be buffered in a second level queue of the onboard Target Software. In Single-shot Standard and Selective Capture mode, the frames will be read directly from the Monitor queue. Allocation of the Monitor queue involves the function **FdxCmdMonQueueControl**. This function will return a Queue ID to be used for all future Monitor queue functions.

4.3.3.3 Additional VL Filter Capability

When in Chronological Receive mode, the default VL-capture setting (initiated with function **FdxCmdRxModeControl** set to Chronological Receive mode) is for all received VL's to be captured. If you want to **specify the VL or range of VLs to be captured**, you need the command(s) defined below. Otherwise, continue with Section 4.3.3.4, Creating Trigger Conditions.

*Note***:** ***This VL filter function is executed prior to Trigger Control Block processing.***

a. **FdxCmdRxVLControl** - enables (or disables) an individual **VL or range of VLs** to be monitored/captured.

Payload mode - **allows the user to override the default payload mode previously defined with** FdxCmdRxModeControl**. The payload modes available are shown in**

- Table 4-12.

- **TCB Index -** allows the user to specify trigger control block processing for the given VL. See Section 4.3.3.4 for TCB setup details.

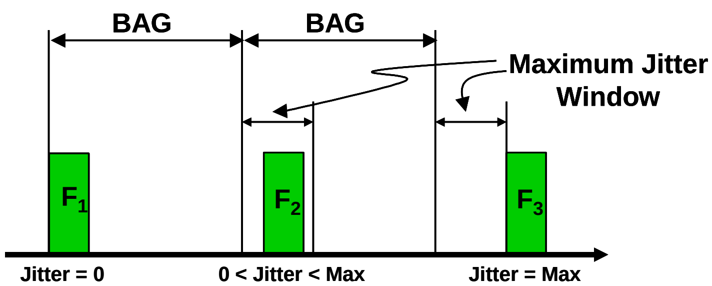
- **Enable mode** - configures the level of monitoring to be performed as defined below:

Monitoring Level	Global statistics only	VL-specific statistics	Store Good frames	Store Erroneous frames	Allow extended Verification mode and extended filtering*
ENA_STAT	X				
ENA_CNT	X	X			
ENA_MON_GOOD	X	X	X		
ENA_MON_ALL	X	X	X	X	
ENA_EXT	X	X	X	X	X

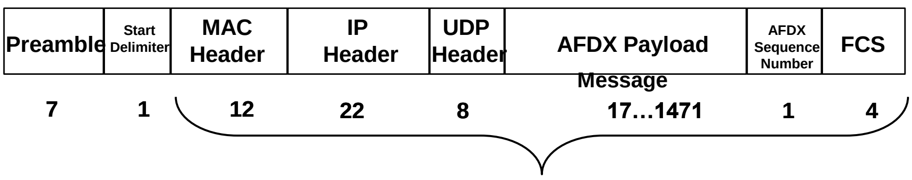
**Verification mode, Extended Filter and FdxCmdRxVLControlEx (for extended strobe signal and interrupt output) is defined below*

- **Verification mode -** allows the user to define the type of verification to be performed on the VL or range of VLs. (Only valid for Enable Mode set to ENA_EXT.) Each verification mode requires that parameters be set to establish the range of acceptable receive frame behavior as shown in Table 4-16 including:

1. **BAG** values are in milliseconds and include 1, 2, 4, 8, 16, 32, 64, 128 msecs. **Jitter** range is 1 to 65535 µsecs



2. **Max/Min Frame Length** includes all fields shown below (except the Preamble and Start Delimiter):



Frame-Size

3. **Max Skew** - the maximum time difference between the arrival time of the redundant frame with the same sequence number. Values are in microseconds with a range of 0 to 65535 µsecs.

- **Extended Filter -** allows the user to specify that the VL frames meet an additional filter before being captured. (Only valid for Enable Mode set to ENA_EXT.) This generic filter **compares up to 4 bytes of the AFDX frame with a user specified value**. The user has the option to store the frame if the values match/don't match.

b. **FdxCmdRxVLControlEx** - (optional) extended VL function to configure output of a **strobe signal** or **interrupt** upon **VL** frame reception or frame reception error or **interrupt** on VL Buffer Full/Half Full/Quarter Full. (Only valid for Enable Mode set to ENA_EXT.)

Table 4-17 Verification Mode Options and Required Parameters (for Chronological Monitor Receive Mode)

Verification Mode	Description	Default Setting		Parameters Required				
		redundant mode	single mode	BAG	Max Jitter	Max Frame Length	Min Frame Length	Max Skew
Redundancy Management	Enable Redundancy Management as described in AFDX End System Detailed Functional Specification. The discard counter is incremented if the current received frame is discarded by the RM facility for either Port A or Port B.	✓				✓		✓
Traffic shaping Verification	Enable Traffic Shaping Verification like described in AFDX Switch Detailed Functional Specification. If during the previous frame check, an error occurs (except if Sequence number error or Invalid Packet Processing is enabled), the frame is not fed to the TS facility		✓	✓	✓	✓	✓	
VL specific Frame size Check	Maximum frame size for the given VL is checked.	✓	✓			✓		
Sequence Number Integrity check	Sequence numbering of the incoming frames are checked	✓						
Invalid Packet processing	All Packets, also the erroneous, will be passed through to the buffer							

The following example sets up a VL Filter to capture only frames with VL ID of 1-10 when in Single-Shot Standard Capture mode. Extended Verification is enabled for these VLs such that Traffic Shaping can be performed. The parameters required to be set for Traffic shaping are shown above and set as indicated below. Any errors associated with the Traffic shaping parameters will be indicated in the Monitor Buffer entry for the associated frame. Counters for errors associated with the Traffic Shaping can be obtained globally for all VLs using the function **FdxCmdRxGlobalStatistics** or for individual VLs by using function **FdxCmdRxVLGetActivity** as indicated in Table 4.3.1.3-I.

```
/*  This example sets up a VL Filter to capture only frames with VL ID of 1-10.  */
/* Extended Verification is enabled such that Traffic Shaping can be performed.  */

TY_FDX_MON_CAP_MODE     x_CapMode;
TY_FDX_RX_VL_CTRL x_VLControl;
TY_FDX_RX_VL_DESCRIPTION x_VLDescription;

x_CapMode.ul_CaptureMode = FDX_MON_SINGLE; /* Single-Shot Standard capture mode */
x_CapMode.ul_TriggerPosition = 50; /*Set Trigger postion to middle of Monitor memory*/
x_CapMode.ul_Strobe  = FDX_MON_STROBE_DIS;/* No strobe on Capture start/stop */

x_VLControl.ul_VLId         = 1;
x_VLControl.ul_VLRange      = 10;
x_VLControl.ul_EnableMode   = FDX_RX_VL_ENA_EXT;
x_VLControl.ul_PayloadMode  = FDX_PAYLOAD_FULL;
x_VLControl.ul_TCBIndex     = 0xFF;

x_VLDescription.ul_Bag            = 16;     /* 16 milliseconds */
x_VLDescription.ul_Jitter         = 40;     /* 40 milliseconds */
x_VLDescription.ul_MaxFrameLength = 1400;   /* 1400 bytes */
x_VLDescription.ul_MaxSkew        = 0;      /* N/A for non-redundant mode */
x_VLDescription.ul_VerificationMode    = FDX_RX_VL_CHECK_TRAFIC;
x_VLDescription.ul_VLBufSize                       = 0;
x_VLDescription.x_VLExtendedFilter.ul_FilterMode     = FDX_DIS;
x_VLDescription.x_VLExtendedFilter.ul_FilterMask     = 0;
x_VLDescription.x_VLExtendedFilter.ul_FilterPosition = 0;
if (FDX_OK != (FdxCmdRxVLControl(ul_Handle,&x_VLControl,&x_VLDescription)))
   printf("\r\nFdxCmdRxVLControl() failed.");
```

4.3.3.4 Creating Trigger Conditions

Chronological Monitor mode has a **default trigger defined to start capture once any frame has been received for all VL's**. If you need more specific triggers to define when to start data capture or to define what is to be captured then this section will guide you in understanding TCB setup.

> *Note***:** ***At packet reception, the Monitor TCB processing will be executed after the Error Verification facility, the Redundancy Management (if enabled) and the VL- Filter processing (Section 4.3.3.3). Therefore, if one of the previous facilities discards the received packet, then the monitor TCB processing will not be executed.***

The API functions described in this section will provide flexible and comprehensive trigger conditions and trigger sequencing for monitor start, enabling detailed analysis of VL packet based traffic. Triggers can be defined for error conditions, external strobe input, data patterns received within the frame, or to make it easy, the reception of any frame received.

The Trigger Control Blocks (TCBs) define an event/condition within a received data packet that trigger data capturing or assert an external strobe. TCB(s) can be linked/assembled to create a sequence of trigger events, which will start the data capturing or assert an external strobe as shown in Table 4-18.

Table 4-18 TCB Content

TCB Parameter	Description
TCB Index	1 - 253
Trigger Type	Trigger on: 1. Error (Error Trigger) 2. External Strobe* 3. Generic Data Pattern (Generic Trigger) 4. Reception of any frame
for Generic Trigger	The position relative to start of frame, the mask and the compare value for the Generic Data Pattern Trigger Type
for Error Trigger	The Error Trigger Condition for the Error Trigger Type (See Table 4-19)
Next True Index	The index of the next TCB to evaluate after the condition for this TCB is True
Next False Index	The index of the next TCB to evaluate after the condition for this TCB is False
Trigger Bits	Trigger Bits in the Monitor Status Trigger Pattern to set/clear if the TCB evaluation is True (for start of capture condition)
TCB Extended Parameters	Assert strobe if TCB evaluation is True Assert interrupt if TCB evaluation is True

**Note: If the 'Trigger on External Strobe Event' is used in redundant operation mode, the TCB process shall allocate the same trigger input line for Port A and Port B.*

Table 4-19 Error Conditions Available for Triggers

Error Definition	Symbol
Wrong physical Symbol during frame reception.	PHY
Wrong Preamble/Start Frame Delimiter received.	PRE
Unaligned Frame length received (not applicable for GNET)	TRI
MAC CRC Error.	CRC
Short Interframe Gap Error (<960ns)	IFG
Frame without valid Start Frame Delimiter received	SFD
AFDX IP Framing Error (AFDX-IP frame specific settings violated).	IPE
AFDX MAC Framing Error (AFDX-MAC frame specific settings violated).	MAE
Long Frame Received (> 1518 Bytes up to 2000 bytes)	LNG
Short Frame Received (40 to < 64 Bytes)	SHR
VL specific Frame size Violation	VLS
Sequence No. Mismatch	SNE
Traffic Shaping Violation	TRS

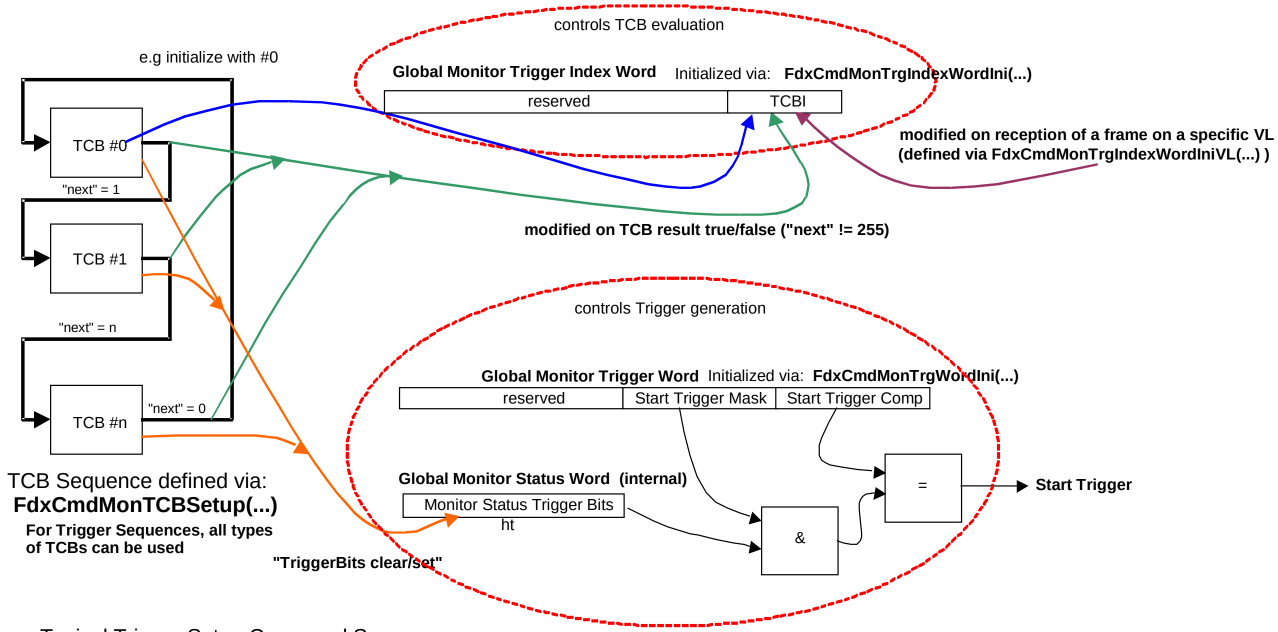
Once you have setup your trigger conditions using the TCB(s), you will then tell the trigger process which TCB to evaluate. This is done by setting up the global **Monitor Trigger Index Word** using **FdxCmdMonTrgIndexWordIni.**

You also need to tell the trigger process what the final value of the **Monitor Status Trigger Word** will be when the TCB or sequence of TCB conditions is true. This is done by setting up the **Start Trigger Compare** and **Mask** value using function **FdxCmdMonTrgWordIni**.

As shown in Figure 4-8, each TCB defines the modification of the **Monitor Status Trigger Word** each time a TCB evaluation is True. If the **Monitor Status Word**, when masked with the user-defined **Start Trigger Mask** is equal to the **Start Trigger Compare**, then the start trigger condition will become true and data capture will be started.

If you need to enable the trigger process to evaluate only a specific VL, you need to use **FdxCmdMonTrgIndexWordIniVL** which will enable the TCB process to evaluate the TCB Index specified with **FdxCmdMonTrgIndexWordIniVL** when that VL is being received/evaluated.

Figure 4-8 TCB Evaluation Process



Typical Trigger Setup Command Sequence
1 Define TCB(s) and implicitly a Trigger Sequence if neccessary **FdxCmdMonTCBSetup(...)**
2 Initialize Monitor Trigger Word **FdxCmdMonTrgWordIni(...)**
3 Initialize Monitor Trigger Index Word with the TCB Start Index **FdxCmdMonTrgIndexWordIni(...)**
4 Initialize VL related Trigger Index with a TCB Index if neccessary **FdxCmdMonTrgIndexIniVL(...)**

To review, the following functions are used to setup the trigger functionality when in Chronological Receive mode.:

a. **FdxCmdMonTCBSetup** - configures one TCB as shown in Table 4-18. Multiple **FdxCmdMonTCBSetup** commands may be required when TCBs are to be linked together.

b. **FdxCmdMonTrgWordIni** - defines the final Start Trigger Compare value that the Monitor Status Trigger Word must equal when masked with the Start Trigger Mask. Once these values are equal, the trigger will be initiated and capture will be started. (Strobe output can be performed on capture start/restart based on how the user defined the parameters in **FdxCmdMonCaptureControl**.)

c. **FdxCmdMonTrgIndexWordIni** - defines the TCB to be used for evaluation by the TCB process.

d. **FdxCmdMonTrgIndexIniVL** - (optional) defines the TCB to be used for a specific VL. This TCB will temporarily override the global Monitor Trigger Index Word while packets from this VL are being received and evaluated. (As described in Section 4.3.3.3, **FdxCmdRxVLControl** defines an initial value for a trigger condition to be evaluated for a particular VL or range of VLs. **FdxCmdMonTrgIndexIniVL** can be used to modify the trigger condition to be evaluated for a specific VL "on-the-fly", thus overriding the TCB Index specified with the function **FdxCmdRxVLControl**.)

```
/*
 * This example shows how to set up a sequence of 2 Trigger Control Blocks when in
 * Selective Capture mode.  The Capture mode and TCBs are setup to capture only the
 * frames received for VL100 with either a short or long frame error.  If an error
 * occurs, a strobe output will be generated.
 */
TY_FDX_MON_CAP_MODE      x_CapMode;
TY_FDX_MON_TCB_SET       x_MonTCBSet;
TY_FDX_MON_TRG_WORD_INI x_MonTrgWordIni;


x_CapMode.ul_CaptureMode = FDX_MON_SELECTIVE; /* switch to selective capture mode */
x_CapMode.ul_TriggerPosition = 0; /*Trigger postion N/A in Selective mode*/
x_CapMode.ul_Strobe   = FDX_MON_STROBE_DIS;/* No strobe on Capture start/stop */

printf("\r\n Setup Capture Mode ");
if (FDX_OK != (FdxCmdMonCaptureControl(ul_Handle,&x_CapMode)))
   printf("\r\nFdxCmdMonCaptureControl() failed.");

/* Setup TCB No 1 to trigger on generic event (VL=100)*/
x_MonTCBSet.ul_TrgType         = FDX_TRG_GENERIC;
x_MonTCBSet.ul_NextTrueIndex   = 2;
x_MonTCBSet.ul_NextFalseIndex  = 1;
x_MonTCBSet.ul_TriggerBits     = 0x010F;    /* Set Bits 0x01; Reset Bits 0x0F */
x_MonTCBSet.ul_TCBEx           = 0;

x_MonTCBSet.x_GenTrg.ul_GenBytePos  = 4;
x_MonTCBSet.x_GenTrg.ul_GenTrgType  = FDX_TRG_TCB_GEN_STD;
```

```
x_MonTCBSet.x_GenTrg.ul_GenTrigComp =0x00640000; /* Compare MAC address of VL 100 */
x_MonTCBSet.x_GenTrg.ul_GenTrigMask = 0xFFFF0000;

if (FDX_OK != FdxCmdMonTCBSetup (ul_Handle,1,&x_MonTCBSet))
   printf("\r\nFdxCmdMonTCBSetup() failed.");

/* Setup TCB No 2 to trigger on long frame or short frame error */
x_MonTCBSet.ul_TrgType         = FDX_TRG_ERROR;
x_MonTCBSet.ul_NextTrueIndex  = 3;
x_MonTCBSet.ul_NextFalseIndex = 1;
x_MonTCBSet.ul_TriggerBits     = 0x020F;  /* Set Bits 0x02; Reset Bits 0x0F */
x_MonTCBSet.ul_TCBEx           = 1;    /* Assert strobe output if TCB eval is true */

x_MonTCBSet.x_ErrTrg.ul_ErrType = FDX_LONG_FRAME_ERROR|FDX_SHORT_FRAME_ERROR;

if (FDX_OK != FdxCmdMonTCBSetup (ul_Handle,2,&x_MonTCBSet))
   printf("\r\nFdxCmdMonTCBSetup() failed.");

/* Setup Function Trigger Word */
x_MonTrgWordIni.ul_StartTriggerComp = 0x03; /* Trigger compare is set to combination
                                               of TCB's set bit. */
x_MonTrgWordIni.ul_StartTriggerMask = 0x0F;
if (FDX_OK != FdxCmdMonTrgWordIni (ul_Handle,&x_MonTrgWordIni))
   printf("\r\nFdxCmdMonTrgWordIni() failed.");

/* Setup Function Trigger Index Word to start evaluating TCB1*/
if (FDX_OK != FdxCmdMonTrgIndexWordIni (ul_Handle,1))
   printf("\r\nFdxCmdMonTrgIndexWordIni() failed.");
```

4.3.3.5 Reading the Captured Data

After you have configured the receiver for the appropriate Capture mode, VL-Filter and/or Trigger setup, the Receiver can be started using **FdxCmdRxControl** as defined in Section 4.3.1.2. Captured data can then be retrieved from the Monitor Queue.

The default size of the Monitor Queue is ((Global RAM size) 4Mbytes / # of ports). So, if you have two ports on your board, the Monitor Queue size will be 2 Mbytes. The size of the Monitor Queue can be specified with the **FdxCmdRxModeControl** function.

When designing your Capture Frame Retrieval software there are several design considerations to remember including:

a. The size of the Monitor Queue and the expected rate of capture - determines how often you need to read captured frames from the Monitor Queue

b. The Capture mode selected

- **Continuous and Record Capture modes** - continuously fills/refills the monitor queue, thus overwriting any entries previously captured after the Queue is full. Therefore, if you want to display/record all data captured you need to read entries from the Monitor Queue (**FdxCmdMonQueueRead**) periodically at a rate that will insure no entries are lost. **FdxCmdMonQueueRead** initially reads the number of frames requested beginning at the first frame in the queue. The next time you call the **FdxCmdMonQueueRead**, the pointer is automatically updated to read the next frame in the queue. Sequential access to the captured frames is the default.

  When in **Record** mode, captured data read from the Monitor Queue can either be used directly for Replay via the **FdxCmdTxQueueWrite** function (if configured for Replay) or saved to a record file to be used for later replay.

- **SingleShot-Standard or SingleShot-Selective** - both modes will only fill the Monitor Queue once. You may want to wait until the Monitor Queue is full before reading by checking Monitor Queue status (**FdxCmdMonQueueStatus**) for Full condition. For SingleShot-Selective, since only frames that meet the trigger condition are captured, the Monitor Buffer may take too long to fill (depending on your trigger setup), therefore, you may want to periodically read the Monitor Queue (**FdxCmdMonQueueRead)** to determine if any new entries have been captured.

  If required, you may need to use the Seek function, **FdxCmdMonQueueSeek**, to override the read pointer in the Monitor Queue used when **FdxCmdMonQueueRead** is called. **FdxCmdMonQueueSeek** can be used, for example, to read the 3rd captured frame on the first call to the Read command. Or, for instance, it can set the internal read pointer to the second Monitor Queue entry from the Start Trigger position.

The Tell function, **FdxCmdMonQueueTell**, just returns the current location of the read pointer within the queue. This can be used to help the user keep track of the internal Monitor Queue pointer if needed.

c. A strobe output signal can be output on Capture stop, on Half Monitor Buffer Full or Capture start/re-start using **FdxCmdMonCaptureControl** as discussed in Section 4.3.1.3.

Each Monitor Queue entry is comprised of a Fixed Header Entry and AFDX Frame (as defined by Payload mode) as shown in Table 4-20 and .

Table 4-20 Monitor Buffer Entry Layout (AFDX)

	Monitor Buffer Entry Layout (AFDX)				
	Word	31 24	23 16	15 8	7 0
Fixed Entry Header	0	**Previous Buffer Start Pointer** - pointer to previous frame			
	1	**Next Buffer Start Pointer** - pointer to the next frame			
	2	**Buffer Entry Control Word (BECW)**			
	3	Reserved			
	4	**Frame Header Word 0 (FHW0)** - Bitmap of frame errors and VL ID			
	5	**Frame Header Word 1 (FHW1)** - Network ID, IFG info, bytes received			
	6	**Frame Header Word 2 (FHW2)** - Sequence Number and frame size (minus CRC)			
	7	**Time Tag High Word (TTHW)**			
	8	**Time Tag Low Word (TTLW)**			
AFDX Frame		Received AFDX- FRAME (802.3 defines: 64 to 1518 bytes but it may be less or more at Frame size violation case) See Figure 4-9 for format			

For Frame Header Contents refer to Table 4-22.

Table 4-21 Monitor Buffer Entry Layout (GNET)

	Monitor Buffer Entry Layout (GNET)				
	Word	31 24	23 16	15 8	7 0
Fixed Entry Header	0	**Buffer Entry Control Word (BECW)**			
	1	**Buffer Entry Size Word (BESW)**			
	2	**Buffer Header Type Word (BHTW)**			
	3	**Next Buffer Start Pointer** – pointer to the next frame			
	4	**Frame Header Word 0 (FHW0)** – MAC ID, bitmap of frame errors and VL ID			
	5	**Frame Header Word 1 (FHW1)** - IFG info, frame bytes received			
	6	**Time Tag High Word (TTHW)**			
	7	**Time Tag Low Word (TTLW)**			
AFDX Frame		Received AFDX- FRAME (802.3 defines: 64 to 1518 bytes but it may be less or more at Frame size violation case) See Figure 4-9 for format			

For Frame Header Contents refer to Table 4-22.

Table 4-22 Buffer Header Contents (AFDX and GNET)

Value	Word AFDX	Bit AFDX	Word GNET	Bit GNET	Description
Buffer Type	BECW	31..30	BECW	31..30	Buffer Entry Type 0: Default Receive Buffer Entry other values: Reserved
FHTY	-	-	BHTW	31..30	Frame Header Type. 0= Standard AFDX- Header Type, 1,2 = reserved, 3= GNET Frame Header Type. (used for application Program to distinguish between legacy AFDX- Headers and GNET- Headers)
VL-ID	FHW0	15..0	FHW0	15..0	Virtual Link ID of RX Frame
Payload store mode (ADSM: Applied data store mode)	BECW	29..27	BESW	15..14	Which payload store mode is selected for this frame. 0: Received frame completely stored 1: Frame Header, MAC,IP Header and 20 Bytes if IP Payload stored 2: Frame Header, MAC,IP Header stored 3: Frame Header and MAC Header stored other values: Reserved
Start Trigger Flag	BECW	26	BESW	16	Capture asserted Start Trigger condition
AFDX Port Map ID	BECW	23..16	BECW	23..16	User defined Port Number which is associated to the physical port (PortMap ID see **FdxCmdRxPortInit**).
Speed Mode	BECW	13	BECW	15..14	0: 100 Mbps 1: 10 Mbps 2: 1000 Mbps 3: Reserved
SN	FHW2	7..0	BECW	7..0	Received Frame sequence number
Buffer Size	FHW2	31..16	FHW1	10..0	The Buffer Size field indicates the size of the total AFDX-Buffer entry in bytes, including the buffer header (36 Bytes for AFDX and 32 bytes for GNET). Due to the burst capabilities of the Hardware, the AFDX- Buffer entry size is aligned to 64byte blocks for AFDX and is aligned to 32 byte blocks for GNET.
MAC-ID	FHW1	31..29	FHW0	31..29	This field contains the physical Interface_ID of the MAC implemented at the Frontend. 001 Packet received on Network A 010 Packet received on Network B others reserved
IFGE	FHW1	28	FHW1	30	Interframe Gap High
IFG-CNT	FHW1	27..14	FHW1	27..14	Interframe gap counter
BCNT	FHW1	10..0	FHW1	10..0	Byte count. This field contains the number of bytes from the received AFDX- MAC- Frame, including the bytes of the FCS- field at the end of the frame.
Sub-Microseconds			TTHW	23..20	0 to 999
Days of year	TTHW	19..11	TTHW	19..11	1 to 365
Hours of day	TTHW	10..6	TTHW	10..6	0..23
Minutes of hour	TTHW	5..0	TTHW	5..0	0..59
Minutes of hour	TTLW	31..26	TTLW	31..26	0 to 59
Seconds of minute	TTLW	25..20	TTLW	25..20	0 to 59
Microseconds of second	TTLW	19..0	TTLW	19..0	0 to 999.999

Figure 4-9 AFDX Frame Layout

Note: Payload mode defines the AFDX data that is captured.

AFDX- Frame Layout				
	31 24	23 16	15 8	7 0
AFDX Frame	**MAC Destination Address** Byte 2	Byte 3	Byte 4	Byte 5
	Byte 4	Byte 5	Byte 0	Byte 1
	MAC Source Address Byte 0	Byte 1	Byte 2	Byte 3
			Length Byte 0	Byte 1
	IP-Header **20 bytes**			
	Received IP-Payload (802.3 defines: 25 to 1479 bytes but it may be less or more at Frame size violation case)			
	SN			

The following Monitor Queue functions are available to enable you to retrieve Captured frames and examine the status of each individual frame received. As stated above, there is no pre-determined order or method to use these functions as each user's requirements are different.

a. **FdxCmdMonGetStatus** - indicates the state of the Monitor as shown in Figure 4-10 and the number of frames captured since trigger start (Continuous or Record mode only). This information can be used to determine whether you want to Read the Monitor Queue or wait. Below are the possible Monitor states and examples of actions to be taken based on the state.

1. Monitor is **off**

2. Monitor is **waiting** for Trigger

3. Monitor **start trigger** has occurred. Data is being captured.

    - When in Continuous or Record capture mode, you may want to use the Number of Frames captured parameter returned to determine the number of entries to read when using **FdxCmdMonQueueRead**. Save the Frames captured counter and use again after you request status again to determine how many new entries to read in the Monitor Queue.

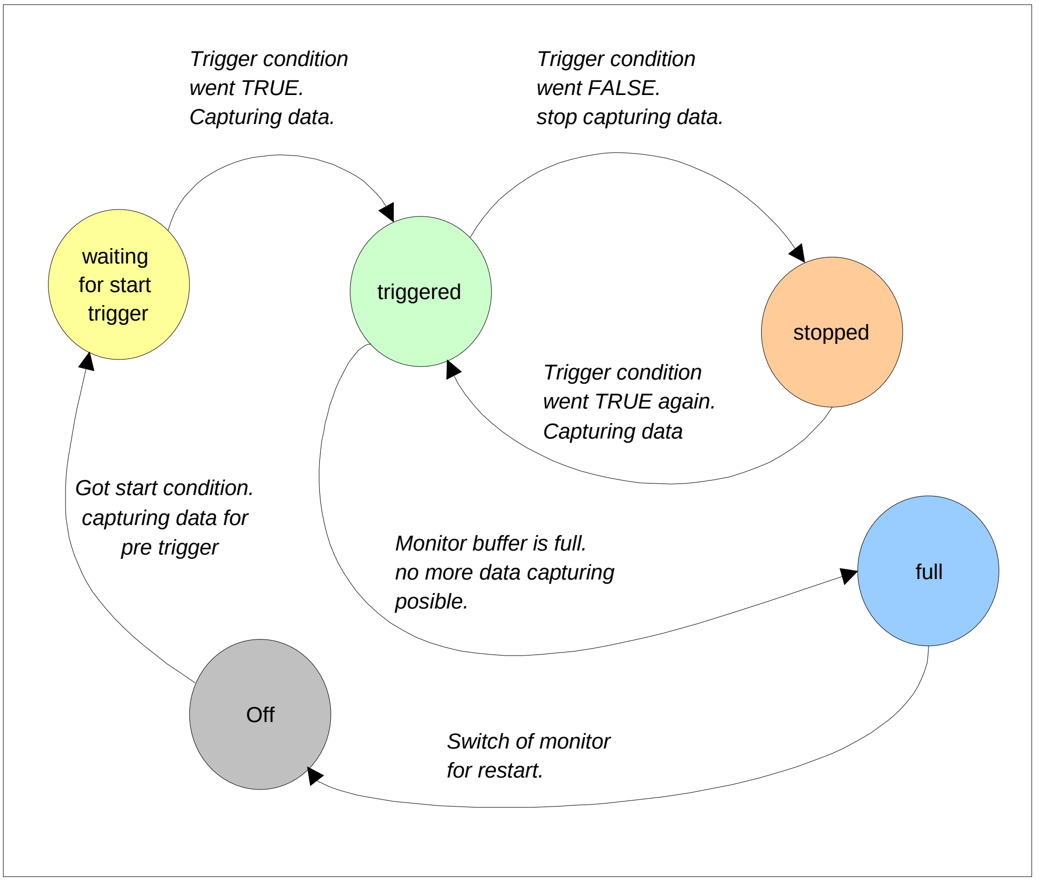
4. Capturing has **stopped -** For SingleShot-Selective Mode

    - If in SingleShot-Selective mode and a transition from Triggered to Stopped has occurred, then at least one frame that meets your trigger conditions defined in the TCB(s) has been captured. You may then want to perform a Monitor Queue read using **FdxCmdMonQueueRead**.

5. Monitor Buffer is **full** and no more data will be captured. This status is only applicable to SingleShot-Standard and SingleShot-Selective capture modes.

    - If in SingleShot-Standard mode you may only want to read the Monitor Queue after it is full. At that time, you can dump the entire Monitor Queue to required memory for display purposes.

Figure 4-10 Capture States


Trigger condition
went TRUE.
Capturing data.
Trigger condition
went FALSE.
stop capturing data.
waiting
for start
trigger
triggered
stopped
Trigger condition
went TRUE again.
Capturing data
Got start condition.
capturing data for
pre trigger
Monitor buffer is full.
no more data capturing
posible.
full
Off
Switch of monitor
for restart.

The following code creates a Monitor Queue, Performs a read of the Monitor Queue, and if the number of entries read is not zero, the frame information is copied to a record file, and information from the frame is printed. The Queue is then deleted - which is required prior to termination of the program.

```
/*
* This example reads one entry from the Monitor Queue
*/
AiUInt32 aul_Data[0x10000];
AiUInt32 ul_QueueId;
TY_FDX_MON_QUEUE_CTRL_IN  x_QueueCtrlIn;
TY_FDX_MON_QUEUE_CTRL_OUT x_QueueCtrlOut;
TY_FDX_MON_QUEUE_READ_IN  x_QueueReadIn;
TY_FDX_MON_QUEUE_READ_OUT x_QueueReadOut;
TY_FDX_FRAME_BUFFER_HEADER* px_FrameBufferHeader;

#define REC_SIZE  0x10000
AiUInt8 ac_RecData[REC_SIZE];
AiUInt32 ul_RecBytes;

/* Create a Queue */
x_QueueCtrlIn.ul_QueueControl = FDX_MON_QUEUE_CREATE;
if (FDX_OK == ( FdxCmdMonQueueControl( ul_Handle, &x_QueueCtrlIn, &x_QueueCtrlOut)))
{
   ul_QueueId = x_QueueCtrlOut.ul_QueueId;
}
else {
   printf("\r\nFdxCmdMonQueueControl() failed.");
   exit (1);  //exit this process - can't do anything without a queue id
}

/* Read a Queue */
x_QueueReadIn.ul_EntryCount    = 1; /*  Read one entry  */
x_QueueReadIn.ul_MaxReadBytes  = sizeof(aul_Data);
x_QueueReadIn.ul_ReadQualifier = FDX_MON_READ_FULL; // Read fixed header + AFDX Frame
x_QueueReadOut.pv_ReadBuffer = aul_Data;
if (FDX_OK != ( FdxCmdMonQueueRead( ul_Handle, ul_QueueId, &x_QueueReadIn,
                                               &x_QueueReadOut)))
{
   printf("\r\nFdxCmdMonQueueRead() failed.");
}

if (x_QueueReadOut.ul_EntryRead > 0)
{
   ul_RecBytes = x_QueueReadOut.ul_BytesRead;
   printf("\n %ld Bytes recorded / Frames read:%ld", ul_RecBytes,
             x_QueueReadOut.ul_EntryRead);

   /* copy all the bytes read for the entry to a record array */

   memcpy(ac_RecData,x_QueueReadOut.pv_ReadBuffer,x_QueueReadOut.ul_BytesRead);

   px_FrameBufferHeader = (TY_FDX_FRAME_BUFFER_HEADER*) x_QueueReadOut.pv_ReadBuffer;

   /* print the sequence number and time tag from the Fixed entry header*/
   printf("\n SN = %08x TtHigh = %08lX TtLo = %08lX ",
            px_FrameBufferHeader->x_FrameHeaderInfo.uc_SequenceNr,
            px_FrameBufferHeader->x_FwIrigTime.ul_TtHigh,
            px_FrameBufferHeader->x_FwIrigTime.ul_TtLow);

   /*  Print the network (A or B) the data was received on which is found in the */
   /*    Frame Header Word 1 of the Fixed Entry Header   **/
   if ( (px_FrameBufferHeader->x_FrameHeaderInfo.ul_FrameHeaderWord_1 & 0x20000000 )
             == 0x20000000)
```

```
      printf("\n Received on Net A");
   else if ( (px_FrameBufferHeader->x_FrameHeaderInfo.ul_FrameHeaderWord_1 &
                  0x40000000 ) == 0x40000000)
      printf("\n Received on Net B");
   else
      printf("\n Network ID wrong");

   /*  Print the first 128 bytes of the AFDX data frame */
   printf("\n Data: 0000: %08lx %08lx %08lx %08lx ", aul_Data[ 0], aul_Data[ 1],
                    aul_Data[ 2], aul_Data[ 3]);
   printf("\n Data: 0010: %08lx %08lx %08lx %08lx ", aul_Data[ 4], aul_Data[ 5],
                    aul_Data[ 6], aul_Data[ 7]);
   printf("\n Data: 0020: %08lx %08lx %08lx %08lx ", aul_Data[ 8], aul_Data[ 9],
                    aul_Data[10], aul_Data[11]);
   printf("\n Data: 0030: %08lx %08lx %08lx %08lx ", aul_Data[12], aul_Data[13],
                    aul_Data[14], aul_Data[15]);
   printf("\n Data: 0030: %08lx %08lx %08lx %08lx ", aul_Data[16], aul_Data[17],
                    aul_Data[18], aul_Data[19]);
   printf("\n Data: 0030: %08lx %08lx %08lx %08lx ", aul_Data[20], aul_Data[21],
                    aul_Data[22], aul_Data[23]);
   printf("\n Data: 0030: %08lx %08lx %08lx %08lx ", aul_Data[24], aul_Data[25],
                    aul_Data[26], aul_Data[27]);
   printf("\n Data: 0030: %08lx %08lx %08lx %08lx \n\n", aul_Data[28], aul_Data[29],
                    aul_Data[30], aul_Data[31]);

}
else
{
   printf("\n No entries available.");
}


printf("\n.Queue Delete....\n");
/* Delete a Queue */
x_QueueCtrlIn.ul_QueueControl = FDX_MON_QUEUE_DELETE;
x_QueueCtrlIn.ul_QueueId = ul_QueueId;
if (FDX_OK != ( FdxCmdMonQueueControl( ul_Handle, &x_QueueCtrlIn, &x_QueueCtrlOut)))
{
   printf("\r\nFdxCmdMonQueueControl() failed.");
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

5 PROGRAM SAMPLES

Within this section, the program samples will be described. There is sample code available with the PCI-FDX BSP. The samples consists of several modules. Each of these modules can be used by program developers as example for learning and developing their code. This section will discuss the following:

a. Overview of Sample Programs

b. Sample Modules:

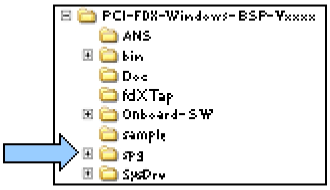
Module	Description
afdx_MainSample.cpp	Sample User Interface, Initialisation
afdx_SystemFunc.cpp	Board level functions, Board Configuration, IRIG
afdx_SampleUtils.cpp	Additional functions (not AFDX specific)
afdx_LogInOut.cpp	Library Administration functions
afdx_GenericRX.cpp	Generic receiver functions.
afdx_GenericTX.cpp	Generic transmitter functions.
afdx_GenRX_CCSE.cpp	Generic receiver functions using continuous capture second edition.
afdx_GenTX_Ext.cpp	Generic transmitter functions using extended generic transmit modes (buffer queues and transmit sub queues).
afdx_InterruptFunc.cpp	Interrupt functions.
afdx_ReplayFunc.cpp	Setup FDX replay mode
afdx_SimulationRX.cpp	Individual receiver functions (UDP and SAP port related)
afdx_SimulationTX.cpp	Individual tranmitter functions (UDP and SAP port related)
afdx_UdpRx.cpp	Setup UDP receiver by using VL and UDP port configuration files (*.csv)
afdx_UdpTx.cpp	Setup UDP tansmitter by using VL and UDP port configuration files (*.csv)

c. Matrix of all API S/W Library Calls vs. Sample Programs

The sample program files contained in the BSP is located in the sample program (spg) file:

`x:\Program Files\AIM GmbH\PCI-FDX-Windows-BSP-Vxxxx\spg`

In order to run a sample project, please refer to the **PCI FDX and fdXTap™ Getting Started Manual**. The **Reference Manual AFDX/ARINC-664** will provide further detail on library calls and parameter naming conventions used within these sample programs.



5.1 Program Samples Overview

Table 5-1-I provides a list and functional description of the sample programs. You may choose to use one or more sample programs as a starter program to be modified.

Table 5-1 Program Samples Overview

afdx_udp_sample01.cpp (Section 5.2.1) - This sample demonstrates how to: 1. Query available resources and login to a local board and two ports. 2. Reset the board and synchronize board-IRIG-time with PC-time. 3. Setup Port 1 for UDP Port-Oriented transmit mode, i.e., VL & UDP Ports are defined. VL traffic shaping is supported in this mode. 4. Setup Port 2 to capture using the VL-Oriented Receive mode i.e. individual buffers for each received VL are provided. 5. Port1 sends data to Port2. *(an ethernet connection between Port 1 and Port 2 is required.)*
afdx_generic_sample01.cpp (Section 5.2.2) - This sample demonstrates how to: 1. Query available resources and login to a local board and two ports. 2. Reset the board and synchronize board-IRIG-time with PC-time. 3. Setup Port 1 for Generic transmit mode i.e., a list of AFDX frames with additional header information can be sent from a specified queue cyclically or a specific number of times 4. Setup Port 2 to capture using the Chronological Monitor Receive mode i.e., Data of all enabled links are stored in one large chronological monitor buffer. 5. Port1 sends data to Port2. *(an ethernet connection between Port 1 and Port 2 is required.)*
afdx_sample.exe (included in the BSP)- This sample program demonstrates API function calls through a user interface as shown in Figure 5-1. (source code included)

Figure 5-1 afdx_Sample.exe User Interface

```
 Testprogramm for AFDX Software==================================================
 AIM, 06.07.2007
Date and Time of creation of sample:Apr  9 2010 14:35:52

 ================================================================================

 FdxInit()

 FdxInit O.K.
Help '?', Abort with 'x'

 Select one of the following Tests:

   ---Board related Tests / Tools---
   'L' Login                      to   BOARD Resource
   'E' Logout                     from BOARD Resource
   'B' Display BSP Versions       from BOARD Resource
   's' Test Board Control              (BOARD Resource required)
   'd' Test Irig Time Setting          (BOARD Resource required)
   'z <Addr>' Read from BIU memory     (Module 0)
   'm <Addr>' Read from Shared memory  (Module 0)
   'I <Addr>' Read from IO memory      (Module 0)
   'J <Addr>' Read from Local memory   (Module 0)
   't' Irig Tools (Add/Sub)
   'r' Replay Test                     (BOARD and PORT Resource required)
   'j' Automatic Rx Test               (BOARD and PORT Resource required)
   'W' BITE Transfer Test              (loop connector required)

   ---Port related Tests---
   'l'   Login                    to   PORT Resource
   'e'   Logout                   from PORT Resource
   'a'   Test Rx Port Control          (PORT Resource required)
   'b'   Test Rx Mode Control          (PORT Resource required)
   'c'   Test Rx Vl Control            (PORT Resource required)
   'f'   Test Trigger Setup            (PORT Resource required)
   'h'   Test Get Status Rx and Mon    (PORT Resource required)
   'i'   Test Capture Control Single   (PORT Resource required)
   'C'   Test Capture Control CCSE     (PORT Resource required)
   'g'   Test Global Statistic         (PORT Resource required)
   'v'   Test Vl Activity              (PORT Resource required)
   'q'   Test Rx Queue                 (PORT Resource required)
   'n'   Set Tx Static Registers       (PORT Resource required)
   'p' <Cycles> 0=Cyclic  Setup Tx Queue,
         Start Tx and get Status       (PORT Resource required)
   'T'   Test Interrupts               (PORT Resource required)

   'u'   UDP related Tests             (PORT Resource required)
   '?' Print this menue
   'x' Exit Test program
>
 FdxQueryServerConfig O.K. Server:local
---ID-Type-Info------(List of available resources)-----------
    1    0 BoardName:AMC-FDX-2 BoardSerialNo:360
    2    1 PortName:Port1 PortNo:1 PortMode:0
    3    1 PortName:Port2 PortNo:2 PortMode:0
    4    0 BoardName:API-FDX-2 V2 BoardSerialNo:563
    5    1 PortName:Port1 PortNo:1 PortMode:0
    6    1 PortName:Port2 PortNo:2 PortMode:0
-(End of resource list)------------------------------------
NOTE: Login to required resources before using functions!!!
>_
```

5.2 Program Sample Code

5.2.1 UDP-Port Oriented Transmission/VL-Oriented Monitor Storage

This sample demonstrates how to:

1. Query available resources and login to a local board and two ports.
2. Reset the board and synchronize board-IRIG-time with PC-time.
3. Setup Port 1 for UDP Port-Oriented transmit mode, i.e., VL & UDP Ports are defined. VL traffic shaping is supported in this mode.
4. Setup Port 2 to capture using the VL-Oriented Receive mode i.e. individual buffers for each received VL are provided.
5. Port1 sends data to Port2. *(an ethernet connection between Port 1 and Port 2 is required.)*

```
// afdx_udp_sample01.cpp
//

#include "stdafx.h"
#include <time.h>


#include "aifdx_def.h"
```

aifdx_def.h *header file is the only header file required for inclusion to support the AIM API. It contains the constant, structure and function definitions used in the API.*

```
//--------------------------------------------------------------
// Defines
//--------------------------------------------------------------

   // communication parameters
#define DEF_VL                      (60)
#define DEF_SUB_VLCNT               (2)
#define DEF_SUB_VLID1               (1)
#define DEF_SUB_VLID2               (2)
#define DEF_SRC_MAC_LSLW            (0x00012120)
#define DEF_SRC_MAC_MSLW            (0x00000200)
#define DEF_SRC_IP                  (0x0a012101)
#define DEF_DST_IP                  (0xe0e0003c)
#define DEF_SRC_UDP1                (24)
#define DEF_DST_UDP1                (23)
#define DEF_SRC_UDP2                (33)
#define DEF_DST_UDP2                (34)
#define DEF_BAG                     (50)
#define DEF_FRAME_MAXLENGTH         (1518)
#define DEF_UDP_MAXMSG              (500)
#define DEF_UDP_SAMPLING_RATE       (100)
```

```
//-------------------------------------------------------------
// Function-Declarations
//-------------------------------------------------------------
bool MyFdxInit();
void MyFdxFreeResources();
void MyFdxResetBoard();
void MyFdxResetPort();
void MyFdxSetupTxPort();
void MyFdxSetupRxPort();
void MyFdxStartTx();
void MyFdxStartRx();
void MyFdxStopTx();
void MyFdxStopRx();
void MyFdxGetStatus();
void MyFdxGetVLActivity();
```

Function defintions used for functions defined within this program.

```
//-------------------------------------------------------------
// Globals
//-------------------------------------------------------------
AiUInt32            g_ulBoardHandle  = 0;
AiUInt32            g_ulPort1Handle  = 0;   // acts as Tx
AiUInt32            g_ulPort2Handle  = 0;   // acts as Rx
AiUInt32            g_pUdp1Port1Handle = 0;
AiUInt32            g_pUdp2Port1Handle = 0;
AiUInt32            g_pUdp1Port2Handle = 0;
AiUInt32            g_pUdp2Port2Handle = 0;
char           g_stop;
```

The main program provides an overview of the order of any basic application program:

Initialization--->Board setup--->Port Setup--->Frame-Data Setup for Tx---> Monitor Setup for Rx--->StartTx/Rx

After starting Tx/Rx - MyFdxGetStatus provides user controlled action for:

1. *Check Tx Status*
2. *Check Rx Status*
3. *Exit*

Each local function contains the API function calls required to perform these capabilities.

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

   /*--- init application interface, query resources on local server and login to get valid
   handles */
   printf("\n");
   printf("Performing API initialization and local resource login..\n");
   if (MyFdxInit())
   {
      printf("API initialized, Login successful\n");

      //--- reset
      printf("\nPerforming board-reset and syncronizing IRIG to PC-Time...\n");
      MyFdxResetBoard();

      printf("\nPerforming port-resets...\n");
      MyFdxResetPort();

      printf("Press Any key to continue\n");
      getchar();
      //--- setup
      printf("\nPerforming Tranmitter setup on Port 1...\n");
      MyFdxSetupTxPort();

      printf("\nPerforming Receiver setup on Port 2...\n");
      MyFdxSetupRxPort();

      //--- Start Receiver
      printf("\nPerforming Receiver startup...\n");
      MyFdxStartRx();

      //--- Start Transmitter
      printf("\nPerforming Transmitter startup...\n");
      MyFdxStartTx();

   }
   else
   {
      printf("API Open Failure!!!\n");
   }

   MyFdxGetStatus();

   return 0;
}
```

```
//----------------------------------------------------------------
// MyFdxInit - returns true on success
//----------------------------------------------------------------
//    Init the application interface and
//    gets the global handles to local resources
//----------------------------------------------------------------
bool MyFdxInit()
{
   DWORD                          dwTmp;
   bool                           bRetSuccess = false;
   bool                           bFoundLocalServer = false;
   TY_SERVER_LIST *               px_ServerNames = NULL;
   TY_SERVER_LIST *               px_TmpServer;
   TY_RESOURCE_LIST_ELEMENT *     pRLE = NULL;
   TY_RESOURCE_LIST_ELEMENT *     pRLEHead = NULL;
   TY_FDX_CLIENT_INFO             x_ClientInfo;

   //--- init client-info
   sprintf(x_ClientInfo.ac_ClApplication, "AFDX-Sample Application");
   sprintf(x_ClientInfo.ac_ClApplicationVersion, "1.0");
   dwTmp = MAX_FDX_CLIENT_HOST_NAME;
   ::GetComputerName((LPWSTR)(x_ClientInfo.ac_ClHostName), &dwTmp);
   dwTmp = MAX_FDX_CLIENT_USER_NAME;
   ::GetUserName((LPWSTR)(x_ClientInfo.ac_ClUser), &dwTmp);
```

*The **first** local function (called by the main program) to be executed performs:*
(1) Initialization of the API
(2) Board login
(3) Port(s) login.

***GetComputerName/GetUserName** are MSWindows functions that retrieve the computer name/user name of the current system. These values are used for the **FdxLogin** function.*

```
   //--- application interface initialize
   // and get a list of available servers
   if (FdxInit(&px_ServerNames) != FDX_OK)
   {

      printf("API Open Failed!!!\n");
      // free the server-list
      if (px_ServerNames != NULL)
      {
         FdxCmdFreeMemory(px_ServerNames, px_ServerNames->ul_StructId);
      }
      return(bRetSuccess);
   }

   // search the server-list for local server
   px_TmpServer = px_ServerNames;
   while ((px_TmpServer != NULL) && (!bFoundLocalServer))
   {
      if (stricmp(px_TmpServer->auc_ServerName, "local") == 0)
      {
         bFoundLocalServer = true;
      }
      else
      {
         px_TmpServer = px_TmpServer->px_Next;
      }
   }

   if (bFoundLocalServer)
   {  // ok, we found a local server
      // lets query the configuration of this server
      if (FdxQueryServerConfig("local", &pRLEHead) == FDX_OK)
      {
         pRLE = pRLEHead;
         while (pRLE != NULL)
         {  //--- login to resources
            switch(pRLE->ul_ResourceType)
            {
```

***FdxInit** returns the names of available servers at px_ServerNames. If px_ServerNames = "local", the AFDX board is located where the API is running. If px_ServerNames = "NULL", the end of the list has been reached. **Note**: this version will only return "local".*

*If a local server was found then the local server is searched for available AFDX boards, using **FdxQueryServerConfig**.*

***FdxQueryServerConfig** will return a list of resouces (board and port) available which will be used as an input for **FdxLogin**. The following code assumes an FDX-2 board configuration, thus only logging into one board and two ports.*

```
case RESOURCETYPE_BOARD:
   if (g_ulBoardHandle == 0)
   {
```

```
               if (FdxLogin("local", &x_ClientInfo, pRLE->ul_ResourceID,
                               PRIVILEGES_ADMIN, &g_ulBoardHandle) != FDX_OK)
               {
                  g_ulBoardHandle = 0;
                  printf("Board Login Failure!!!\n");
               }
            }
         break;

         case RESOURCETYPE_PORT:
            if (g_ulPort1Handle == 0)
            {
               if (FdxLogin("local", &x_ClientInfo, pRLE->ul_ResourceID,
                               PRIVILEGES_ADMIN, &g_ulPort1Handle) != FDX_OK)
               {
                  g_ulPort1Handle = 0;
                  printf("Port 1 Login Failure!!!\n");
               }
            }
            else if (g_ulPort2Handle == 0)
            {
               if (FdxLogin("local", &x_ClientInfo, pRLE->ul_ResourceID,
                               PRIVILEGES_ADMIN, &g_ulPort2Handle) != FDX_OK)
               {
                  g_ulPort2Handle = 0;
                  printf("Port 2 Login Failure!!!\n");
               }
            }
         break;
         }

         pRLE = pRLE->px_Next;
      }

   }

   }
   // free the resource-list
   if (pRLEHead != NULL)
   {
      FdxCmdFreeMemory(pRLEHead, pRLEHead->ul_StructId);
   }

   // free the server-list
   if (px_ServerNames != NULL)
   {
      FdxCmdFreeMemory(px_ServerNames, px_ServerNames->ul_StructId);
   }

   // we define it as success if we have valid handles for all global variables....
   bRetSuccess = (g_ulBoardHandle != 0) && (g_ulPort1Handle != 0) && (g_ulPort2Handle != 0);

   return bRetSuccess;
}
```

__FdxLogin__ returns the handle for the board or port resource.

The memory allocated when either resources were found using __FdxQueryServerConfig__, or server was found using __FdxInit__, must be released prior to termination of the program.

```
//---------------------------------------------------------------
// MyFdxResetBoard
//---------------------------------------------------------------
void MyFdxResetBoard()
{
   int i;
   AiUInt32                 ul_Mode;
   time_t                   loc_time;
   struct tm *              ptm;
   TY_FDX_BOARD_CTRL_IN     x_BoardCtrlIn;
   TY_FDX_BOARD_CTRL_OUT    x_BoardCtrlOut;
```

*The **second** local function (called by the main program) demonstrates two **System (Board-level) functions** to:*

*(1) Configure each **port** as either **single or redundant**, and set up the **network bit rate** (default, as in this case, is 100 Mbps)*

(2) Initialize the internal Board IRIG time.

```
TY_FDX_IRIG_TIME        x_IrigTime;

memset(&x_IrigTime, 0, sizeof(x_IrigTime));
memset(&x_BoardCtrlIn, 0, sizeof(x_BoardCtrlIn));
memset(&x_BoardCtrlOut, 0, sizeof(x_BoardCtrlOut));

if (g_ulBoardHandle > 0)
{
   //--- init input structure
   for (i=0; i<FDX_MAX_BOARD_PORTS; i++)
   {
      x_BoardCtrlIn.aul_PortConfig[i] = FDX_SINGLE;
      x_BoardCtrlIn.aul_ExpertMode[i] = FDX_EXPERT_MODE;
   }
   x_BoardCtrlIn.ul_RxVeriMode = FDX_BOARD_VERIFICATION_TYPE_DEFAULT;


   //--- reset board
   if (FDX_OK != (FdxCmdBoardControl(g_ulBoardHandle,
                     FDX_WRITE, &x_BoardCtrlIn, &x_BoardCtrlOut)))
      for (i=0; i<FDX_MAX_BOARD_PORTS; i++)
      {
         x_BoardCtrlIn.aul_PortConfig[i] = FDX_SINGLE;
                  x_BoardCtrlIn.aul_ExpertMode[i] = FDX_EXPERT_MODE;
      }
      x_BoardCtrlIn.ul_RxVeriMode = FDX_BOARD_VERIFICATION_TYPE_DEFAULT;

      //--- reset board
      if (FDX_OK != (FdxCmdBoardControl(g_ulBoardHandle,
                     FDX_WRITE, &x_BoardCtrlIn, &x_BoardCtrlOut)))
      {
         printf("Board Reset Failure!!!\n");
      }
      else
      {
         printf("Board Initialized\n");
      }

      //--- sync board-internal irigtime-source with PC-time
      loc_time     = time(NULL);
      ptm          = localtime(&loc_time);
      if (ptm != NULL)
      {
         x_IrigTime.ul_Day      = ptm->tm_yday + 1;   /* tm.YearOfDay is ZeroBased but we are
                                                          OneBased */
         x_IrigTime.ul_Hour     = ptm->tm_hour;
         x_IrigTime.ul_Min      = ptm->tm_min;
         x_IrigTime.ul_Second   = ptm->tm_sec;
         x_IrigTime.ul_MilliSec = 0;
         x_IrigTime.ul_MicroSec = 0;
```

*Using **FdxCmdBoardControl**, each **port** is configured as a **single port** and the **network bit rate** is set to the default 100 Mbps. Verification mode is set to default which means the firmware will determine the program-specific board in use and setup the verification register accordingly.*

*Using **FdxCmdIrigTimeControl**, the Board internal IRIG time can be synchronized to any time required by your application, in this case it is synched to the local PC time.*

Note: An external IRIG source can be used. In that case, initializing the internal IRIG time would not be applicable.

```
         if (FDX_OK != (FdxCmdIrigTimeControl(g_ulBoardHandle, FDX_IRIG_WRITE, &x_IrigTime,
                           &ul_Mode)))
         {
            printf("IRIG sync failure!!!\n");
         }
         else
         {
            printf("IRIG sync successful\n");
         }
      }
   }
}

//----------------------------------------------------
```

*The **third** local function (called by the main program) demonstrates **Port-level Transmitter and Receiver initialization**:*

The user must also assign a PortMap ID to each Tx/Rx port. This Port Map ID is a virtual ID assigned to the physical Port.

(a) Using multiple AFDX cards
(b) Using receive ports in redundant mode

```
// MyFdxResetPort
//---------------------------------------------------------------
void MyFdxResetPort()
{
   TY_FDX_PORT_INIT_IN  x_PortInitIn;
   TY_FDX_PORT_INIT_OUT x_PortInitOut;

   if (g_ulPort1Handle > 0)
   {
      x_PortInitIn.ul_PortMap = 1;
      if (FDX_OK != (FdxCmdTxPortInit(g_ulPort1Handle, &x_PortInitIn, &x_PortInitOut)))
      {
         printf("Port 1 Reset failure!!!\n");
      }
      else
      {
         printf("Port 1 Transmitter Initialized\n");
      }
   }
```

The initialized state after ***FdxCmdTxPortInit*** *is performed includes:*

(1) No Transmit Queues defined
(2) No VL created, No UPD Ports created
(3) FdxCmdTxControl command has no effect

```
   if (g_ulPort2Handle > 0)
   {
      x_PortInitIn.ul_PortMap = 2;
      if (FDX_ERR == FdxCmdRxPortInit(g_ulPort2Handle, &x_PortInitIn, &x_PortInitOut))
      {
         printf("Port 2 Reset failure!!!\n");
      }
      else
      {
         printf("Port 2 Receiver Initialized\n");
      }
   }
}
```

The initialized state after ***FdxCmdRxPortInit*** *is performed includes:*

(1) Global Statistics available
(2) All Virtual Links, enabled for Activity information
(3) Chronological Receive Mode, No VLs enabled for capturing
(4) No Trigger Control Block Processing Enabled

```
//---------------------------------------------------------
// MyFdxSetupTxPort
//---------------------------------------------------------
void MyFdxSetupTxPort()
{
   TY_FDX_TX_MODE_CTRL      x_TxModeCtrl;
   TY_FDX_TRANSMIT_VL       x_TxVL;
   TY_FDX_UDP_DESCRIPTION   x_UdpDesc;
   AiUInt32 ul_BytesWritten;
   AiUInt32 uiBufLen;
   char Buf[512];
```

This local function is called by the main program. Now that the board and ports have been initialized we can setup Port 1 to transmit data as follows:

(1) Individual / UDP Port-Oriented Transmit Mode
(1) Define the VL & Sub VL characteristics
(2) Write UDP port messages created toTx port

Two Sampling ports are setup to transmit data every 100 milliseconds.

Individual/UDP-Port Oriented mode *- this mode simulates the AFDX Comm Ports (defined by ARINC-653) including:*

Queuing Ports *- AFDX messages are sent over several AFDX frames (fragmentation by IP layer), no data is lost or overwritten.*
Sampling Ports *- AFDX messages are sent in 1 frame, data may be lost or overwritten.*

The end-systems, VLs, and partitions are represented by the IP-Addresses and communication-end points are described by the UDP-Port.

```
//--- mode control -> individual/UDP-Port oriented
x_TxModeCtrl.ul_TransmitMode = FDX_TX_INDIVIDUAL;
if (FDX_OK != (FdxCmdTxModeControl(g_ulPort1Handle, &x_TxModeCtrl)))
{
   printf("Port 1 Mode Control Failure!!!\n");
}
else
{
   printf("Port 1 set to individual/UDP-oriented Transmit mode\n");
}
```

After setting up the mode, you then need to define the characteristics of the VL.

VL-Definitions are identified by the VL-ID. The MAC address, BAG and the maximum frame length are properties of the VL-Definition. The VL-Definition is the "parent" of a set of up to 4 S/Q-Ports (identified by Sub VL ID (1-4)). So if the VL-Definition is disabled/deleted all S/Q-Ports of this VL are disabled/deleted.

```
//--- create vl, define communication parameters for VL 60 or Port 1
x_TxVL.ul_Bag            = DEF_BAG;            //Bag
x_TxVL.ul_MACSourceLSLW = DEF_SRC_MAC_LSLW;    //MAC Source
x_TxVL.ul_MACSourceMSLW = DEF_SRC_MAC_MSLW;    //MAC Source
x_TxVL.ul_MaxFrameLength= DEF_FRAME_MAXLENGTH;//Maximum Frame Length
x_TxVL.ul_SubVls         = DEF_SUB_VLCNT;      // # of Sub VLs
x_TxVL.ul_VlId           = DEF_VL;             // VL
x_TxVL.ul_FrameBufferSize = 0;

if (FDX_OK != (FdxCmdTxCreateVL(g_ulPort1Handle, &x_TxVL)))
{
   printf("VL Creation on Port 1 failed!!!\n");
}
else
{
   printf("VL 60 Created on Port 1\n");
}
```

Thr VL ID defined for Port 1 is 60 with two Sub VLs.

After setting up the VL, you then need to define the characteristics of the Sub VLS (individual S/Q port). There can be up to 4 S/Q Ports per VL ID.

SubVL-Definitions are identified by the Sub VL-ID. The address-quintuplet (UDP, IP, VL ID) message size and the sampling rate length are properties of the Sub VL-Definition.

This Sub VL is defined as follows:
- Sub VL ID = 1
- Sampling Port
- number of messages = 1 (for sampling port always equal to one.)
- sampling rate = 100 milliseconds
- DEF_SRC_IP (0x0a012101)
- DEF_DST_IP (0xe0e0003c)
- DEF_SRC_UDP1 (24)

```
//--- create udp-port 1 for write on Port 1
x_UdpDesc.ul_PortType          = FDX_UDP_SAMPLING;
x_UdpDesc.x_Quint.ul_IpDst     = DEF_DST_IP;
x_UdpDesc.x_Quint.ul_IpSrc     = DEF_SRC_IP;
x_UdpDesc.x_Quint.ul_UdpDst    = DEF_DST_UDP1;
x_UdpDesc.x_Quint.ul_UdpSrc    = DEF_SRC_UDP1;
x_UdpDesc.x_Quint.ul_VlId      = DEF_VL;
x_UdpDesc.ul_SubVlId           = DEF_SUB_VLID1;
x_UdpDesc.ul_UdpNumBufMessages= 1; // 0=default
x_UdpDesc.ul_UdpMaxMessageSize= DEF_UDP_MAXMSG;
x_UdpDesc.ul_UdpSamplingRate  = DEF_UDP_SAMPLING_RATE;
if (FDX_OK != (FdxCmdTxUDPCreatePort(g_ulPort1Handle, &x_UdpDesc, &g_pUdp1Port1Handle)))
{
```

```
      printf("UDP Port Creation Failure on Port 1!!!\n");
   }
   else
   {
      printf("Tx UDP Port Created on Port 1 -- VL:%d UDP Port:%d\n", DEF_VL, DEF_SRC_UDP1);
   }
```

This Sub VL is defined as follows:
- Sub VL ID = 2
- Sampling Port
- number of messages = 1 (for sampling port
- always equal to one.)
- sampling rate = 100 milliseconds
- DEF_SRC_IP (0x0a012101)
- DEF_DST_IP (0xe0e0003c)
- DEF_SRC_UDP2 (33)
- DEF_DST_UDP2 (34)

```
//--- create udp-port 2 for write on Port 1
   x_UdpDesc.ul_PortType            = FDX_UDP_SAMPLING;
   x_UdpDesc.x_Quint.ul_IpDst       = DEF_DST_IP;
   x_UdpDesc.x_Quint.ul_IpSrc       = DEF_SRC_IP;
   x_UdpDesc.x_Quint.ul_UdpDst      = DEF_DST_UDP2;
   x_UdpDesc.x_Quint.ul_UdpSrc      = DEF_SRC_UDP2;
   x_UdpDesc.x_Quint.ul_VlId        = DEF_VL;
   x_UdpDesc.ul_SubVlId             = DEF_SUB_VLID2;
   x_UdpDesc.ul_UdpNumBufMessages= 1; // 0=default
   x_UdpDesc.ul_UdpMaxMessageSize= DEF_UDP_MAXMSG;
   x_UdpDesc.ul_UdpSamplingRate  = DEF_UDP_SAMPLING_RATE;
   if (FDX_OK != (FdxCmdTxUDPCreatePort(g_ulPort1Handle, &x_UdpDesc, &g_pUdp2Port1Handle)))
   {
      printf("UDP Port Creation Failure on Port 1!!!\n");
   }
   else
   {
      printf("Tx UDP Port Created on Port 1 -- VL:%d UDP Port:%d\n", DEF_VL, DEF_SRC_UDP2);
   }

   //Write message to UDP Tx Port
   if (g_pUdp1Port1Handle != NULL) {
      sprintf(Buf, "Testing UDP Port");
      uiBufLen = (AiUInt32)strlen(Buf);
```

Now write UDP port 1 message created above

```
      if (FDX_OK != (FdxCmdTxUDPWrite(g_ulPort1Handle, g_pUdp1Port1Handle, uiBufLen,
            (const void *) Buf, &ul_BytesWritten))) {
         printf("UDP Transmit Port Write failure!!!\n");
      }
      else {
         printf("%d bytes written to UDP Port -- VL:%d UPD Port:%d\n", ul_BytesWritten,
            DEF_VL, DEF_SRC_UDP1);
      }
   }
   //Write message to UDP Tx Port
   if (g_pUdp2Port1Handle != NULL) {
      sprintf(Buf, "Testing UDP Port");
      uiBufLen = (AiUInt32)strlen(Buf);
```

Now write UDP port 2 message created above

```
      if (FDX_OK != (FdxCmdTxUDPWrite(g_ulPort1Handle, g_pUdp2Port1Handle, uiBufLen, (const void
*) Buf, &ul_BytesWritten))) {
         printf("UDP Transmit Port Write failure!!!\n");
      }
      else {
         printf("%d bytes written to UDP Port -- VL:%d UDP Port:%d\n", ul_BytesWritten, DEF_VL,
                     DEF_SRC_UDP2);
      }
   }
}

//-----------------------------------------------------
// MyFdxSetupRxPort
//-----------------------------------------------------
void MyFdxSetupRxPort()
{
   TY_FDX_RX_MODE_CTRL_IN   x_ModeCtrlIn;
   TY_FDX_RX_MODE_CTRL_OUT x_ModeCtrlOut;
```

This local function (called by the main program) will configure the Receive Port to capture the data transmitted by Port1. (Assuming the appropriate ethernet connection has been configured between ports 1 and 2). Port 2 will be setup as follows:

(1) VL-Oriented Receive Mode
(2) Continuous Capture
(3) Create monitor queue to receive the captured data.

```
TY_FDX_RX_VL_CTRL        x_VLControl;
TY_FDX_RX_VL_DESCRIPTION x_VLDesc;
TY_FDX_UDP_DESCRIPTION   x_UdpDesc;

// initialize structures
memset(&x_VLControl,0,sizeof(TY_FDX_RX_VL_CTRL));
memset(&x_VLDesc,0,sizeof(TY_FDX_RX_VL_DESCRIPTION));

//--- mode control -> select VL-Oriented receive
x_ModeCtrlIn.ul_ReceiveMode = FDX_RX_VL;

if (FDX_OK != (FdxCmdRxModeControl(g_ulPort2Handle, &x_ModeCtrlIn, &x_ModeCtrlOut))) {
   printf("Port 2 Mode Control Failure!!!\n");
}
else {
   printf("Port 2 set to VL/UDP-oriented Receive mode\n");
}
```

Since we are in VL-Oriented receive mode, we need to tell port 2 what VL characteristics to look for. These characteristics will be used as a filter, and only data matching those characteristics will be stored in the VL-Oriented receive buffer. Filter characteristics include:

- VL ID
- number of VLs affected by these settings (starting with the VL ID above)

```
//--- VL control (per VL which we want to watch)
x_VLControl.ul_VLId            = DEF_VL;
x_VLControl.ul_VLRange         = 1;
x_VLControl.ul_EnableMode      = FDX_RX_VL_ENA_EXT;
x_VLControl.ul_PayloadMode     = FDX_PAYLOAD_FULL;
x_VLControl.ul_TCBIndex        = 0;

x_VLDesc.ul_VerificationMode  = FDX_RX_VL_CHECK_DISA;
x_VLDesc.ul_VLBufSize         = 0x8000;

if (FDX_OK != (FdxCmdRxVLControl(g_ulPort2Handle, &x_VLControl, &x_VLDesc)))
{
   printf("Receive VL Control Failure!!!\n");
}
else
{
   printf("VL:%d Enabled for Capturing on Port 2\n", DEF_VL);
}
```

Now we need to setup the receive UDP port1
- Sampling Port
- address quintuplet:
- *- DEF_SRC_IP (0x0a012101)*
- *- DEF_DST_IP (0xe0e0003c)*
- *- DEF_SRC_UDP1 (24)*
- *- DEF_DST_UDP1 (23)*

- number of messages = 1 (for sampling port - always equal to one.)

```
//--- create udp-port for read
x_UdpDesc.ul_PortType          = FDX_UDP_SAMPLING;
x_UdpDesc.x_Quint.ul_IpDst     = DEF_DST_IP;
x_UdpDesc.x_Quint.ul_IpSrc     = DEF_SRC_IP;
x_UdpDesc.x_Quint.ul_UdpDst    = DEF_DST_UDP1;
x_UdpDesc.x_Quint.ul_UdpSrc    = DEF_SRC_UDP1;
x_UdpDesc.x_Quint.ul_VlId      = DEF_VL;
x_UdpDesc.ul_UdpNumBufMessages= 1; // 0=default
x_UdpDesc.ul_UdpMaxMessageSize= DEF_UDP_MAXMSG;

if (FDX_OK != FdxCmdRxUDPCreatePort(g_ulPort2Handle, &x_UdpDesc, &g_pUdp1Port2Handle))
{
   printf("Receive UDP Port Creation Failure!!!n");
}
else
{
   printf("Rx UDP Port Created on Port 2 -- VL:%d UDP Port:%d\n", DEF_VL, DEF_DST_UDP1);
}
```

Now we need to setup the receive UDP port2
- *Sampling Port*
- *address quintuplet:*
  - *DEF_SRC_IP (0x0a012101)*
  - *DEF_DST_IP (0xe0e0003c)*
  - *DEF_SRC_UDP2 (33)*
  - *DEF_DST_UDP2 (34)*

- *number of messages = 1 (for sampling port - always equal to one.)*

```
   //--- create udp-port for read
   x_UdpDesc.ul_PortType          = FDX_UDP_SAMPLING;
   x_UdpDesc.x_Quint.ul_IpDst     = DEF_DST_IP;
   x_UdpDesc.x_Quint.ul_IpSrc     = DEF_SRC_IP;
   x_UdpDesc.x_Quint.ul_UdpDst    = DEF_DST_UDP2;
   x_UdpDesc.x_Quint.ul_UdpSrc    = DEF_SRC_UDP2;
   x_UdpDesc.x_Quint.ul_VlId      = DEF_VL;
   x_UdpDesc.ul_UdpNumBufMessages= 1; // 0=default
   x_UdpDesc.ul_UdpMaxMessageSize= DEF_UDP_MAXMSG;

   if (FDX_OK != FdxCmdRxUDPCreatePort(g_ulPort2Handle, &x_UdpDesc, &g_pUdp2Port2Handle)) {
      printf("Receive UDP Port Creation Failure!!!n");
   }
   else{
      printf("Rx UDP Port Created on Port 2 -- VL:%d UDP Port:%d\n", DEF_VL, DEF_DST_UDP2);
   }
}

//---------------------------------------------------------------
// MyFdxStartTx
//---------------------------------------------
void MyFdxStartTx()
{
   TY_FDX_TX_CTRL x_TxControl;

   x_TxControl.ul_Count = 0;
   x_TxControl.e_StartMode = FDX_START;

   if (g_ulPort1Handle != NULL)
   {
      if (FDX_OK != (FdxCmdTxControl(g_ulPort1Handle, &x_TxControl))) {
         printf("Failure to start transmitter\n");
      }
      else {
         printf("Transmitter started\n");
      }
   }
}
```

This local function (called by the main program) will start the transmission of AFDX frames via Port1. Send configuration includes:

(1) Send the AFDX frame cyclically (ul_Count = 0)
(2) Setup to start immediately (vs. wait for trigger)
.

```
//----------------------------------------------------
// MyFdxReceive
//----------------------------------------------------
void MyFdxStartRx()
{
   TY_FDX_RX_CTRL x_RxControl;
   if (g_ulPort2Handle != NULL)
   {
      x_RxControl.ul_StartMode            = FDX_START;
      x_RxControl.ul_GlobalStatisticReset = FDX_RX_GS_RES_ALL_CNT;

      if (FDX_OK != (FdxCmdRxControl(g_ulPort2Handle, &x_RxControl))) {
         printf("Failure to start Receiver!!!\n");
      }
      else {
         printf("Receiver Started\n");
      }
   }
}
```

This local function (called by the main program) will start the reception of AFDX frames via Port2. Receive configuration includes:

(1) Receive start
(2) Reset all counters prior to receive start

```
//------------------------------------------------------------------
// MyFdxStopTx
//--------------------------------------------
void MyFdxStopTx()
{
  TY_FDX_TX_CTRL x_TxControl;

  x_TxControl.ul_Count = 0;
  x_TxControl.e_StartMode = FDX_STOP;


  if (FDX_ERR == FdxCmdTxControl(g_ulPort1Handle, &x_TxControl))
  {
    printf("FdxCmdTxControl Error");
  }
}
```

This local function (called by `MyFdxGetStatus)` *will stop the transmission of AFDX frames via Port1.*

```
//------------------------------------------------------------------
// MyFdxStopRx
//--------------------------------------------
void MyFdxStopRx()
{
  TY_FDX_RX_CTRL x_RxControl;

  x_RxControl.ul_StartMode = FDX_STOP;
  x_RxControl.ul_GlobalStatisticReset = FDX_RX_GS_RES_ALL_CNT;

  if (FDX_OK != (FdxCmdRxControl(g_ulPort2Handle, &x_RxControl)))
  {
     printf("FdxCmdRxControl Error");
  }
}
```

This local function (called by `MyFdxGetStatus)` *will stop the reception of AFDX frames via Port2.*

```
//--------------------------------------------
// MyFdxGetStatus
//--------------------------------------------
void MyFdxGetStatus()
{
   char l_command[10];
   bool l_continue = TRUE;
   TY_FDX_TX_STATUS x_TxStatus;
   TY_FDX_TX_UDP_STATUS x_UdpTxStatus;
   TY_FDX_RX_STATUS x_RxStatus;
   TY_FDX_RX_UDP_STATUS x_UdpRxStatus;
   AiUInt32 ul_Control;
   TY_FDX_RX_GLOB_STAT x_GlobalStatisticA, x_GlobalStatisticB;

   while (l_continue == TRUE)
   {
      printf("\r\n '1'  Get Transmitter Status\n");
      printf(" '2'  Get Receiver Status\n");
      printf(" 'x'  Exit\n");
      printf("Select a Command: ");

      scanf("%s", l_command);

      switch (l_command[0])
      {
      case '1':
         {
            // Retrieve Transmitter Status
            printf("\nTransmitter Status:\n");

            if (FDX_OK != (FdxCmdTxStatus(g_ulPort1Handle, &x_TxStatus)))
            {
               printf("FdxCmdTxStatus Error\n");
            }
```

This local function (called from the main program) allows the user to select the action to be taken by the program including:

1 - Get Tranmsmitter Status
2 - Get Receiver Status
x - Exit the program

1 - Get Tranmsmitter Status

```
        printf("Port 1 Status: ");
        switch (x_TxStatus.e_Status)
        {
        case FDX_STAT_STOP:
           printf("Stopped\n");
           break;
        case FDX_STAT_RUN:
           printf("Running\n");
           break;
        case FDX_STAT_ERROR:
           printf("Error\n");
        }

        if (FDX_OK != (FdxCmdTxUDPGetStatus(g_ulPort1Handle,
              g_pUdp1Port1Handle, &x_UdpTxStatus)))
        {
           printf("FdxCmdTxUDPGetStatus Error\n");
        }

        printf("UDP Message count(VL:%d UDP Port:%d): %d\n", DEF_VL,
           DEF_SRC_UDP1, x_UdpTxStatus.ul_MsgCount);


        if (FDX_OK != (FdxCmdTxUDPGetStatus(g_ulPort1Handle,
           g_pUdp2Port1Handle, &x_UdpTxStatus)))
        {
           printf("FdxCmdTxUDPGetStatus Error\n");
        }

        printf("UDP Message count(VL:%d UDP Port:%d): %d\n", DEF_VL,
           DEF_SRC_UDP2, x_UdpTxStatus.ul_MsgCount);

        break;
     }

  case '2':
     {
        // Retrieve Receiver Status
        printf("\nReceiver Status:\n");

        if (FDX_OK != (FdxCmdRxStatus(g_ulPort2Handle, &x_RxStatus)))
        {
           printf("FdxCmdRxStatus Error\n");
        }

        printf("Port 2 Status: ");
        switch (x_RxStatus.ul_Status)
        {
        case FDX_STAT_STOP:
           printf("Stopped\n");
           break;
        case FDX_STAT_RUN:
           printf("Running\n");
           break;
        case FDX_STAT_ERROR:
           printf("Error\n");
           break;
        }

        ul_Control = FDX_RX_GS_RES_NO_CNT;
        if (FDX_OK != (FdxCmdRxGlobalStatistics(g_ulPort2Handle,
           ul_Control, &x_GlobalStatisticA, &x_GlobalStatisticB)))
        {
           printf("\nFdxCmdRxGlobalStatistics Error");
        }

        printf("Port 2 Global Statistics:\n");
        printf("Good Frame Count: %d\n",
           x_GlobalStatisticA.ul_FrameGoodCount);
```

1 - Get UDP Port1 Status

1 - Get UDP Port2 Status

2 - Get Receiver Status

Local function call `MyFdxGetVLActivity` *will retrieve the frame count for the number of active virtual links.*

```
            printf("Bad Frame Count:  %d\n",
               x_GlobalStatisticA.ul_FrameErrorCount);


            //--- Get VL Activity
            MyFdxGetVLActivity();

            //--- Get UDP port Status
            if (FDX_OK != (FdxCmdRxUDPGetStatus(g_ulPort2Handle,
                  g_pUdp1Port2Handle, &x_UdpRxStatus)))
            {
               printf("FdxCmdRxUDPGetStatus Error");
            }

            printf("UDP Port 1 Message Count: %d\n", x_UdpRxStatus.ul_MsgCount);
            printf("UDP Port 1 Error Count: %d\n",   x_UdpRxStatus.ul_MsgErrorCount);


            //--- Get UDP port Status
            if (FDX_OK != (FdxCmdRxUDPGetStatus(g_ulPort2Handle,
                  g_pUdp2Port2Handle, &x_UdpRxStatus)))
            {
               printf("FdxCmdRxUDPGetStatus Error");
            }

            printf("UDP Port 2 Message Count: %d\n", x_UdpRxStatus.ul_MsgCount);
            printf("UDP Port 2 Error Count: %d\n", x_UdpRxStatus.ul_MsgErrorCount);

            break;
         }

      case 'x':
         {
            //Exit Application
            //--- Stop Tx/Rx, logout, and free handles
            MyFdxStopTx();
            MyFdxStopRx();
            MyFdxFreeResources();
            l_continue = FALSE;
            break;
         }
      default:
         ;
      }
   }
}

//-------------------------------------------------------
// MyFdxGetVLActivity
//-------------------------------------------------------
void MyFdxGetVLActivity()
{
   TY_FDX_RX_VL_ACTIVITY_IN x_VLActivityIn;
   TY_FDX_RX_VL_ACTIVITY_OUT x_VLActivityOut;
   TY_FDX_RX_VL_ACTIVITY * px_VLActivity;

   x_VLActivityIn.ul_Mode = FDX_RX_VL_ACT_ALL;
   x_VLActivityIn.ul_MaxReadBytes = 10*sizeof(TY_FDX_RX_VL_ACTIVITY);
   x_VLActivityOut.pax_VLActivity =
   (TY_FDX_RX_VL_ACTIVITY*)malloc(10*sizeof(TY_FDX_RX_VL_ACTIVITY));

   if (FDX_OK != (FdxCmdRxVLGetActivity(g_ulPort2Handle, &x_VLActivityIn,
         &x_VLActivityOut)))
   {
      printf("\nFdxCmdRxVLGetActivity Error");
   }
```

1 - Get UDP1 Port2 Status

1 - Get UDP2 Port2 Status

x - Exit Program

See the local functions for API function calls required.

Resources should be freed before exit. See the local function `MyFdxFreeResources` *for API function calls required.*

Local function call `MyFdxGetVLActivity` *will retrieve the frame count for the number of active virtual links.*

```
   printf("Number of Active VLs: %d\n", x_VLActivityOut.ul_NumOfActivVL);
   px_VLActivity = x_VLActivityOut.pax_VLActivity;

   AiUInt32 i;

   for (i=1; (i <= x_VLActivityOut.ul_NumOfActivVL); i++)
   {
      printf("VLid: %d Frame Count: %d\n", px_VLActivity->ul_VLIdent,
                                 px_VLActivity->ul_FrameCountA);
      px_VLActivity++;
   }
}
```

This local function is called prior to termination of the program within the MyFdxGetStatus local function. this function demonstrates:
(1) Logout of each board/port resource using ***FdxLogout*** *and.*
(2) Deletion of the UDP ports associated with the physical port using ***FdxCmdTxUDPDestroyPort*** *and* ***FdxCmdRxUDPDestroyPort***

```
//--------------------------------------------------------
// MyFdxFreeResources
//--------------------------------------------------------
void MyFdxFreeResources()
{
   if (g_ulBoardHandle != 0)
   {
      if (FDX_ERR == FdxLogout(g_ulBoardHandle))
      {
         printf("FdxLogout Board Error");
      }

      if (g_ulPort1Handle != 0)
      {
         if (g_pUdp1Port1Handle != NULL)
         {
            if (FDX_ERR == FdxCmdTxUDPDestroyPort(g_ulPort1Handle, g_pUdp1Port1Handle))
            {
               printf("FdxCmdTxUDPDestroyPort Error 1");
            }
         }
         if (g_pUdp2Port1Handle != NULL)
         {
            if (FDX_ERR == FdxCmdTxUDPDestroyPort(g_ulPort1Handle, g_pUdp2Port1Handle))
            {
               printf("FdxCmdTxUDPDestroyPort Error 1");
            }
         }
         if (FDX_ERR == FdxLogout(g_ulPort1Handle))
         {
            printf("FdxLogout Error 1");
         }
      }

      if (g_ulPort2Handle != 0)
      {
         if (g_pUdp1Port2Handle != NULL)
         {
            if (FDX_ERR == FdxCmdRxUDPDestroyPort(g_ulPort2Handle, g_pUdp1Port2Handle))
            {
               printf("FdxCmdRxUDPDestroyPort Error 2");
            }
         }
         if (g_pUdp2Port2Handle != NULL)
         {
            if (FDX_ERR == FdxCmdRxUDPDestroyPort(g_ulPort2Handle, g_pUdp2Port2Handle))
            {
               printf("FdxCmdRxUDPDestroyPort Error 2");
            }
         }
         if (FDX_ERR == FdxLogout(g_ulPort2Handle))
         {
            printf("FdxLogout Error 2");
         }
      }
   }
}
```

5.2.2 Generic Transmission/Chronological Monitor Reception Sample

This sample demonstrates how to:

1. Query available resources and login to a local board and two ports.
2. Reset the board and synchronize board-IRIG-time with PC-time.
3. Setup Port 1 for Generic transmit mode i.e., a list of AFDX frames with additional header information can be sent from a specified queue cyclically or a specific number of times
   - A UDP and IP checksum computation is performed and checksum entered
4. Setup Port 2 to capture using the Chronological Monitor Receive mode i.e., Data of all enabled links are stored in one large chronological monitor buffer.
5. Port1 sends data to Port2. *(an ethernet connection between Port 1 and Port 2 is required.)*

stfafx.h - MSWindows standard system include files or project specific include files that are used

```
#include "stdafx.h"
#include <time.h>
```

aifdx_def.h *header file is the only header file required for inclusion to support the AIM API. It contains the constant, structure and function definitions used in the API.*

```
#include "aifdx_def.h"
```

Function defintions used for functions defined within this program.

```
//---------------------------------------------------------------
// Function-Declarations
//---------------------------------------------------------------
bool MyFdxInit();
void MyFdxFreeResources();
void MyFdxResetBoard();
void MyFdxResetPort();
void MyFdxSetupTxPort();
void MyFdxSetupRxPort();
void MyFdxStartTx();
void MyFdxStartRx();
void MyFdxStopTx();
void MyFdxStopRx();
void MyFdxGetStatus();
void MyFdxGetVLActivity();

//---------------------------------------------------------------
// Globals
//---------------------------------------------------------------
AiUInt32            g_ulBoardHandle  = 0;
AiUInt32            g_ulPort1Handle  = 0;   // acts as Tx
AiUInt32            g_ulPort2Handle  = 0;   // acts as Rx
AiUInt32            g_ulQueueId = 0;
char                g_stop;
```

A handle (ID) for the board and each port on the board is required.

The main program provides an overview of the order of any basic application program:

Initialization--->Board setup--->Port Setup--->Frame-Data Setup for Tx---> Monitor Setup for Rx--->StartTx/Rx

After starting Tx/Rx - MyFdxGetStatus provides user controlled action for:

1. *Check Tx Status*
2. *Check Rx Status*
3. *Read from Monitor Queue*
4. *Exit*

Each local function contains the API function calls required to perform these capabilities.

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

   /*--- init application interface, query resources on local server and login to get valid
   handles */
   printf("\n");
   printf("Performing API initialization and local resource login..\n");
   if (MyFdxInit())
   {
      printf("API initialized, Login successful\n");

      //--- reset
      printf("\nPerforming board-reset and syncronizing IRIG to PC-Time...\n");
      MyFdxResetBoard();

      printf("\nPerforming port-resets...\n");
      MyFdxResetPort();

      printf("Press Any key to continue\n");
      getchar();
      //--- setup
      printf("\nPerforming Tranmitter setup on Port 1...\n");
      MyFdxSetupTxPort();

      printf("\nPerforming Receiver setup on Port 2...\n");
      MyFdxSetupRxPort();

      //--- Start Receiver
      printf("\nPerforming Receiver startup...\n");
      MyFdxStartRx();

      //--- Start Transmitter
      printf("\nPerforming Transmitter startup...\n");
      MyFdxStartTx();
   }
   else
   {
      printf("API Open Failure!!!\n");
   }

   MyFdxGetStatus();

   /* use as last function to free the resource list, the device list and the server list */
   if (FDX_OK != FdxExit())
      printf("\r\n FdxExit() FAIL");

   return 0;
}
```

```
//----------------------------------------------------------------
// MyFdxInit - returns true on success
//----------------------------------------------------------------
//    Init the application interface and
//    gets the global handles to local resources
//----------------------------------------------------------------
bool MyFdxInit()
{
   DWORD                          dwTmp;
   bool                           bRetSuccess = false;
   bool                           bFoundLocalServer = false;
   TY_SERVER_LIST *               px_ServerNames = NULL;
   TY_SERVER_LIST *               px_TmpServer;
   TY_RESOURCE_LIST_ELEMENT *     pRLE = NULL;
   TY_RESOURCE_LIST_ELEMENT *     pRLEHead = NULL;
   TY_FDX_CLIENT_INFO             x_ClientInfo;

   //--- init client-info
   sprintf(x_ClientInfo.ac_ClApplication, "AFDX-Sample Application");
   sprintf(x_ClientInfo.ac_ClApplicationVersion, "1.0");
   dwTmp = MAX_FDX_CLIENT_HOST_NAME;
   ::GetComputerName((LPWSTR)(x_ClientInfo.ac_ClHostName), &dwTmp);
   dwTmp = MAX_FDX_CLIENT_USER_NAME;
   ::GetUserName((LPWSTR)(x_ClientInfo.ac_ClUser), &dwTmp);
```

The **first** local function (called by the main program) to be executed performs:
(1) Initialization of the API
(2) Board login
(3) Port(s) login.

GetComputerName/GetUserName are MSWindows functions that retrieve the computer name/user name of the current system. These values are used for the ***FdxLogin*** function.

```
   //--- application interface initialize
   // and get a list of available servers
   if (FdxInit(&px_ServerNames) != FDX_OK)
   {
      printf("API Open Failed!!!\n");
      // free the server-list
      if (px_ServerNames != NULL)
      {
         FdxCmdFreeMemory(px_ServerNames, px_ServerNames->ul_StructId);
      }
      return(bRetSuccess);
   }

   // search the server-list for local server
   px_TmpServer = px_ServerNames;
   while ((px_TmpServer != NULL) && (!bFoundLocalServer))
   {
      if (stricmp(px_TmpServer->auc_ServerName, "local") == 0)
      {
         bFoundLocalServer = true;
      }
      else
      {
         px_TmpServer = px_TmpServer->px_Next;
      }
   }

   if (bFoundLocalServer)
   {  // ok, we found a local server
      // lets query the configuration of this server
      if (FdxQueryServerConfig("local", &pRLEHead) == FDX_OK)
      {
         pRLE = pRLEHead;
         while (pRLE != NULL)
         {  //--- login to resources
            switch(pRLE->ul_ResourceType)
            {
```

FdxInit returns the names of available servers at px_ServerNames. If px_ServerNames = "local", the AFDX board is located where the API is running. If px_ServerNames = "NULL", the end of the list has been reached. **Note**: this version will only return "local".

If a local server was found then the local server is searched for available AFDX boards, using ***FdxQueryServerConfig***.

FdxQueryServerConfig will return a list of resouces (board and port) available which will be used as an input for ***FdxLogin***. The following code assumes an FDX-2 board configuration, thus only logging into one board and two ports.

```
            case RESOURCETYPE_BOARD:
               if (g_ulBoardHandle == 0)
               {
                  if (FdxLogin("local", &x_ClientInfo, pRLE->ul_ResourceID, PRIVILEGES_ADMIN,
                  &g_ulBoardHandle) != FDX_OK)
                  {
                     g_ulBoardHandle = 0;
                     printf("Board Login Failure!!!\n");
                  }
               }
               break;
```

FdxLogin returns the handle for the board or port resource.

```
            case RESOURCETYPE_PORT:
               if (g_ulPort1Handle == 0)
               {
                  if (FdxLogin("local", &x_ClientInfo, pRLE->ul_ResourceID, PRIVILEGES_ADMIN,
                  &g_ulPort1Handle) != FDX_OK)
                  {
                     g_ulPort1Handle = 0;
                     printf("Port 1 Login Failure!!!\n");
                  }
               }
               else
                  if (g_ulPort2Handle == 0)
                  {
                     if (FdxLogin("local",&x_ClientInfo,pRLE->ul_ResourceID, PRIVILEGES_ADMIN,
                                 &g_ulPort2Handle) != FDX_OK)
                     {
                        g_ulPort2Handle = 0;
                        printf("Port 2 Login Failure!!!\n");
                     }
                  }
               break;
            }

            pRLE = pRLE->px_Next;
         }

      }

   }
   // free the resource-list
   if (pRLEHead != NULL)
   {
      FdxCmdFreeMemory(pRLEHead, pRLEHead->ul_StructId);
   }

   // free the server-list
   if (px_ServerNames != NULL)
   {
      FdxCmdFreeMemory(px_ServerNames, px_ServerNames->ul_StructId);
   }


   // we define it as success if we have valid handles for all global variables....
   bRetSuccess = (g_ulBoardHandle != 0) && (g_ulPort1Handle != 0) && (g_ulPort2Handle != 0);


   return bRetSuccess;
}
```

*The memory allocated when either resources were found using **FdxQueryServerConfig**, or server was found using **FdxInit**, must be released prior to termination of the program.*

> *The **second** local function (called by the main program) demonstrates two **System (Board-level) functions** to:*
>
> *(1) Configure each **port** as either **single or redundant**, and set up the **network bit rate** (default, as in this case, is 100 Mbps)*
>
> *(2) Initialize the internal Board IRIG time.*

```
//-----------------------------------------------------------
// MyFdxResetBoard
//-----------------------------------------------------------
void MyFdxResetBoard()
{
   int i;
   AiUInt32                 ul_Mode;
   time_t                   loc_time;
   struct tm *              ptm;
   TY_FDX_BOARD_CTRL_IN     x_BoardCtrlIn;
   TY_FDX_BOARD_CTRL_OUT    x_BoardCtrlOut;
   TY_FDX_IRIG_TIME         x_IrigTime;


   if (g_ulBoardHandle > 0)
   {
      //--- init input structure
      for (i=0; i<FDX_MAX_BOARD_PORTS; i++)
      {
         x_BoardCtrlIn.aul_PortConfig[i] = FDX_SINGLE;
         x_BoardCtrlIn.aul_ExpertMode[i] = FDX_EXPERT_MODE;
      }
      x_BoardCtrlIn.ul_RxVeriMode = FDX_BOARD_VERIFICATION_TYPE_I
```

> *Using **FdxCmdBoardControl**, each **port** is configured as a **single port** and the **network bit rate** is set to the default 100 Mbps. Verification mode is set to default which means the firmware will determine the program-specific board in use and setup the verification register accordingly.*

```
      //--- reset board
      if (FDX_OK != (FdxCmdBoardControl(g_ulBoardHandle, FDX_WRITE, &x_BoardCtrlIn,
            &x_BoardCtrlOut)))
     {
        printf("Board Reset Failure!!!\n");
     }
     else
     {
        printf("Board Initialized\n");
     }

      //--- sync board-internal irigtime-source with PC-time
      loc_time     = time(NULL);
      ptm          = localtime(&loc_time);
      memset(&x_IrigTime, 0, sizeof(x_IrigTime));
      if (ptm != NULL)
      {
         x_IrigTime.ul_Day      = ptm->tm_yday + 1;    // tm.YearOfDay is ZeroBased but we are
                            OneBased
         x_IrigTime.ul_Hour     = ptm->tm_hour;
         x_IrigTime.ul_Min      = ptm->tm_min;
         x_IrigTime.ul_Second   = ptm->tm_sec;
```

> *Using **FdxCmdIrigTimeControl**, the Board internal IRIG time can be synchronized to any time required by your application, in this case it is synched to the local PC time.*
>
> *Note: An external IRIG source can be used. In that case, initializing the internal IRIG time would not be applicable.*

```
         if (FDX_OK != (FdxCmdIrigTimeControl(g_ulBoardHandle, FDX_IRIG_WRITE, &x_IrigTime,
               &ul_Mode)))
         {
          printf("IRIG sync failure!!!\n");
       }
       else
       {
          printf("IRIG sync successful\n");
       }
     }

   }
}
```

```
//-----------------------------------------------------------------
// MyFdxResetPort
//-----------------------------------------------------------------
void MyFdxResetPort()
{
```

The ***third*** *local function (called by the main program) demonstrates* ***Port-level Transmitter and Receiver initialization****:*

The user must also assign a PortMap ID to each Tx/Rx port. This Port Map ID is a virtual ID assigned to the physical Port. The Receive PortMap ID is contained in the data read from the monitor queue (FdxCmdMonQueueRead). The Portmap ID aids in the identification of the physical port from which the data came, especially for applications

(a) Using multiple AFDX cards
(b) Using receive ports in redundant mode

```
   TY_FDX_PORT_INIT_IN  x_PortInitIn;
   TY_FDX_PORT_INIT_OUT x_PortInitOut;

   if (g_ulPort1Handle > 0)
   {
      x_PortInitIn.ul_PortMap = 1;
      if (FDX_OK != (FdxCmdTxPortInit(g_ulPort1Handle, &x_PortInitIn, &x_PortInitOut)))
      {
         printf("Port 1 Reset failure!!!\n");
      }
      else
      {
         printf("Port 1 Transmitter Initialized\n");
      }
   }
```

The initialized state after ***FdxCmdTxPortInit*** *is performed includes:*

(1) No Transmit Queues defined
(2) No VL created, No UPD Ports created
(3) FdxCmdTxControl command has no effect

```
   if (g_ulPort2Handle > 0)
   {
      x_PortInitIn.ul_PortMap = 2;
      if (FDX_ERR == FdxCmdRxPortInit(g_ulPort2Handle, &x_PortInitIn, &x_PortInitOut))
      {
         printf("Port 2 Reset failure!!!\n");
      }
      else
      {
         printf("Port 2 Receiver Initialized\n");
      }
   }

}
```

The initailized state after ***FdxCmdRxPortInit*** *is performed includes:*

(1) Global Statistics available
(2) All Virtual Links, enabled for Activity information
(3) Chronological Receive Mode, No VLs enabled for capturing
(4) No Trigger Control Block Processing Enabled

```
//------------------------------------------------------------
// MyFdxSetupTxPort
//------------------------------------------------------------
void MyFdxSetupTxPort()
{
   TY_FDX_TX_MODE_CTRL      x_TxModeCtrl;
   TY_FDX_TX_QUEUE_SETUP    x_TxQueueCreate;
   TY_FDX_TX_QUEUE_INFO     x_TxQueueInfo;
   AiUInt8 Dt[100];
   struct my_Frame_tag
   {
      TY_FDX_TX_FRAME_HEADER x_Frame;
      AiUInt8 uc_Data[1000];
   } My_Frame;

   int i;
```

This local function is called by the main program. Now that the board and ports have been initialized we can setup Port 1 to transmit data as follows:

(1) Generic Transmit Mode
(2) Initialize a Transmit queue for data transmission
(3) Define the data to be transmitted in the queues and the frame attributes including protocol and error injection

Two AFDX frames are setup for cyclic transmission

***Generic mode** - in this mode, the user creates a list of AFDX frames which can be sent cyclically or a specified number of times.*

```
   //--- mode control -> Set TX port to Generic mode
   x_TxModeCtrl.ul_TransmitMode = FDX_TX_GENERIC;
   if (FDX_OK != (FdxCmdTxModeControl(g_ulPort1Handle, &x_TxModeCtrl)))
   {
      printf("Port 1 Mode Control Failure!!!\n");
   }
   else
   {
      printf("Port 1 set to Generic Transmit mode\n");
   }
```

Memory must be allocated for the storage of the frames to be transmitted. One queue is used for one port. If the queue size is zero, the default queue size will be selected.

```
   //--- Create Generic Tx Message Queue
   // 0 Creates a queue of default size.
   x_TxQueueCreate.ul_QueueSize = 0;
   if (FDX_OK != (FdxCmdTxQueueCreate(g_ulPort1Handle, &x_TxQueueCreate, &x_TxQueueInfo)))
   {
      printf("Message Queue Creation failure!!!\n");
   }
   else
   {
      printf("Message Queue Created\n");
   }
```

TY_FDX_TX-FRAME_ATTRIB is the structure within `TY_FDX_TX_FRAME_HEADER` used to define the attributes (non-data) of the frame. The following frame attributes will be used for both frames (Frames 1 and 2) written to the transmit queue.

```
   //--- Create 2 Frames for the Tx Queue
   My_Frame.x_Frame.uc_FrameType = FDX_TX_FRAME_STD;
   My_Frame.x_Frame.x_FrameAttrib.uw_FrameSize = 64;  //bytes (includes CRC)
   My_Frame.x_Frame.x_FrameAttrib.ul_InterFrameGap = 25;  // 25=1usec; 1000=40usec;
   My_Frame.x_Frame.x_FrameAttrib.ul_PacketGroupWaitTime = 1000;  // 1000=1msec; 0=0usec;
   My_Frame.x_Frame.x_FrameAttrib.uc_PayloadBufferMode = FDX_TX_FRAME_PBM_STD;
   My_Frame.x_Frame.x_FrameAttrib.uc_PayloadGenerationMode = FDX_TX_FRAME_PGM_USER;
      //no payload generation - all frame data defined by the user in this frame entry
   My_Frame.x_Frame.x_FrameAttrib.ul_BufferQueueHandle = 0;  //used when payload buffer mode
                        is not standard
   My_Frame.x_Frame.x_FrameAttrib.uc_ExternalStrobe = FDX_DIS;
   My_Frame.x_Frame.x_FrameAttrib.uc_PreambleCount = FDX_TX_FRAME_PRE_DEF;
   My_Frame.x_Frame.x_FrameAttrib.ul_Skew = 0; // usec, used only for redundant mode
```

```
My_Frame.x_Frame.x_FrameAttrib.uc_NetSelect = FDX_TX_FRAME_BOTH;  // used only for
                          redundant mode
My_Frame.x_Frame.x_FrameAttrib.uc_FrameStartMode = FDX_TX_FRAME_START_PGWT;
My_Frame.x_Frame.x_FrameAttrib.ul_PhysErrorInjection = FDX_TX_FRAME_ERR_OFF;
My_Frame.x_Frame.x_FrameAttrib.uw_SequenceNumberInit = FDX_TX_FRAME_SEQ_INIT_AUTO;
My_Frame.x_Frame.x_FrameAttrib.uw_SequenceNumberOffset = FDX_TX_FRAME_SEQ_OFFS_AUTO;
```

Now we must insert the data into Frame 1. (The structure of the data and the fixed data inserted into the frame is defined in the AFDX End System Detailed Functional Specification.)

First, initialize the each byte of the data buffer with incrementing ASCII characters, starting from ASCII 0.

Then store the MAC/IP/UDP header and payload data into the frame. The frame is defined for VL 60.

```
//--- Frame 1 --- VL 60
for ( i = 0 ; i<1000; i++)
   My_Frame.uc_Data[i] = (unsigned char) i;

//---MAC Dst= 0x03000000003c (VL 60)
Dt[ 0]=0x03;Dt[ 1]=0x00;Dt[ 2]=0x00;Dt[ 3]=0x00;Dt[ 4]=0x00;Dt[ 5]=0x3c;

//---MAC Src= 0x020000012120
Dt[ 6]=0x02;Dt[ 7]=0x00;Dt[ 8]=0x00;Dt[ 9]=0x01;Dt[10]=0x21;Dt[11]=0x20;

//---MAC Type/Length
Dt[12]=0x08;Dt[13]=0x00;

//---IP Header (Version/IHL, Type of service, Total length, Fragment ID, Time to live,
// Protocol, Header Checksum)
Dt[14]=0x45;Dt[15]=0x00;Dt[16]=0x00;Dt[17]=0x2d;Dt[18]=0x00;Dt[19]=0x00;Dt[20]=0x40;
Dt[21]=0x00;Dt[22]=0x01;Dt[23]=0x11;Dt[24]=0x6d;Dt[25]=0xa2;

//---IP Source Address 10.001.33.1
Dt[26]=0x0a;Dt[27]=0x01;Dt[28]=0x21;Dt[29]=0x01;

//---IP Destination Address 224.224.0.60 (VL 60)
Dt[30]=0xe0;Dt[31]=0xe0;Dt[32]=0x00;Dt[33]=0x3C;

//---UDP Source Port = 24
Dt[34]=0x00;Dt[35]=0x18;

//---UDP Dest Port = 23
Dt[36]=0x00;Dt[37]=0x17;

//---UDP Length = 25
Dt[38]=0x00;Dt[39]=0x19;

//---UDP Checksum
Dt[40]=0x00;Dt[41]=0x00;

//---AFDX Payload
Dt[42]=0x41;Dt[43]=0x42;Dt[44]=0x43;Dt[45]=0x44;Dt[46]=0x45;
Dt[47]=0x46;Dt[48]=0x47;Dt[49]=0x48;Dt[50]=0x49;Dt[51]=0x4a;
Dt[52]=0x4b;Dt[53]=0x4c;Dt[54]=0x4d;Dt[55]=0x4e;Dt[56]=0x4f;
Dt[57]=0x50;Dt[58]=0x51;

for ( i = 0 ; i<59; i++)
   My_Frame.uc_Data[i] = (unsigned char) Dt[i];
```

Write the Frame attributes and the Frame1 data to the Transmit Queue.

```
if (FDX_OK != (FdxCmdTxQueueWrite(g_ulPort1Handle,
         FDX_TX_FRAME_HEADER_GENERIC,1,sizeof(My_Frame),&My_Frame)))
{
   printf("Write to Queue Failed!!!\n");
}
else
{
   printf("Frame successfully written to Queue\n");
}
```

Now we must insert the data into Frame 2 (also for VL60). The UDP source and destination ports are different from Frame 1

First, initialize the each byte of the data buffer with incrementing ASCII characters, starting from ASCII 0.

Then store the MAC/IP/UDP header and payload data into the frame.

```
//--- Frame 2 --- VL 60
for ( i = 0 ; i<1000; i++)
   My_Frame.uc_Data[i] = (unsigned char) i;

//---MAC Dst= 0x03000000003c (VL 60)
Dt[ 0]=0x03;Dt[ 1]=0x00;Dt[ 2]=0x00;Dt[ 3]=0x00;Dt[ 4]=0x00;Dt[ 5]=0x3c;

//---MAC Src= 0x020000012120
Dt[ 6]=0x02;Dt[ 7]=0x00;Dt[ 8]=0x00;Dt[ 9]=0x01;Dt[10]=0x21;Dt[11]=0x20;

//---MAC Type/Length
Dt[12]=0x08;Dt[13]=0x00;

//---IP Header (Version, IHL, Type of service, Total length, Fragment ID, Time to live,
// Protocol, Header Checksum)
Dt[14]=0x45;Dt[15]=0x00;Dt[16]=0x00;Dt[17]=0x2d;Dt[18]=0x00;Dt[19]=0x00;Dt[20]=0x40;
   Dt[21]=0x00;Dt[22]=0x01;Dt[23]=0x11;  Dt[24]=0x6d;Dt[25]=0xa2;

//---IP Source Address 10.001.33.1
Dt[26]=0x0a;Dt[27]=0x01;Dt[28]=0x21;Dt[29]=0x01;

//---IP Destination Address 224.224.0.60 (VL 60)
Dt[30]=0xe0;Dt[31]=0xe0;Dt[32]=0x00;Dt[33]=0x3c;

//---UDP Source Port = 34
Dt[34]=0x00;Dt[35]=0x22;

//---UDP Dest Port = 33
Dt[36]=0x00;Dt[37]=0x21;

//---UDP Length = 25
Dt[38]=0x00;Dt[39]=0x19;

//---UDP Checksum
Dt[40]=0x00;Dt[41]=0x00;

//---Payload
Dt[42]=0x41;Dt[43]=0x42;Dt[44]=0x43;Dt[45]=0x44;Dt[46]=0x45;
Dt[47]=0x46;Dt[48]=0x47;Dt[49]=0x48;Dt[50]=0x49;Dt[51]=0x4a;
Dt[52]=0x4b;Dt[53]=0x4c;Dt[54]=0x4d;Dt[55]=0x4e;Dt[56]=0x4f;
Dt[57]=0x50;Dt[58]=0x51;

for ( i = 0 ; i<58; i++)
   My_Frame.uc_Data[i] = (unsigned char) Dt[i];

if (FDX_OK != (FdxCmdTxQueueWrite(g_ulPort1Handle,
         FDX_TX_FRAME_HEADER_GENERIC,1,sizeof(My_Frame),&My_Frame))) {
   printf("Write to Queue Failed!!!\n");
}
else {
   printf("Frame successfully written to Queue\n");
}
```

```
}
```

```
//-------------------------------------------------------
// MyFdxSetupRxPort
//-------------------------------------------------------
void MyFdxSetupRxPort()
{
   TY_FDX_RX_MODE_CTRL_IN  x_ModeCtrlIn;
   TY_FDX_RX_MODE_CTRL_OUT x_ModeCtrlOut;
   TY_FDX_MON_CAP_MODE x_MonCapMode;
   TY_FDX_MON_QUEUE_CTRL_IN x_QueueCtrlIn;
   TY_FDX_MON_QUEUE_CTRL_OUT x_QueueCtrlOut;
```

This local function (called by the main program) will configure the Receive Port to capture the data transmitted by Port1. (Assuming the appropriate ethernet connection has been configured between ports 1 and 2). Port 2 will be setup as follows:

(1) Chronological Receive Mode
(2) Continuous Capture
(3) Create monitor queue to receive the captured data.

***Chronological Receive Mode** - indicates that VL data streams are captured and stored into one Monitor buffer.(vs. VL-oriented storage)*

```
   //--- mode control -> select Chrono Mode
   x_ModeCtrlIn.ul_ReceiveMode = FDX_RX_CHRONO;
```

***Payload mode** - allows you to store the entire frame, or other specific portions of the frame. In this case, the entire frame is stored.*

```
   x_ModeCtrlIn.ul_DefaultPayloadMode = FDX_PAYLOAD_FULL;
```

***Default Chronological mode** - allows you to capture all VL data, only the good frames, or only perform statistics without capturing the frame. In this case, all VLs are captured.*

```
   x_ModeCtrlIn.ul_DefaultCronoMode = FDX_RX_DEFAULT_MON_ENA_ALL;
   x_ModeCtrlIn.ul_GlbMonBufferSize = 0;  // if zero, a default value will be used
   if (FDX_OK != (FdxCmdRxModeControl(g_ulPort2Handle, &x_ModeCtrlIn, &x_ModeCtrlOut)))
   {
      printf("Port 2 Mode Control Failure!!!\n");
   }
   else
   {
      printf("Port 2 Set to Chrono Monitor Receive Mode\n");
      printf("Port 2 Global Mon Buffer Size: %d bytes\n",
            x_ModeCtrlOut.ul_GlbMonBufferSize);
   }
```

***Continuous Capture mode** - indicates the monitor buffer will be filled in a cyclic manner, such that once full, the oldest frames will be overwritten.*

```
   //--- Monitor Capture Control
   x_MonCapMode.ul_CaptureMode = FDX_MON_CONTINUOUS;
   x_MonCapMode.ul_Strobe = FDX_MON_STROBE_DIS; //no strobe will be ouput on capture
                        start or stop
   if (FDX_OK != (FdxCmdMonCaptureControl(g_ulPort2Handle, &x_MonCapMode)))
   {
      printf("Chrono Monitor Capture control failure!!!\n");
   }
   else
   {
      printf("Chrono Monitor Capture Mode set to Continuous\n");
   }
   //--- Create Monitor Queue
   x_QueueCtrlIn.ul_QueueControl = FDX_MON_QUEUE_CREATE;
   if (FDX_OK != (FdxCmdMonQueueControl(g_ulPort2Handle, &x_QueueCtrlIn, &x_QueueCtrlOut)))
   {
      printf("Monitor Queue Creation Failure!!!\n");
   }
   else
   {
      printf("Monitor Queue Created\n");
      g_ulQueueId = x_QueueCtrlOut.ul_QueueId;
   }
}
```

```
//---------------------------------------------------
// MyFdxStartTx
//---------------------------------------------------
void MyFdxStartTx()
{
   TY_FDX_TX_CTRL x_TxControl;

   x_TxControl.ul_Count = 2;
   x_TxControl.e_StartMode = FDX_START;

   if (g_ulPort1Handle != NULL)
   {
      if (FDX_OK != (FdxCmdTxControl(g_ulPort1Handle, &x_TxControl)))
      {
         printf("Failure to start transmitter\n");
      }
      else
      {
         printf("Transmitter started\n");
      }
   }
}
```

This local function (called by the main program) will start the transmission of AFDX frames via Port1. Send configuration includes:

(1) Send the AFDX frame 2 times
(2) Setup to start immediately (vs. wait for trigger)
.

```
//---------------------------------------------------
// MyFdxReceive
//---------------------------------------------------
void MyFdxStartRx()
{
   TY_FDX_RX_CTRL x_RxControl;
   if (g_ulPort2Handle != NULL)
   {

      x_RxControl.ul_StartMode             = FDX_START;
      x_RxControl.ul_GlobalStatisticReset = FDX_RX_GS_RES_ALL_CNT;

      if (FDX_OK != (FdxCmdRxControl(g_ulPort2Handle, &x_RxControl)))
      {
         printf("Failure to start Receiver!!!\n");
      }
      else
      {
         printf("Receiver Started\n");
      }

   }

}
```

This local function (called by the main program) will start the reception of AFDX frames via Port2. Receive configuration includes:

(1) Receive start
(2) Reset all counters prior to receive start

```
//-----------------------------------------------------
// MyFdxStopTx
//-----------------------------------------------------
void MyFdxStopTx()
{
   TY_FDX_TX_CTRL x_TxControl;

   x_TxControl.ul_Count = 0;
   x_TxControl.e_StartMode = FDX_STOP;


   if (FDX_ERR == FdxCmdTxControl(g_ulPort1Handle, &x_TxControl))
   {
      printf("FdxCmdTxControl Error");
   }

}
```

This local function (called by `MyFdxGetStatus)` *will stop the transmission of AFDX frames via Port1.*

```
//-----------------------------------------------------
// MyFdxStopRx
//-----------------------------------------------------
void MyFdxStopRx()
{
   TY_FDX_RX_CTRL x_RxControl;

   x_RxControl.ul_StartMode = FDX_STOP;
   x_RxControl.ul_GlobalStatisticReset = FDX_RX_GS_RES_ALL_CNT;

   if (FDX_OK != (FdxCmdRxControl(g_ulPort2Handle, &x_RxControl)))
   {
      printf("FdxCmdRxControl Error");
   }
}
```

This local function (called by `MyFdxGetStatus)` *will stop the reception of AFDX frames via Port2.*

```
//---------------------------------------------------
// MyFdxGetStatus
//---------------------------------------------------
```

This local function (called from the main program) allows the user to select the action to be taken by the program including:

1 - Get Tranmsmitter Status
2 - Get Receiver Status
3 - Read Frame from Monitor Queue
x - Exit the program

```
void MyFdxGetStatus()
{
   char l_command[10];
   bool l_continue = TRUE;
   TY_FDX_TX_STATUS x_TxStatus;
   TY_FDX_RX_STATUS x_RxStatus;
   TY_FDX_E_MON_STATUS e_MonStatus;
   TY_FDX_MON_REC_STATUS x_MonRecStatus;
   AiUInt32 ul_Control;
   TY_FDX_RX_GLOB_STAT x_GlobalStatisticA, x_GlobalStatisticB;
   TY_FDX_MON_QUEUE_READ_IN x_QueueReadIn;
   TY_FDX_MON_QUEUE_READ_OUT x_QueueReadOut;
   AiUInt8 ReadBuffer[2000];

   TY_FDX_FRAME_BUFFER_HEADER* px_FrameBufferHeader;

   while (l_continue == TRUE)
   {

      printf("\r\n '1'  Get Transmitter Status\n");
      printf(" '2'  Get Receiver Status\n");
      printf(" '3'  Read Frame from Monitor Queue\n");
      printf(" 'x'  Exit\n");
      printf("Select a Command: ");

      scanf("%s", l_command);

      switch (l_command[0])
      {
      case '1':
```

1 - Get Tranmsmitter Status

```
         {
            // Retrieve Transmitter Status
            printf("\nTransmitter Status:\n");

            if (FDX_OK != (FdxCmdTxStatus(g_ulPort1Handle, &x_TxStatus)))
            {
               printf("FdxCmdTxStatus Error\n");
            }

            printf("Port 1 Status: ");
            switch (x_TxStatus.e_Status)
            {
            case FDX_STAT_STOP:
               printf("Stopped\n");
               break;
            case FDX_STAT_RUN:
               printf("Running\n");
               break;
            case FDX_STAT_ERROR:
               printf("Error\n");
            }

            printf("Port 1 Frame Count: %d\n", x_TxStatus.ul_Frames);
            break;
         }
      case '2':
```

2 - Get Receiver Status

```
         {
            // Retrieve Receiver Status
            printf("\nReceiver Status:\n");

            if (FDX_OK != (FdxCmdRxStatus(g_ulPort2Handle, &x_RxStatus)))
            {
               printf("FdxCmdRxStatus Error\n");
            }
```

```
printf("Port 2 Status: ");
switch (x_RxStatus.ul_Status)
{
case FDX_STAT_STOP:
   printf("Stopped\n");
   break;
case FDX_STAT_RUN:
   printf("Running\n");
   break;
case FDX_STAT_ERROR:
   printf("Error\n");
   break;
}

ul_Control = FDX_RX_GS_RES_NO_CNT;
if (FDX_OK != (FdxCmdRxGlobalStatistics(g_ulPort2Handle,
   ul_Control, &x_GlobalStatisticA, &x_GlobalStatisticB)))
{
   printf("\nFdxCmdRxGlobalStatistics Error");
}

printf("Port 2 Global Statistics:\n");
printf("Good Frame Count: %d\n", x_GlobalStatisticA.ul_FrameGoodCount);
printf("Bad Frame Count:  %d\n", x_GlobalStatisticA.ul_FrameErrorCount);
printf("Total Byte Count on Port: %d\n", x_GlobalStatisticA.ul_TotalByteCount);

//--- Get VL Activity
MyFdxGetVLActivity();
```

Local function call `MyFdxGetVLActivity` *will retrieve the frame count for the number of active virtual links.*

This function will indicate the status of the monitor. Status values shown in case structure below.

```
//--- Monitor Status
if (FDX_OK != (FdxCmdMonGetStatus(g_ulPort2Handle, &e_MonStatus, &x_MonRecStatus)))
{
   printf("\nFdxCmdMonGetStatus Error");
}

printf("Monitor Status: ");

switch (e_MonStatus)
{
case FDX_MON_OFF:
   printf("Not Running\n");
   break;
case FDX_MON_WAIT_FOR_TRIGGER:
   printf("Waiting for Start Trigger\n");
   break;
case FDX_MON_TRIGGERED:
   printf("Monitor Triggered, Capturing Frames\n");
   break;
case FDX_MON_STOPPED:
   printf("Stopped\n");
   break;
case FDX_MON_FULL:
   printf("Monitor Buffer Full\n");
}

break;
}
case '3':
{
x_QueueReadIn.ul_EntryCount = 1;
x_QueueReadIn.ul_ReadQualifier = FDX_MON_READ_FULL;
x_QueueReadIn.ul_MaxReadBytes = sizeof(ReadBuffer);
x_QueueReadOut.pv_ReadBuffer = ReadBuffer;

if (FDX_OK != FdxCmdMonQueueRead(g_ulPort2Handle, g_ulQueueId,
   &x_QueueReadIn, &x_QueueReadOut))
{
```

3 - Get Monitor Status

```
            printf("FdxCmdMonQueueRead Error\n");
         }

         printf("Bytes Read: %d  Frames Read: %d\n",x_QueueReadOut.ul_BytesRead,
                                x_QueueReadOut.ul_EntryRead);
         printf("VL from MAC Addr: %d\n", ReadBuffer[41]);
         printf("ReadBuffer[78]: %04lx\n", ReadBuffer[78]);
         printf("ReadBuffer[79]: %04lx\n", ReadBuffer[79]);


         px_FrameBufferHeader = (TY_FDX_FRAME_BUFFER_HEADER*) x_QueueReadOut.pv_ReadBuffer;

         printf("VlId: %d\n",px_FrameBufferHeader->x_FrameHeaderInfo.uw_VlId);
         printf("Sequence Num: %d\n",px_FrameBufferHeader->x_FrameHeaderInfo.uc_SequenceNr);
         break;
      }
   case 'x':
      {
         //Exit Application
         //--- Stop Tx/Rx, logout, and free handles
         MyFdxStopTx();
         MyFdxStopRx();
         MyFdxFreeResources();
         l_continue = FALSE;
         break;
      }
   default:
      ;
   }


  }
}
```

x - Exit Program

See the local functions for API function calls required.

Resources should be freed before exit. See the local function `MyFdxFreeResources` *for API function calls required.*

```
//-----------------------------------------------------------------
// MyFdxGetVLActivity
//-----------------------------------------------------------------
void MyFdxGetVLActivity()
{
   TY_FDX_RX_VL_ACTIVITY_IN x_VLActivityIn;
   TY_FDX_RX_VL_ACTIVITY_OUT x_VLActivityOut;
   TY_FDX_RX_VL_ACTIVITY * px_VLActivity;

   x_VLActivityIn.ul_Mode = FDX_RX_VL_ACT_ALL;
   x_VLActivityIn.ul_MaxReadBytes = 10*sizeof(TY_FDX_RX_VL_ACTIVITY);
   x_VLActivityOut.pax_VLActivity =
   (TY_FDX_RX_VL_ACTIVITY*)malloc(10*sizeof(TY_FDX_RX_VL_ACTIVITY));
   if (FDX_OK != (FdxCmdRxVLGetActivity(g_ulPort2Handle, &x_VLActivityIn,
         &x_VLActivityOut)))
   {
      printf("\nFdxCmdRxVLGetActivity Error");
   }

   printf("Number of Active VLs: %d\n", x_VLActivityOut.ul_NumOfActivVL);
   px_VLActivity = x_VLActivityOut.pax_VLActivity;

   AiUInt32 i;

   for (i=1; (i <= x_VLActivityOut.ul_NumOfActivVL); i++)
   {
      printf("VLid:%d Frame Count:%d\n",px_VLActivity->ul_VLIdent,px_VLActivity->ul_FrameCountA);
      px_VLActivity++;
```

Local function call `MyFdxGetVLActivity` *will retrieve the frmae count for the number of active virtual links.*

```
    }
}
```

```
//-------------------------------------------------------------
// MyFdxFreeResources
//-------------------------------------------------------------
void MyFdxFreeResources()
{
   TY_FDX_MON_QUEUE_CTRL_IN  x_QueueCtrlIn;
   TY_FDX_MON_QUEUE_CTRL_OUT x_QueueCtrlOut;

   if (g_ulBoardHandle != 0)
      if (FDX_ERR == FdxLogout(g_ulBoardHandle))
      {
         printf("FdxLogout Board Error");
      }

      if (g_ulPort1Handle != 0)
      {
         if (FDX_ERR == FdxLogout(g_ulPort1Handle))
         {
            printf("FdxLogout Error 1");
         }
      }

      if (g_ulPort2Handle != 0)
      {
         if (g_ulQueueId != 0)
         {
            x_QueueCtrlIn.ul_QueueControl = FDX_MON_QUEUE_DELETE;
            x_QueueCtrlIn.ul_QueueId = g_ulQueueId;
            if (FDX_ERR == FdxCmdMonQueueControl(g_ulPort2Handle, &x_QueueCtrlIn,
               &x_QueueCtrlOut))
            {
               printf("FdxCmdMonQueueControl Error");
            }
         }

         if (FDX_ERR == FdxLogout(g_ulPort2Handle))
         {
            printf("FdxLogout Error 2");
         }
      }

}
```

This local function is called prior to termination of the program within the MyFdxGetStatus local function. This function demonstrates:
*(1) Logout of each board/port resource using **FdxLogout**.*
*(2) Deletion of the queue(s) associated with the chronological monitor using **FdxCmdMonQueueControl** (Port 2 was setup for chronological monitor.)*

5.3 API S/W Library Function Calls vs. Program Samples

Table 5-2 provides a list of all the function calls within the API S/W Library and which sample program contains the function call. This table is useful for searching for program examples of how a function call is used within a program.

Table 5-2 API S/W Library Function Calls vs. Program Samples

	afdx_MainSample.cpp	afdx_SystemFunc.cpp	afdx_SampleUtils.cpp	afdx_LogInOut.cpp	afdx_GenericRX.cpp	afdx_GenericTX.cpp	afdx_GenRX_CCSE.cp	afdx_GenTX_Ext.cpp	afdx_InterruptFunc.cpp	afdx_ReplayFunc.cpp	afdx_SimulationRX.cpp	afdx_SimulationTX.cpp	afdx_UdpRx.cpp	afdx_UdpTx.cpp
Library Administration Functions														
FdxInit	•													
FdxQueryServerConfig		•		•										
FdxQueryResource				•										
FdxInstallServerConfigCallback														
FdxLogin				•										
FdxLogout				•										
FdxInstIntHandler									•					
FdxDelIntHandler									•					
FdxExit	•													
System Functions														
FdxCmdBoardControl		•												
FdxCmdIrigTimeControl	•	•							•					
FdxCmdStrobeTriggerLine														
FdxReadBSPVersion		•												
FdxCmdBITETransfer		•												
Transmitter Functions														
FdxCmdTxPortInit						•		•		•		•		•
FdxCmdTxModeControl						•		•		•		•		•
FdxCmdTxControl						•				•		•		•
FdxCmdTxStatus						•				•				
FdxCmdTxTrgLineControl						•								
FdxCmdTxStaticRegsControl						•								
FdxCmdTxVLControl												•		
FdxCmdTxQueueCreate						•		•		•				
FdxCmdTxQueueStatus										•				

	afdx_MainSample.cpp	afdx_SystemFunc.cpp	afdx_SampleUtils.cpp	afdx_LogInOut.cpp	afdx_GenericRX.cpp	afdx_GenericTX.cpp	afdx_GenRX_CCSE.cp	afdx_GenTX_Ext.cpp	afdx_InterruptFunc.cpp	afdx_ReplayFunc.cpp	afdx_SimulationRX.cpp	afdx_SimulationTX.cpp	afdx_UdpRx.cpp	afdx_UdpTx.cpp
FdxCmdTxQueueWrite						•		•		•		•		
FdxCmdTxQueueUpdate								•						
FdxCmdTxCreateVL						•						•		•
FdxCmdTxCreateHiResVL														
FdxCmdTxUDPCreatePort												•		•
FdxCmdTxUDPChgSrcPort														
FdxCmdTxUDPDestroyPort												•		•
FdxCmdTxUDPWrite												•		•
FdxCmdTxUDPBlockWrite												•		
FdxCmdTxSAPCreatePort												•		
FdxCmdTxSAPWrite												•		
FdxCmdTxSAPBlockWrite												•		
FdxCmdTxUDPGetStatus												•		•
FdxCmdTxUDPControl												•		
FdxCmdTxVLWrite												•		•
FdxCmdTxVLWriteEx												•		
Receiver Functions														
FdxCmdRxPortInit					•				•		•		•	
FdxCmdRxModeControl					•		•				•		•	
FdxCmdRxControl					•		•		•		•		•	
FdxCmdRxStatus					•									
FdxCmdRxGlobalStatistics					•									
FdxCmdRxVLControl					•				•		•		•	
FdxCmdRxVLControlEx									•					
FdxCmdRxVLGetActivity					•									
FdxCmdRxTrgLineControl														
FdxCmdRxUDPCreatePort									•		•		•	
FdxCmdRxUDPChgDestPort														
FdxCmdRxUDPDestroyPort									•		•		•	
FdxCmdRxUDPRead											•			
FdxCmdRxUDPBlockRead											•		•	
FdxCmdRxUDPControl														
FdxCmdRxSAPCreatePort											•			
FdxCmdRxSAPWrite														
FdxCmdRxSAPBlockWrite														
FdxCmdRXUDPGetStatus									•		•			
FdxCmdMonCaptureControl					•		•							
FdxCmdMonTCBSetup					•									
FdxCmdMonTrgWordIni					•									

	afdx_MainSample.cpp	afdx_SystemFunc.cpp	afdx_SampleUtils.cpp	afdx_LogInOut.cpp	afdx_GenericRX.cpp	afdx_GenericTX.cpp	afdx_GenRX_CCSE.cp	afdx_GenTX_Ext.cpp	afdx_InterruptFunc.cpp	afdx_ReplayFunc.cpp	afdx_SimulationRX.cpp	afdx_SimulationTX.cpp	afdx_UdpRx.cpp	afdx_UdpTx.cpp
FdxCmdMonTrgIndexWordIni					•									
FdxCmdMonTrgIndexWordIniVL					•									
FdxCmdMonGetStatus	•				•									
FdxCmdMonQueueControl					•									
FdxCmdMonQueueRead					•									
FdxCmdMonQueueSeek														
FdxCmdMonQueueTell														
FdxCmdMonQueueStatus					•		•							
Target Indep Admin Function														
FdxCmdFreeMemory	•	•		•										
FdxFwIrig2StructIrig					•						•		•	
FdxStructIrig2FwIrig														
FdxAddIrigStructIrig		•								•				
FdxSubIrigStructIrig		•											•	
FdxTranslateErrorWord														
FdxInitTxFrameHeader														

THIS PAGE IS INTENTIONALLY LEFT BLANK

6 NOTES

6.1 Acronyms and Abbreviations

µsec	microseconds
AFDX	Avionic Full Duplex Switched Ethernet
API	Application Programming Interface
ARINC	Aeronautical Radio, Incorporated
ARM	Advanced RISC Machine
ASCII	American Standard Code for Information Exchange
ASP	Application Support Processor
ASP	Application Support Processor
BAG	Bandwidth Allocation Gap
BIP	Bus Interface Unit Processor
BIT	Built IN Test
BIU	Bus Interface Unit
BSP	Board Support Package
DCT	Dynamic Counter Table
E/S	End System
FCS	Frame Check Sequence
FIFO	First in - First out
FS	Frame Size
GTM	Generic Transmit Mode
GTU	Gap Time Unit
I/O	Input / Output
IC	Integrity Checking
ID	Identifier
IFG	Inter-frame Gap
IP	Internet Protocol
IPP	process invalid frames
IRIG B	Inter Range Instrumentation Group, Time Code Format Type B
LCA	Xilinx Logic Cell Array (Field Programmable Logic)
LSB	Least Significant Byte
MAC	Medium Access Controller
Mbps	Mega bits per second
MCFL	Maximum Consecutive Frames Lost
MSB	Most Significant Byte
ns	Nanoseconds
OIN	Open Information Network
OS	Operating System
OSI	Open System Interconnect

PBI	Physical Bus Interface
PC	Personal Computer.
PCI	Peripheral Component Interconnect
PGWT	Packet group wait time
PMC	PCI Mezzanine Card
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer.
RM	Redundancy Management
RMA	Redundancy Management Algorithm
RP(M)	Replay Mode
S/Q	Sampling & Queuing
SAP	Service Access Point
SCB	System Control Block
SFD	Start Frame Delimiter
SN	Sequence Number
STM	Simulator Transmit Mode
TAP	Test Access Point
TBD	To be defined
TCB	Monitor Trigger Control Block
TFTP	Trivial File Transfer Protocol
TS	Traffic Shaping
UDP	User Datagram Protocol
VL	Virtual Link
VME	Versatile Bus Modular European (computer bus)

6.2 Definition of Terms

address quintuplet	the address of an AFDX Comm port which consists of UDP Source/Destination, IP Source/Destination, and MAC Destination address (VL)
Bandwidth Allocation Gap	The time difference between the start of one frame and the beginning of the next frame transmitted on the port.
Big Endian	a system of memory addressing in which numbers that occupy more than one byte in memory are stored "big end first" with the uppermost 8 bits at the lowest address.
Channel	Two physical AFDX ports
Driver Command	command used by the AIM target s/w to control the FDX device
FLASH	page oriented electrical erasable and programmable memory
function	a self-contained block of code with a specific purpose that returns a single value.
Interframe Gap	Gap between the end of the preceding frame and the current frame.
interrupt	a signal from a device attached to a computer or from a program within the computer that causes the main program that operates the computer (the operating system) to stop and figure out what to do next
Jitter	The difference between the minimum and maximum time from when a source node sends a message to when the sink node receives the message. Jitter is generally a function of the network design and multiplexing multiple VLs on one port.
Little Endian	a system of memory addressing in which numbers that occupy more than one byte in memory are stored "little end first" with the lowest 8 bits at the lowest address.
multicast	Multicast is communication between a single sender and multiple receivers on a network.
Packet Group Wait Time	The time from the transmission start point of the last frame to the start point of the current frame with a resolution of 1us.
Port	One physical AFDX Port
Strobe	a strobe is a signal that is generated based on the conditions defined in the API
Target	Refers to the software/communication active on the target device
unicast	Unicast is communication between a single sender and a single receiver over a network.

THIS PAGE IS INTENTIONALLY LEFT BLANK

7 API S/W LIBRARY INDEX