



*United
Electronic
Industries*

AMETEK[®]

DNx-ARINC-664

—

User Manual

ARINC-664 Protocol Communications Interface
for the PowerDNA Cube or RACK series chassis

November 2023

PN Man-DNx-ARINC-664

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form by any means, electronic, mechanical, by photocopying, recording, or otherwise without prior written permission.

Information furnished in this manual is believed to be accurate and reliable. However, no responsibility is assumed for its use, or for any infringement of patents or other rights of third parties that may result from its use.

All product names listed are trademarks or trade names of their respective companies.

See the UEI website for complete terms and conditions of sale:

<http://www.ueidaq.com/cms/terms-and-conditions>



Contacting United Electronic Industries

Mailing Address:

249 Vanderbilt Avenue
Norwood, MA 02062
U.S.A.

Shipping Address:

24 Morgan Drive
Norwood, MA 02062
U.S.A.

For a list of our distributors and partners in the US and around the world, please contact a member of our support team:

Support:

Telephone: (508) 921-4600
Fax: (508) 668-2350

Also see the FAQs and online "Live Help" feature on our web site.

Internet Support:

Support: uei.support@ametek.com
Web-Site: www.ueidaq.com
FTP Site: <ftp://ftp.ueidaq.com>

Product Disclaimer:

WARNING!

DO NOT USE PRODUCTS SOLD BY UNITED ELECTRONIC INDUSTRIES, INC. AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS.

Products sold by United Electronic Industries, Inc. are not authorized for use as critical components in life support devices or systems. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness. Any attempt to purchase any United Electronic Industries, Inc. product for that purpose is null and void and United Electronic Industries Inc. accepts no liability whatsoever in contract, tort, or otherwise whether or not resulting from our or our employees' negligence or failure to detect an improper purchase.

Specifications in this document are subject to change without notice. Check with UEI for current status.

Table of Contents

Chapter 1 Introduction	1
1.1 Organization of this Manual	1
1.2 DNx-ARINC-664 Board Overview	3
1.2.1 ARINC-664 Receiver	3
1.2.2 ARINC-664 Monitor	3
1.2.3 ARINC-664 Transmitter	3
1.2.4 Certification	3
1.2.5 Software Support	3
1.3 Features	4
1.4 Indicators	4
1.5 Specification	5
1.6 ARINC-664 Overview	6
1.6.1 DNx-ARINC-664 Applications	6
1.6.2 ARINC-664 Network Overview	7
1.6.3 OSI Model Structure	10
1.6.4 ARINC-664 Packet Structure	12
1.6.5 Bandwidth	13
1.7 Device Architecture	14
1.7.1 Device RTOS	14
1.8 Wiring & Connectors	15
1.8.1 Connecting to the ARINC-664 Network	15
Chapter 2 Programming with the Low-level API	16
2.1 About the Low-level API	16
2.2 Low-level Functions	16
2.3 Tutorial	18
2.3.1 Initialization	18
2.3.2 Configuration	19
2.3.3 Send / Receive Messages	20
2.3.4 Stop Cleanly	28
2.4 UEI ARINC-664 XML Configuration	29
Chapter 3 Tools and Diagnostics	32
3.1 Diagnostic Panel for PowerDNA Explorer	32
3.2 Device RTOS Processes	33
3.2.1 Reception	33
3.3 ARINC-664 Network Packet Inspection	35
Appendix A	38
A.1 Accessories	38



List of Figures

1-1	The DNR-ARINC-664 Board.....	4
1-2	Example of the DNx-ARINC-664 in an ARINC-664 Network.....	6
1-3	Basic ARINC-664 Network (Logical Addressing Perspective).....	7
1-4	The OSI Model for ARINC-664 Systems	10
1-5	ARINC-664 Packet Structure.....	12
1-6	DNx-ARINC-664 Logic Block Diagram	14
1-7	Connection diagram for the DNR-ARINC-664	15
2-1	Immediate (left), VMap (top right), aEvent (lower right).....	20
2-2	DaqBios packet format for VMap refresh.....	22
2-3	VMap control & data	23
2-4	VMap+ control & data payload	23
2-5	ARINC-664 Configurator	29
3-1	DNx-ARINC-664 Panel in PowerDNA Explorer	32



Chapter 1 Introduction

This document outlines the feature set of the DNx-ARINC-664 board and its use as an ARINC-664 communications interface.

1.1 Organization of this Manual

This DNx-ARINC-664 User Manual is organized as follows:

- **Introduction**
This section provides an overview of the DNx-ARINC-664 avionics interface board features, device architecture, and connectivity.
- **Programming with the Low-Level API**
Chapter 3 describes low-level API commands for configuring and using the DNx-ARINC-664 series board for operating modes.
- **Tools and Diagnostics**
Chapter 4 describes available diagnostic tools and provides procedures for troubleshooting the DNx-ARINC-664 in a system.
- **Appendix - Accessories**
This appendix provides a list of accessories available for use with the DNx-ARINC-664 communication interface board.



Manual Conventions

To help you get the most out of this manual and our products, please note that we use the following conventions:



Tips are designed to highlight quick ways to get the job done or to reveal good ideas you might not discover on your own.

NOTE: Notes alert you to important information.



CAUTION! Caution advises you of precautions to take to avoid injury, data loss, and damage to your boards or a system crash.

Text formatted in **bold** typeface generally represents text that should be entered verbatim. For instance, it can represent a command, as in the following example: “You can instruct users how to run setup using a command such as **setup.exe**.”

Text formatted in `fixed` typeface generally represents source code or other text that should be entered verbatim into the source code, initialization, or other file.

Examples of Manual Conventions



Before plugging any I/O connector into the Cube or RACKtangle, be sure to remove power from all field wiring. Failure to do so may cause severe damage to the equipment.

Usage of Terms



Throughout this manual, the term “Cube” refers to either a PowerDNA Cube product or to a PowerDNR RACKtangle™ rack mounted system, whichever is applicable. The term DNR is a specific reference to the RACKtangle, DNA to the PowerDNA I/O Cube, and DNx to refer to both.



- 1.2 DNx-ARINC-664 Board Overview** The DNx-ARINC-664 is a 2 channel communications interface compatible with ARINC-664 Part 7 (a.k.a. ARINC-664 and the Airbus variant). The DNA-ARINC-664 and DNR-ARINC-664 versions are designed for UEI's Cube and RACKtangle I/O chassis respectively.
- Channels may be configured as a single A or B channel or as one dual redundant channel. The network implementation supports 10/100/1000BASE-T. Each channel may operate as transceiver, transmitter, or receiver.
- 1.2.1 ARINC-664 Receiver** In input mode, the user may time tag inputs with resolutions as low as 10 microseconds. The input automatically provides error/integrity checking, but this feature may be disabled in software if the application requires. Receive filtering of virtual link (VL), port, and error properties is also supported.
- 1.2.2 ARINC-664 Monitor** Monitor mode allows the user to capture network traffic, providing the capability of capturing select information with automatic filtering. Monitor mode will also gather a variety of statistics from the bus/network. If desired, monitor mode may be set to capture UDP network traffic statistics, regardless of whether it is configured based on the ARINC-664 protocol.
- 1.2.3 ARINC-664 Transmitter** Transmit channels automatically configure traffic shaping via Bandwidth Allocation Gaps (BAGs) that can be set for 1, 2, 4, 8, 16, 32, 64, or 128 millisecond timing. Transmission may be based on an automatic scheduler or in one-shot asynchronous mode. Both unicast and multicast virtual links are fully supported. The transmitter generates and tags consecutive Sequence Numbers.
- 1.2.4 Certification** The board is based on the Freescale 8347 processor running the DO-178 certified μ C Operating System. In PowerDNA mode, the Cube/RACK itself also uses μ C/OS, so even though the units are not certified to DO-178, the fact that the operating system already is will dramatically reduce certification time. Advanced users may also wish to implement special functions in the board's firmware which can be accessed with custom μ C code. The Cube/RACK is well supported with a variety of debugging tools, and additionally a dedicated RS-232 diagnostic port is provided on the board allowing easy access to the lowest levels of the board's functionality.
- 1.2.5 Software Support** Software for the DNx-ARINC-664 series is provided with the UEI Software Suite, which includes an easy-to-use API for Windows and Linux.



1.3 Features

The DNx-ARINC-664 features are as follows:

- Compatible with ARINC-664 Part 7 (a.k.a. ARINC-664 and the Airbus variant)
- 2 independent or 1 dual redundant channels
- 100 BASE-T default (10/100/1000BASE-T capability)
- Transmitter and/or Receiver functions
- Extensive error detection
- Extensive filtering and traffic scheduling
- Tested to withstand 5 g Vibration, 50 g Shock, -40 to +85°C Temperature, and Altitude up to 70,000 ft or 21,000 meters
- Weight of 150 g or 5.3 oz for DNA-ARINC-664; 162 g or 5.7 oz for DNR

1.4 Indicators

The Rack version of the DNx-ARINC-664 is shown in **Figure 1-1**.

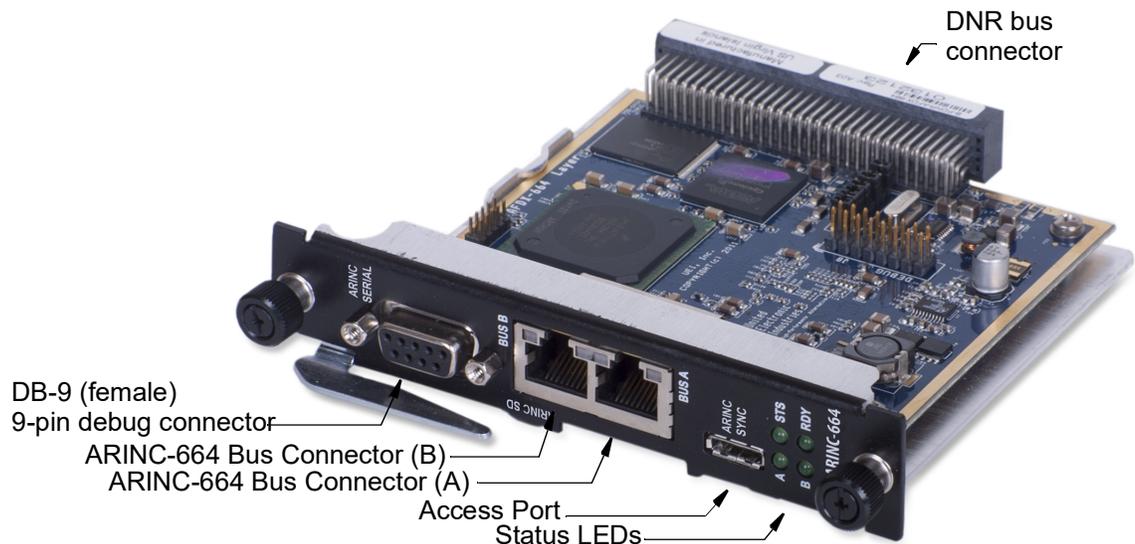


Figure 1-1 The DNR-ARINC-664 Board

1.5 Specification The technical specification for DNx-ARINC-664 is provided in Table 1-1.

Table 1-1 . DNx-ARINC-664 Technical Specifications

Configuration	
Number of channels	2: supports A only, B only or dual redundant
Ethernet BASE	10 / 100 / 1000 BASE-T
Channel functions	Transmit, Receive or Monitor
VLs supported	Up to 2000 VLs or ports with up to 664 active
Underlying Processor	Freescale 8347 running DO-178 certified OS
Receive Specifications	
Time tagging resolution	10 μ S
Error/Integrity checking	Link-level, Integrity, Sequence Number (SN)
Filtering	VL, Port, and error detection filters
Monitor Specifications	
Configuration	Selectable
Error Checking	Integrated
Statistics Gathering	Counters: PHY, ETH, IP, UDP, ARINC-664
Transmit Specifications	
Traffic shape via BAG	1, 2, 4, 8, 16, 32, 64 or 128 mS
Transmission scheduling	10, 500 μ S resolution periodic scheduling of messages; all scheduling done in firmware
Transmission configuration	Unicast and multicast addressing
Sequence Numbers	Auto-sequenced consecutive (per VL)
General Specifications	
Debugging options	via Cube/RACKtangle chassis backplane or directly to board via RS-232 port
Self-test	Power-on self-test (POST) of computer board, dual AFDX-664 loop test of network interface
Operating temperature	tested -40 $^{\circ}$ C to +85 $^{\circ}$ C
Vibration <i>IEC 60068-2-6</i> <i>EC 60068-2-64</i>	5 g, 10-500 Hz, sinusoidal 5 g (rms), 10-500 Hz, broad-band random
Shock <i>IEC 60068-2-27</i>	50 g, 3 ms half sine, 18 shocks @ 6 orientations 30 g, 11 ms half sine, 18 shocks @ 6 orientations 30 g, 11 ms half sine, 18 shocks @ 6 orientations
Humidity	5 to 95%, non-condensing
Power consumption	6 Watts, maximum



1.6 ARINC-664 Overview

The ARINC-664 standard defines an aircraft data network in 8 parts. Part 7¹ of the standard (ARINC-664P7) specifies a deterministic network on the OSI data link layer.

The UEI DNx-ARINC-664 board supports the standard ARINC-664 specification, but does not support the Boeing EDE extension.

ARINC-664 is similar to ARINC-429, ARINC-629, MIL-STD-1553 in that it allows any two avionics devices to communicate reliably and deterministically within a specific time-interval across a fault-tolerant dual-redundant bus but with faster rates of communication, less wire, and lower cost using COTS components.

1.6.1 DNx-ARINC-664 Applications

The DNx-ARINC-664 can be used to send, receive, or report traffic on an ARINC-664 network. In practice, the DNx-ARINC-664 is used for the following applications:

- Test and debug of avionics network equipment during development, such as:
 - passively monitoring traffic (data/statistics) from specific ARINC-664 ports
 - actively inserting traffic onto the bus to provide input to equipment
- Emulate real plane traffic for physical avionic equipment (e.g., display LRUs) of commercial aircraft simulator(s)

Figure 1-2 shows an example of multiple avionics devices emulated by the DNx-ARINC-664:

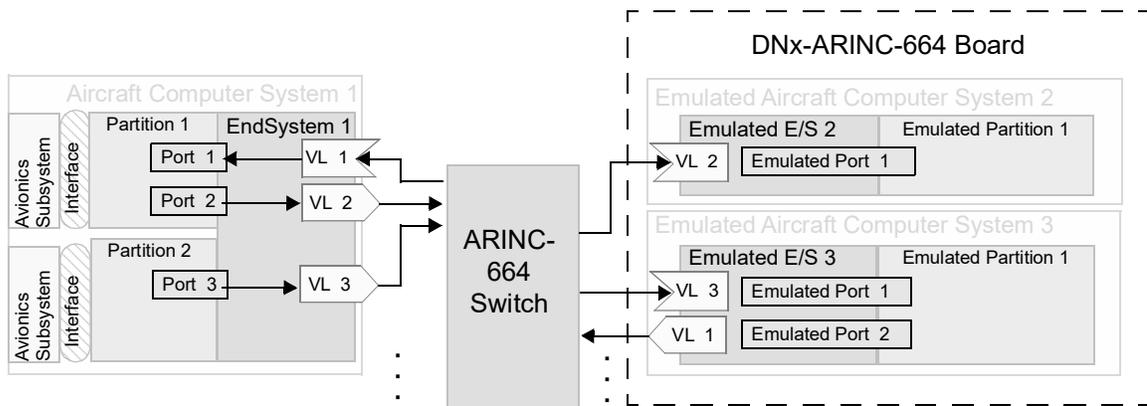


Figure 1-2 Example of the DNx-ARINC-664 in an ARINC-664 Network

The DNx-ARINC-664 provides the following capabilities in an ARINC-664 network:

- emulation of the ARINC-664 endsystem, partition, virtual links, & ports
- filtering of received message by address and error
- scheduling transmission and periodic retransmissions required by the underlying ARINC-664 communication protocol

The DNx-ARINC-664 board is configurable by XML into the mode you require.

1. Aeronautical Radio Incorporated, *ARINC Specification 664 P7-1: Aircraft Data Network Part 7 - Avionics Full-Duplex Switched Ethernet Network*. AEEC, Annapolis, MD: 2009.

Once configured, the DNx-ARINC-664 allows your program to abstract away everything to ARINC-664 messages, allowing you to communicate using a handful of simple function calls documented in the PowerDNA API.

Refer to **Chapter 2** for more information about the DNx-ARINC-664 API and XML configuration.

The remainder of this section explains the structure and behavior of ARINC-664 network nodes, as well as defines the most common attributes, terms and definitions.

1.6.2 ARINC-664 Network Overview

Figure 1-3 shows an example of aircraft network topology:

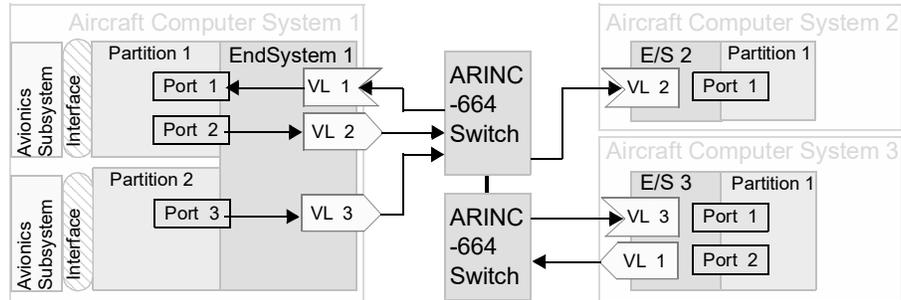


Figure 1-3 Basic ARINC-664 Network (Logical Addressing Perspective)

ARINC-664 Part 7 describes the data network as containing one or more of the following:

- **EndSystem (ES):** connects an aircraft computer system to the data network. Each endsystem must have a unique address in the network. An aircraft computer system will generally have multiple subsystems and applications, each of which is isolated within an ARINC-653 **partition**. These applications communicate with one another by transferring data over sampling or queuing (S/Q) communications **ports** defined by ARINC-653. ARINC-664 Part 7 additionally defines service access ports (SAP). Data from one **port** on one endsystem is sent across the network to a receiving port on another endsystem via a unidirectional logical channel called a **Virtual Link (VL)**.
- **Switch:** transfers (and polices) data traveling between end systems. Policing is the process of checking that traffic flowing from one ES to another ES complies with rules of ARINC-664 networks (traffic shaping) and ensures that the network is deterministic even if an ES malfunctions. Each switch pre-loads a configuration file with all endsystems and their parameters (e.g., VLs, etc).

- 1.6.2.1 Endsystems & Partitions** Each endsystem has these characteristics:
- Each endsystem must have a unique address in the network consisting of 16 bits from an 8-bit Network ID plus 8-bit End System ID and an additional 8-bit Partition ID.
 - An endsystem can have 0 to 128 VLs.

Endsystem addresses are represented in IP address notation (Section 1.6.3):

- Unicast IP addresses (for source and destination) addresses are 10.[8-bit Network Id].[8-bit End System Id].[8-bit Partition ID]. These point-to-point addresses are always from 10.0.0.0 to 10.255.255.255. For example, 10.1.2.3 is Network 1, End System 2 and Partition 3.
- Multicast IP addresses (for destination addresses) are 224.224.[VLID] with the 16-bit VLID is split into two 8-bit octets. These point-to-multi-point addresses are always from 224.224.0.0 to 224.224.255.255.

- 1.6.2.2 Virtual Link** Each Virtual Link (VL) has these characteristics:
- Each is an input or output (unidirectional).
 - Each has one or more receiving partitions (via unicast or multicast).
 - Each VL has only one source address on the network (unique sender). Each sending VL can operate in "normal" (1 S/Q ports per VL) or "subVL" mode (one VL can be assigned 2, 3, or 4 Q/S ports per subVL). Each subVL is assigned an individual subVL ID and a sending queue.
 - Each VL must send packets in the order that they are queued. A VL divided into 2, 3, or 4 sub-VL queues and sends using a round-robin algorithm.
 - Each VL can transmit once every BAG=1, 2, 4, 8, 16, 32, 64, or 128 ms.
 - SkewMax should be between 0 and 253*BAG
 - Ethernet frame length (L_{max}) should be between 64 and 1518 bytes

Virtual Links are represented in Ethernet MAC address notation (see Section 1.6.3). The destination address has a 16-bit VLID in the last two octets [03:00:00:00:[VL:ID]]. The source MAC address format contains the source (transmitter) information: [02:00:00:8-bit Network ID:8-bit Equipment ID:8-bit Bus ID (20 is A, 40 is B)].

- 1.6.2.3 ARINC-664 COM Ports** There are three types of communication ports defined in ARINC-664P7:
- Sampling (SMP) as defined in ARINC-653 for avionics data.
 - Queuing (QUE) as defined in ARINC-653 for avionics data.
 - Service Access Ports (SAP) defined in ARINC-664P7 for other uses.

In TCP/IP protocol notation this is a 16-bit UDP port from 0-65535 (see 1.6.3).

Sampling ports have the following characteristics:

- Connectionless UDP with no flow or error management.
- Data is transmitted as the payload of a single non-fragmented UDP packet. Received data is of a fixed size that is preconfigured. The largest maximum data size is 1471 bytes (when $L_{max}=1518B$).



- Multiple partitions on an endsystem may read from the same sampling port's received message buffer. This buffer is not cleared upon read. This receive buffer has a freshness indicator. Only a single buffer exists and is overwritten when a new message arrives.

Queuing ports have the following characteristics:

- Queuing ports use connectionless UDP with no acknowledgment.
- Data is transmitted as the payload of a single fragmentable UDP packet. Data size is set at transmission time. Maximum data size is 8192 bytes.
- When a message is completely received, it is added to a received message FIFO queue (depth preconfigured) where it can be retrieved by any partition. Retrieving a message removes it from the buffer.
- On buffer overflow (configured receive buffer too small) the message is discarded and an error status is provided to the receiving partition.
- IP fragmentation occurs when data exceeds $L_{max} - 47$ bytes and is transmitted in fragments (up to 8 fragments max).
 As an example, an 8192 byte queuing message exceeds 1471 bytes and would require $(8+8192) / 1479$ bytes = 6 fragments to transmit when $L_{max} = 1518$. Refer to **Figure 1-5** for packet descriptions.
- The loss of a fragmented packet during reception causes the entire reassembly buffer to be discarded.

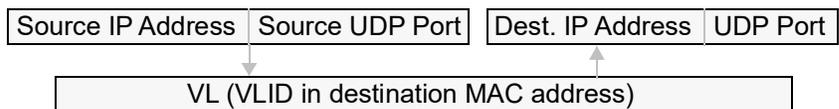
Service Access Ports have the following characteristics:

- SAPs provide UDP services to communicate with a Compliant Network, and access to Compliant Network is done through a gateway or router.
- Messages are identical to Queuing messages, but the destination address and port are user-configurable and translated by the Gateway.
- SAPs are commonly used for management functions that include an SNMP agent, booting from TFTP, or ARINC 615A data loading.

1.6.2.4 Connections

On an ARINC-644 network, unique connections are identified as follows:

- Sampling and queuing communication port messages are unique for the quintuplet $srcIP:srcPort + dstMAC(VL):dstIP:dstUdpPort$ below:



- A transmitting service access point port (SAP) is identified by the quadruplet $srcMAC(VL):srcIP:srcPort \rightarrow dstMAC(VL)$. The destination $dstIP:dstPort$ is set by the user application for a Compliant Network. A receiving SAP is configured as $interface:dstMAC(VL):dstIP:dstPort$.



1.6.3 OSI Model Structure

The following descriptions are in reference to the OSI 7-layer model for network communication. Refer to **Figure 1-4** for layer stack:

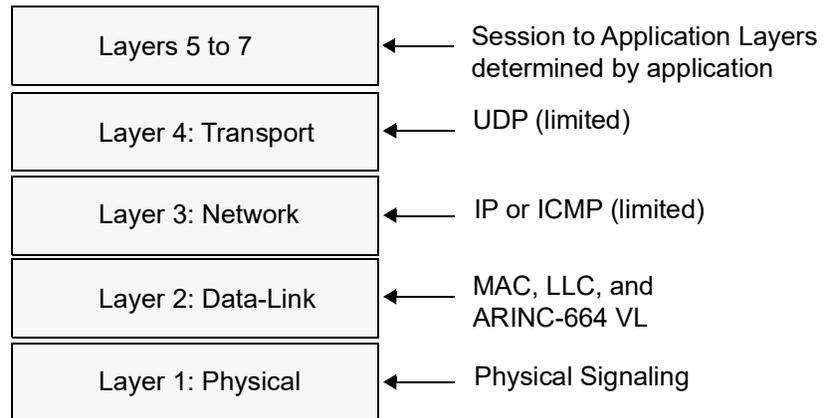


Figure 1-4 The OSI Model for ARINC-664 Systems

1.6.3.1 OSI Layer 1

Layer 1 is the **Physical** layer defining the hardware connection between sender and receiver. Note that it transports *raw bits* rather than *logical data*. ARINC 667 specifies that the IEEE 802.3 Ethernet standard shall be used for the physical layer. Each ARINC-664 device has a pair of physical interfaces, one used for channel/bus/network A, and for redundancy one for network B.

In practice the DNx-ARINC-664 implements two independent bus connectors that are IEEE 802.3ab 10/100/1000BASE-T Ethernet ports, that accept twisted-pair copper wiring with both receive and transmit, or “Ethernet cable”, as explained in architecture Section 1.7; for wiring, see Section 1.8.

1.6.3.2 OSI Layer 2

Layer 2 is the **Data Link** layer. The data-link layer transfers entire *frames* of *logical data* from sender to receiver, at least as far as Layer 3 is concerned. In the Ethernet standard the Data Link layer (DLL) is composed of two parts: the medium access control (MAC) and logical link control (LLC) sublayer.

The MAC sublayer provides mechanisms to access the physical medium including packet switching and scheduling to transmit data (as Ethernet frames) assembled by the LLC’s multiplexing of data from Layer 3. The physical layer and MAC sublayer are designed to be embedded in hardware, so that changing from twisted-pair copper to fiber or other medium is done by changing the COTS part. Higher layers are implemented in software.

The IEEE 802.3ab MAC sublayer specifies use of the carrier sense multiple access with collision detection (CSMA/CD) protocol to choose which device will have exclusive access to the physical medium to transmit, but CSMA/CD alone is non-deterministic due to collisions that can cause indefinitely long contention for the transmission medium by devices with long transmissions. To ensure deterministic on-time delivery of data between devices on the avionics network ARINC-664 uses the Virtual Link mechanism in the data-link layer. The Virtual Link is a logical communication channel which guarantees bandwidth by limiting one transmission of 1518 bytes or less (<122 μs at 100 Mbps) once every 1-128 ms; Virtual Links are explained in more detail in later sections. ARINC-664 also adds a frame sequence number to the end of the Ethernet frame for redundancy management of identical frames sent across bus A and B.



- 1.6.3.3 OSI Layer 3** Layer 3 is the **Network** layer. ARINC-664 specifies a connectionless communication network with no routing (gateways for SAP are not considered) with a very restricted IP header carrying either a ICMP ECHO or a UDP datagram. The IP header must have the following flags set or be discarded: Version=4, Type of Service=0, Flag={0,1,2}, TTL=1 (hop), Protocol={1:ICMP,17:UDP}.
- The Internet Control Message Protocol is restricted to ECHO datagrams only corresponding to ICMP type={0,8} code=0, which are used to “ping” an endsystem to see if it is online; all other types and codes are not used.
- 1.6.3.4 OSI Layer 4** Layer 4 is the **Transport** layer. ARINC-664 defines UDP as the only transport layer protocol to carry a data payload. The UDP CRC is not used, and the length can be set to as low as the 4 byte header with no data up to 4+8192 bytes where 8192 bytes is defined as the maximum payload for a queuing port.
- 1.6.3.5 OSI Layers 5 to 7** Layers 5 to 7 above the Transport layer are not policed by ARINC-664 switches, but the respective data formats are found in other avionics standards.



1.6.4 ARINC-664 Packet Structure

Figure 1-5 illustrates an ARINC-664 network packet as a UDP datagram.

The traditional Ethernet UDP datagram structure varies for ARINC-664 packets in the following ways:

- An Ethernet frame contains an ARINC-664 sequence number (SN), 0-255.
- Ethernet MAC addresses use the ARINC-664 addressing scheme.
- Padding is added to ensure that the Ethernet frame size is at least 64 bytes, and that the Ethernet length field is at least 46 bytes. A sampling port carrying 1 byte payload: $14+20+8+1_{\text{payload}}+16_{\text{padding}}+1+4 = 64$, for example, or a fragment carrying 1 byte: $14+20+1+24_{\text{padding}}+1+4 = 64$. The minimum frame size is 64 bytes whether the packet is carrying up to 17 bytes of UDP payload: $14+20+8+17_{\text{payload}}+1+4 = 64$ byte frame.
- The IP header is simplified.
 - VER=4, ToS=0, TTL=1, Protocol is only UDP (or ICMP ECHO)
 - IP addresses use ARINC-664 EndSystem & Partition addressing scheme.
- The UDP CRC is ignored. UDP payload may not exceed 8192 bytes.

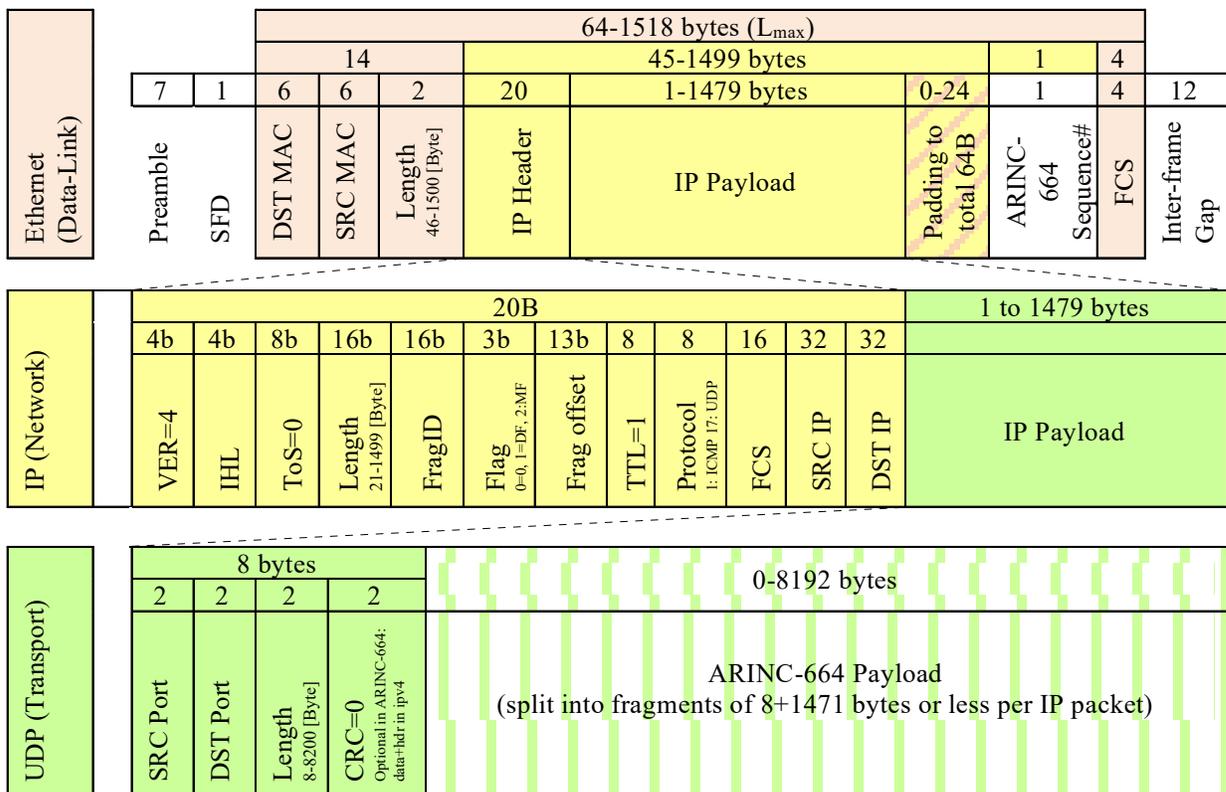


Figure 1-5 ARINC-664 Packet Structure



1.6.5 Bandwidth

Bandwidth utilization for 17 byte and 1471 byte payloads is:

- 17 byte payload: $100\text{Mbps}/(8\text{bits})/(84\text{ octets/frame})=148810\text{ frames/sec}$
or effectively $148810\text{ FPS} * 17\text{ Bytes/frame} = 2,529,770\text{ B/s}$ (2.4 MiB/s)
- 1471 byte payload: $100\text{Mbps}/(8\text{ bits})/(1538\text{ octets/frame})=8127\text{ frames/sec}$
or effectively $8127\text{ FPS} * 1471\text{ Bytes/frame} = 11,954,817\text{ B/s}$ (11.4 MiB/s)

8127 frames / 1000 msec is about 8 sampling messages of 1471 bytes per ms,
and at 17 bytes that is 148 sampling messages per millisecond.



1.7 Device Architecture

This section describes the hardware used in the DNx-ARINC-664 board.

A block diagram of the board is shown in **Figure 1-6**.

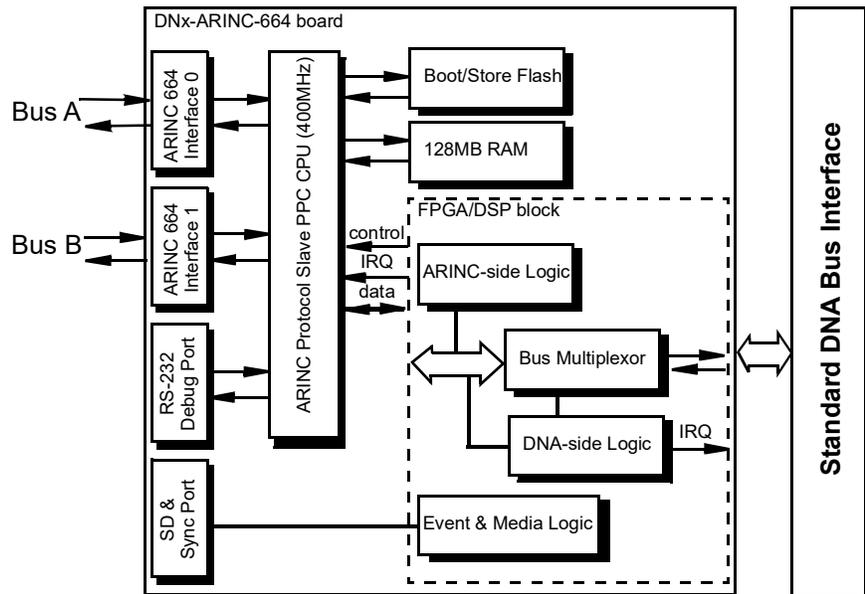


Figure 1-6 DNx-ARINC-664 Logic Block Diagram

The front-end of the DNx-ARINC-664 provides five ports and four status LEDs:

- Two TIA/EIA-568 sockets to connect ARINC-664 bus A/B to the board
- One RS-232 debug port for firmware updates or debugging
- One SD-card port (for future use with storage media logic)
- One SYNC port (for future use with triggering and events logic)
- Four LEDs to provide status information

The TIA/EIA-568 sockets for Bus A and Bus B are wired into a 10/100/1000BASE-T Ethernet chip set that is forced into 100BASE-TX mode without auto-negotiation that allows for a fast initialization.

The Ethernet ports are wired into a 400MHz processor that manages communication with the ARINC-664 bus. This processor operates independently and is connected to a logic chip that interfaces it to the Cube/RACK's DNA bus which the card is plugged into. This logic chip provides an access mechanism to exchange commands, data, and interrupts.

To indicate that a new command/data is ready for exchange, either the processor or the Cube/RACK's main CPU (managing the DNA bus) will write to a doorbell register which triggers an interrupt to perform the exchange. The logic chip also provides access to the SD & SYNC ports.

The 400MHz processor is additionally directly connected to the RS-232 debug port, from which it can be debugged or updated.

1.7.1 Device RTOS

The DNx-ARINC-664 incorporates a DO-178 certified real-time operating system "MicroC/OS" that facilitates ARINC-664 emulation. Important processes like the reception routine and statistic collection are described in "Device RTOS Processes" on page 33.

1.8 Wiring & Connectors

The following ports are located on the front-end of the DNx-ARINC-664 board:

- RS-232 female connector to debug the DNx-ARINC-664.
- Dual TIA/EIA-568 female sockets accepts Category 5/6 straight-through unshielded twisted-pair (UTP) wire - the same Fast Ethernet or Gigabit Ethernet copper wiring that is used to connect PCs to LANs.

Pin	Signal
1	Tx+
2	Tx-
3	Rx+
4	(none)
5	(none)
6	Rx-
7	(none)
8	(none)

The socket closer to the RS-232 port corresponds to Bus B. The socket closer to the SYNC/LED port is Bus A. See **Figure 1-1** for bus labeling. Note that to connect to an avionics interface that uses an optic fiber link please contact Technical Support for a recommended media converter.

- SYNC port hardware for synchronization and triggering.

1.8.1 Connecting to the ARINC-664 Network

To connect the DNx-ARINC-664 board, use the following procedure:

1. Verify the DNx-ARINC-664 board is powered down.
2. Connect the network interface(s) to the switch(es) (**Figure 1-7**):
 - Connect “Bus A” network interface port to the Switch for Bus A.
 - Connect “Bus B” network interface port to the Switch for Bus B.
3. Apply power to the RACK or Cube IOM. The board will link to the network.

Note: The board performs a fast link, and disconnecting either network cable will disable that network interface until it is power-cycled.

4. Confirm that network link lights are on as orange (10/100Mbps) or green. You are now connected to the ARINC-664 network.

Optionally, you can connect to the serial debug port using MTTTY (or PuTTY) at 57600 baud, no parity, 8 data bits, 1 stop bit to confirm connectivity.

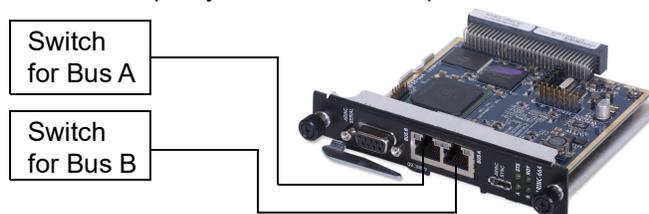


Figure 1-7 Connection diagram for the DNx-ARINC-664

NOTE: It is preferred that both the Bus A and Bus B Switch be certified ARINC-664 network switches that are properly pre-configured for the avionics network. COTS Ethernet switches will act as hubs which degrade overall performance.



Chapter 2 Programming with the Low-level API

This chapter provides the following information about programming the DNx-ARINC-664 using the low-level API:

- About the Low-level API (Section 2.1)
- Low-level Functions (Section 2.2)
- Tutorial (Section 2.3)
- UEI ARINC-664 XML Configuration (Section 2.4)

2.1 About the Low-level API

The low-level API provides direct access to the DAQBIOS protocol structure and registers in C. The low-level API is intended for speed-optimization, when programming unconventional functionality, or when programming under Linux or real-time operating systems.

UEI also offers a high-level Framework API for use when programming in Windows OS; however, DNx-ARINC-664 is not supported in the Framework and must be programmed using the low-level API.

For additional information regarding low-level API, refer to the “*PowerDNA API Reference Manual*” located in either of the following directories:

- On Linux systems:
 <PowerDNA-x.y.z>/doc
- On Windows systems:
 Start » All Programs » UEI

2.2 Low-level Functions

Low-level functions are described in detail in the *PowerDNA API Reference Manual*. Table 2-1 provides a summary of DNx-ARINC-664-specific functions.

Table 2-1 Summary of Low-level API Functions for DNx-ARINC-664

Function	Description
DqAdv664AddPort	Adds a Sampling, Queuing, or Service Access Port to a VL.
DqAdv664AddVL	Adds a VL.
DqAdv664BusControl	Controls ARINC bus parameters.
DqAdv664ClearConfig	Deletes the entire configuration.
DqAdv664ConfigEvents	Configures asynchronous events for the DNx-ARINC-664.
DqAdv664Enable	Enables operation.
DqAdv664EnableVLPort	Enables or disables VLs or Ports.
DqAdv664GetBusStat	Gets transceiver statistics accumulated during operation.
DqAdv664GetDeviceInfo	Gets the device information for the DNx-ARINC-664 board as stored in the pAR664_DEV_INFO structure.



Table 2-1 Summary of Low-level API Functions for DNx-ARINC-664 (Continued)

Function	Description (Continued) (Continued)
DqAdv664GetHandle	Gets a VL or port handle by their ID.
DqAdv664RecvMessage	Gets a Sampling or Queuing message.
DqAdv664RecvMessageHdr	Gets a Sampling, Queuing, or SAP message and headers.
DqAdv664SendMessage	Puts a Sampling or Queuing message.
DqAdv664SendMessageHdr	Puts a Sampling, Queuing, or SAP message and headers.
DqAdv664SendScheduleTable	Sends the transmitter scheduler table for high performance bin-based scheduling.
DqAdv664SetConfig	Configures from file.
DqAdv664ValidateVlPortCfg	Validates parameters in a AR664_VL_CFG or AR664_PORT_CFG and performs boundary-checking functions.
DqAdv664VlPortStatus	Gets VL or port status for a specific handle.



2.3 Tutorial

The following tutorial provides a brief overview of how to set up and use your DNx-ARINC-664 using the low-level API.

For best results, use this tutorial in conjunction with an actual code example, which can be found in either of the following directories:

- <PowerDNA-x.y.z>/src/DAQLib_Samples/Sample664_xml (Linux)
- %PDNAROOT%\Examples\Visual C++\ARINC\Sample664_xml (Windows).

The following topics are explained in this tutorial:

- Initialization
 - Initializing the Cube or RACK and enabling DNx-ARINC-664 board(s).
- Configuration
 - Clearing any existing configuration from previous runs.
 - Adding each VL and associated Port with a function call.
 - Adding all VLs and associated Port from an XML file.
- Operation: Send / Receive Messages
 - Sending and receiving messages in simple mode or VMap mode.
- Stop Cleanly
 - Disabling boards cleanly.

2.3.1 Initialization

To initiate communication with the RACK or Cube, you must first get a DAQLib handle for the IOM by calling DqOpenIOM():

```
// Connect to the IOM and obtain a library handle for the connection
DqOpenIOM("192.168.100.2", DQ_UDP_DAQ_PORT, 1000, &hd, &DQRdCfg);
```



2.3.2 Configuration Prepare to configure the DNx-ARINC-664 card by clearing any existing configuration from its memory:

```
// Clear the configuration on both bus A and B
DqAdv664ClearConfig(hd, DEVN, AR664_VL_USE_A|AR664_VL_USE_B);
```

Configure the VLs and ports by creating an XML configuration file and specifying its path to `DqAdv664SetConfig()` call as shown in the following code snippet.

NOTE: To create the XML file, use the UEI ARINC-664 Configurator tool or use one of the <config>.xml samples in `Sample664_xml` as a template. Refer to “UEI ARINC-664 XML Configuration” on page 29 for a list of programmable configuration attributes and for more information about the ARINC-664 Configurator.

```
// Enable both ARINC-664 network interface A and B (bus A and B)
// and create variables for the next setconfig call

AR664_ARCFG AR664NetworkInterfaceConfiguration = {
    (AR664_CTRL_ENABLE_A|AR664_CTRL_ENABLE_B),
    (AR664_CTRL_ETH100_A|AR664_CTRL_ETH100_B)};

int size_tbl;
AR664_CFG_HANDLES *handle_tbl;

// Set the configuration with the above network interface config,
// VL/Ports from my_file.xml, and where to return a handle table and size
DqAdv664SetConfig(hd, devn, 0, 0, &AR664NetworkInterfaceConfiguration,
    "my_file.xml", &size_tbl, &handle_tbl);
```

The `handle_tbl` array of `AR664_CFG_HANDLES` contains the ARINC-664 VL/Port handle for each entry, which will allow you to address those ports later. See the definitions for `AR664_VL_CFG` and `AR664_PORT_CFG` in `powerdna.h` for details.

You can add more VLs/ports later at runtime using the `DqAdv664AddVL()`, `DqAdv664AddPort()`, and `DqAdv664ValidateVlPortCfg()` function calls described in the *PowerDNA API Reference Manual*. The maximum number of active ports is 664; processing limits this to 128 ports at medium load, and 64 ports for heavy load for revision 1 boards.

`DqAdv664SetConfig()` returns a set of VL/Port handles, and with these you can optionally configure transmission VMap or reception aEvent modes; these modes are explained in Section 2.3.3.

Finally, enable operation to allow the card to receive or send messages as follows:

```
// Enable operation on a DNx-ARINC-664 board
DqAdv664Enable(hd, 0, AR664_VL_USE_A|AR664_VL_USE_B);
```



2.3.3 Send/Receive Messages

The DNx-ARINC-664 board can send or receive messages in three ways: (1) Immediate mode; (2) Variable Data Map or VMap+; (3) Asynchronous mode. The difference between the modes is described below with code snippets in the following sections. **Figure 2-1** is provided for reference:

- **Immediate mode:** each function call is directed to a single ARINC-664 port. The function call gets received data or puts transmit data and returns port status. Practical for use with tens of SMP/QUE/SAP/ICMP ports.
- **VMap/VMap+:** each variable-size data map contains multiple receive & transmit ARINC-664 ports that are refreshed simultaneously per function call. Designed to perform a bulk update of sampling ports of small data size in a single call and is ideal for frequently refreshed transmit ports. Each legacy VMap requires a fixed list of ARINC-664 ports to be configured before `DqAdv664Enable`, and is practical for tens to a hundred ports. VMap+ allows ports to be set dynamically at runtime (VMap is only at configuration time) and practical for refreshing hundreds of ports.
- **Asynchronous** Events or aEvent mode for ARINC-664 reception only: the IOM can forward messages received from the ARINC-664 bus directly to the PC, but requires a separate listening thread on the PC.

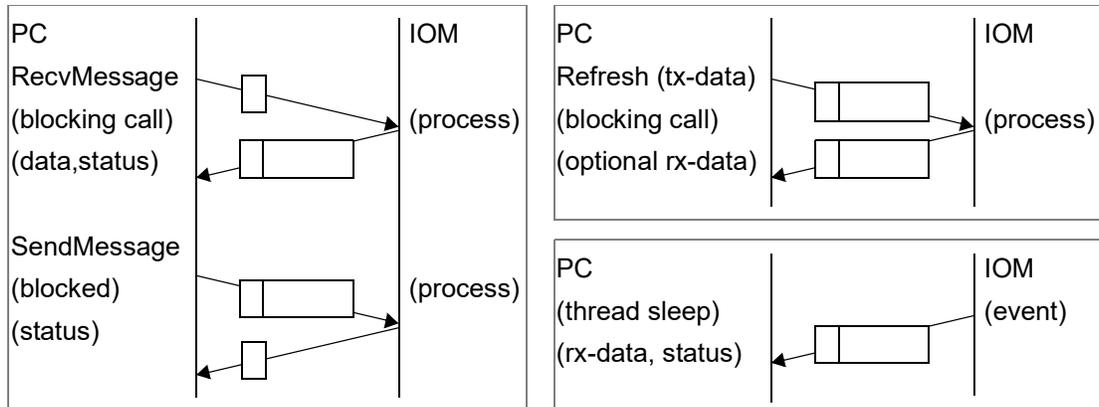


Figure 2-1 Immediate (left), VMap (top right), aEvent (lower right)



- 2.3.3.1 Immediate Mode** To send messages, call `DqAdv664SendMessage()` using the ARINC-664 port handle. The following example shows how to do this for our previously-configured sampling port:

```
// Set up variables to return send status, and a message

int written;
uint32 available, status;
uint8 message_s[AR664_MAX_MSG_SZ_N] = "my message";

// Send the message

DqAdv664SendMessage(hd, 0, myTxP1hdl, 0, myTxP1.d_size,
                    message_s,
                    &written, &available, &status);
```

For this example, let's assume that we also set up a VL to receive, which has a corresponding Port named `myRxP1hdl`.

To receive messages, call `DqAdv664RecvMessage()` for the port as described for the following sampling port:

```
// Set up variables to return send status, and a message buffer

int received;
uint32 available, status;
uint8 message_r[AR664_MAX_MSG_SZ_N];

// Receive a message into the message buffer

int received, available, status;
uint8 message_r[AR664_MAX_MSG_SZ_N];
DqAdv664RecvMessage(hd, 0, myRxP1hdl, 0, myRxP1.d_size,
                    message_r,
                    &received, &available, &status);
```

If a message is available, it will be stored in the `message_r` buffer.

The `status` variable returns the result of the send or receive. Refer to the API documentation for meaning of the bits, along with how to interpret the `written` and `available` variables; variables' meaning depends on the port type.

Immediate mode works on all platforms and is designed for debug and test.



2.3.3.2 VMap and Vmap+ Introduction

Rather than receiving or transmitting data from each port, it is possible to set up a variable-size data map to receive and transmit multiple messages at once.

VMap/VMap+ is described in detail in the “PowerDNx Protocol Manual”. Using a VMap/+ consists of the following function calls (shown in the next two sections):

- Initial configuration of a VMap/VMap+:
 - a. set up VMap parameters
 - b. add input/output channels (fixed for VMap, dynamic for VMap+)
 - c. start the VMap
- Operation:
 - d. schedule data to write upon next refresh from channel(s)
 - e. schedule data to read upon next refresh from channel(s)
 - f. refresh (see Figure 2-1)
 - g. read retrieved data from input channel (returned in reply to refresh)
- Stop and close the VMap

Both (a) and (b) are performed in the PowerDNA Library on the PC and that data map is eventually built and sent to the IOM at (c) when the start command is called.

The number of input and output channels (up to 64) from (b) are thereafter fixed and in that VMap control even if no data is scheduled to be sent/received in (d)/(e) for that channel upon refresh (f).

The DaqBios command for VMap Refresh has the process shown in Figure 2-1 and the packet format shown in Figure 2-2:

14	20	8	16	8	4	(variable)	4
ETH	IP	UDP	DQPKT Header	VMap Header	VMap Flags	VMap control+data (see below)	FCS

Figure 2-2 DaqBios packet format for VMap refresh

Extended VMap can be configured send up to 8 fragments of 1500 bytes (when initialized within a 1518 byte Ethernet frame) that allows for a maximum VMap payload of 11764 bytes that is further partitioned into up to 64 I/O “channels”.

For the DNx-ARINC-664, each added channel corresponds to an ARINC-664 port handle. The format of the payload is: *output channel size* (and tx port handle for VMap+) array, *input channel size* (and ARINC-664 receiving port handle for VMap+) array, and output data of length specified in *output channel size* earlier.



For traditional **VMap** each ARINC-664 receive/transmit port handle added as an input/output channel is put into a transfer list. Scheduling data to write and be read can only be performed for those added ARINC-664 handles and only requires specifying the size to be written (and data in case of write) or read:

2-11764 bytes											
2-128 Bytes				2-128				(sum of out_size[0] thru [63])			
2	2	2	2	2	2	2	2	out_size[0]	[1]	...	[63]
out_size[0]	out_size[1]	out_size[...]	out_size[63]	in_size[0]	in_size [1]	in_size [...]	in_size [63]	out_data[0]	out_data[1]	...	out_data[1]

Figure 2-3 VMap control & data

Figure 2-3 shows the VMap control+data portion of Figure 2-2’s Refresh command sent from PC to IOM. The output and input array indexes correspond to the channels added in (b) and set in (c) is always of fixed size until the VMap that they are assigned to is destroyed. For example, for 64 input + 64 output channels they use up the first 128+128 bytes of 11764 bytes, even if the data transferred for most channels is mostly 0, leaving only 11508 bytes for output channel data. The sizes for output channels and data to write are assigned in (d), and the sizes for input channels to return with the refresh’s reply are assigned in (e).

The reply from IOM to PC returns a packet with the size of the read data (up to in_size[] specified in the command) followed by the data itself for that channel.

Up to 256 VMaps, each with 0-64 input and 0-64 output channels, can be assigned at configuration time. However, for hundreds of ARINC-664 ports, of which only a few are frequently updated, traditional VMap is not as efficient as VMap+, since VMap+ gives control to which of thousands of ARINC-664 ports to refresh.

For **VMap+**, channels are added with a special flag that allows the ARINC-664 handle to be specified at (d) or (e) using a VMapPlus function call. As seen in Figure 2-4 the extra control information can create a larger header.

4-11764 bytes											
4-256 Bytes				4-256				(sum of out_size[0]..[63])			
2	2	2	2	2	2	2	2	out_size[0]	[1]	...	[63]
out_size[0]	out_port[0]	...	out_port[63]	in_size[0]	in_port [0]	...	in_port [63]	out_data[0]	out_data[1]	...	out_data[1]

Figure 2-4 VMap+ control & data payload

VMap+ is useful in providing finer-grained control over ARINC-664 ports. The VMAP+ is refreshed at a particular time or in a particular order, which provides more efficient use of bandwidth and IOM processing capability.



VMap and Vmap+ should be used when multiple ARINC-664 ports are to be updated simultaneously on the following platforms:

- (a) ARINC-664 ports used for transmit on PC in Slave mode
- (b) ARINC-664 ports for both transmit and receive on UEIPAC

The examples that follow show how to use VMap and VMap+ in practice.

2.3.3.3 Variable Data Map (VMap) This section exemplifies how to configure, operate, and close a VMap. **Configuration**

To create a new VMap, call:

```
// Create & initialize a VMap (extended capacity VMap+)
int vmapid = -1; // negative value indicates invalid

DQ_RTMAP_PARAM vmapparam;
vmapparam.max_payload_sz = DQ_MAX_PAYLOAD_FRAG;
vmapparam.mtu = DQ_MAX_ETH_SIZE_100;
vmapparam.refreshRate = 0; //0 means it is not implemented in ARINC-664
DqRtVmapInitEx(hd0, &vmapid, &vmapparam);
```

Any DNx-ARINC-664 handle provided by `DqAdv664AddPort` or `DqAdv664SetConfig` (or even `DqAdv664AddV1`) can be addressed as a VMap channel. Add a channel for any ARINC-664 handle by calling `DqRtVmapAddChannel`:

```
// Command PowerDNA Library to add up to 64 channels to the new VMap
// The handles "myRxPlhdl" and "myTxPlhdl" are from previous section
int flags = 0;
DqRtVmapAddChannel(hd0, vmapid, DEVN, DQ_SS0IN, &myRxPlhdl, &flags, 1)
DqRtVmapAddChannel(hd0, vmapid, DEVN, DQ_SS0OUT, &myTxPlhdl, &flags, 1)
```

Start the operation of the DNx-ARINC-664 with `DqAdv664Enable()` to command the DNx-ARINC-664 to begin to use the ARINC-664 network if not already done so.

Start the VMap with the configuration and channels requested above, sending the configuration from PC to IOM over the network, by calling:

```
// Start the VMap
DqRtVmapStart(hd0, vmapid);
```

Operation

Now the VMap is configured, operation involves preparing to Refresh the VMap as explained in steps (d) through (g) in the previous section.

Begin by using this convenience function to reset everything to 0 (otherwise you must reset all channels to 0 yourself):

```
// Reset all channel sizes to 0
DqRtVmapInitOutputPacket(hd0, vmapid);
```



To write data to a transmitting ARINC-664 port, first request it in the VMap packet:

```
// Request how much data to write for channel for "myTxPlhdl"

uint8 out_data[1440];
len = sprintf((char*) (out_data), "0123456789");

DqRtVmapWriteOutput(hd0, vmapid, DEVN, myTxPlhdl, len, out_data);
```

The `DqRtVmapWriteOutput` call will return the number of bytes still available in the VMap packet (`ret`). When (`ret!=len`) your request has been denied; if (`ret<0`) then an error has occurred (for detail, call `DqTranslateError(ret)`). The VMap request has been prepared so it can be sent with `DqRtVmapRefresh`.

To read a message from an ARINC-664 port, first request it in the VMap packet:

```
// Request to read data to get using channel for "myRxPlhdl" in reply

DqRtVmapRequestInput(hd0, vmapid, DEVN, myRxPlhdl, 8192);
```

The `DqRtVmapRequestInput` call will return the number of bytes still available in the VMap packet (`ret`) as with `DqRtVmapWriteOutput`. This VMap request has been prepared so it can be sent with the next `DqRtVmapRefresh`.

Note that the above `DqRtVmapWriteOutput` and `DqRtVmapReadInput` only create requests but do not send the actual VMap packet to the IOM. To send the actual packet use the `VMapRefresh` call seen below:

```
// Send the VMap refresh request (sends the actual packet prepared above)

DqRtVmapRefresh(hd0, vmapid, 0);
```

Note that the Refresh command only operates on the data of an ARINC-664 port and does not currently retrieve the status, number of messages in queue, or SAP headers as the simple messaging mode does. Requesting data from a queuing port only receives a single message from the queue, not the whole queue.

The VMap refresh command packet (Figure 2-1 and Figure 2-3) contains both the data transmitted, (i.e., including `out_data`) and any request for data to be returned with the VMap reply from the IOM.

To read the input received after a refresh you must call `DqRtVmapReadInput`:

```
// Read the data returned by the DqRtVmapReadInput request above

uint8 data[1440];
int rxsz = 0;
DqRtVmapReadInput(hd0, vmapid, DEVN, myRxPlhdl, 1440, &rxsz, data);
printf("Input channel %x data: %d (%s)\n", myRxPlhdl, rxsz, data);
```

Repeat the above sequence of `DqRtVmapWriteOutput`, `DqRtVmapReadInput`, `DqRtVmapRefresh`, `DqRtVmapReadInput` until the simulation is complete.



Stop & Close

Once your simulation is complete, call `DqAdv664Enable()` to disable the DNx-ARINC-664 network operations, and then stop and clean up the VMaps with the calls:

```
// Clean up VMap

finish_up:
if (vmapid) {
    DqRtVmapStop(hd0, vmapid);           // Stop VMap
    DqRtVmapClose(hd0, vmapid);         // Destroy it
}
```

For more information, refer to the sample code, “SampleAsync664”.

2.3.3.4 VMap+

To create an Extended VMap+ call (non-UEIPAC):

```
// Prepare an extended VMap

int vmapid;
DQ_RTMAP_PARAM vmapparam = {12000,1518,0.1};
DqRtVmapInitEx(hd0, &vmapid, &vmapparam);
```

Add a VMap channel corresponding to an ARINC-664 handle or use any unique number to identify the channel.

Add a VMap channel for any handle by calling:

```
// Add VMap channels (no ARINC-664 port handles are used, just unique #s)

int vmapch_tx[1] = {0};
int vmapch_rx[1] = {1};

int flags = DQ_VMAP_SPEC_CHANNEL;
DqRtVmapAddChannel(hd0, vmapid, DEVN, DQ_SS0IN, vmapch_rx, &flags, 1)
DqRtVmapAddChannel(hd0, vmapid, DEVN, DQ_SS0OUT, vmapch_tx, &flags, 1)
```

You can then start the operation of the DNx-ARINC-664 with `DqAdv664Enable()` to allow the DNx-ARINC-664 to interact with the network.

Start the VMap by calling:

```
// Start the VMap

DqRtVmapStart(hd0, vmapid);
```

To write data to a transmitting ARINC-664 port, first request it in the VMap packet:D

```
// Request to write data to the previously added VMap channel

uint8 out_data[8192];
len = sprintf((char*)(out_data), "0123456789");
DqRtVmapPlusWriteOutput(hd0, vmapid, DEVN, vmapch_tx[0], myTxP1hdl, len,
out_data);
```



The `DqRtVmapWriteOutput` call returns the number of bytes still available in the VMap packet (`ret`). When (`ret!=len`) your request has been denied; if (`ret<0`) then an error has occurred (for detail, call `DqTranslateError (ret)`). The VMap request is prepared so it can be sent with `DqRtVmapRefresh`.

To read a message from an ARINC-664 port, first request it in the VMap packet:

```
// Request to read data to the previously added VMap channel
DqRtVmapPlusRequestInput(hd0, vmapid, DEVN, vmapch_rx[0], myRxPlhdl,
8192);
```

`DqRtVmapPlusRequestInput` returns the number of bytes still available in the VMap packet (`ret`) as with `DqRtVmapWriteOutput`. This VMap request has been prepared so it can be sent with the next `DqRtVmapRefresh`.

Once all input and output channels are requested, perform the refresh:

```
// Send the VMap request (sends the actual packet prepared above)
DqRtVmapRefresh(hd0, vmapid, 0);
```

Note that the Refresh command only accesses the ARINC-664 port's message payload and does not retrieve the status, remaining messages in queue, or SAP headers as the simple messaging mode does. Requesting data from a queuing port only receives a single message from the queue, not the whole queue.

The VMap refresh command packet (Figures 2-1 and 2-3) contains both the data transmitted (i.e., including `out_data`) and any request for data to be returned with the VMap reply from the IOM. The reply contains only ARINC-664 message data without any status, remaining messages in queue, or SAP headers as seen with simple messaging mode. The status and number of messages may be retrieved with the `DqAdv664PortMsgStatus` and `DqAdv664VLPortStatus` commands. It is useful to run these commands to see which ports are worthwhile refreshing.

To read the input received after a refresh you must call `DqRtVmapReadInput`:

```
// Read the data returned by the DqRtVmapReadInput request above
DqRtVmapReadInput(hd0, vmapid, DEVN, vmapch_rx[0], 8192, &rxsz, data);
printf("Input channel %x data: %d (%s)\n", myRxPlhdl, rxsz, data);
```

Repeat the sequence of `DqAdv664VLPortStatus`, `DqRtVmapPlusWriteOutput`, `DqRtVmapPlusReadInput`, `DqRtVmapRefresh`, `DqRtVmapReadInput`.

Once your simulation is complete, call `DqAdv664Enable()` to disable DNx-ARINC-664 network operations, and then stop and clean up the VMaps with the calls:

```
// Clean up VMap
if (vmapid) {
    DqRtVmapStop(hd0, vmapid);           // Stop VMap
    DqRtVmapClose(hd0, vmapid);        // Destroy it
}
```



2.3.3.5 Asynchronous Events or aEvent Mode Asynchronous events were implemented to improve efficiency in receiving large quantities of data quickly from the ARINC-664 bus and transferring them to the PC. The UEIPAC does not use aEvent mode because all traffic is local.

Asynchronous events are described in detail in the *PowerDNx Protocol Manual*. In summary, configuring and using asynchronous events follows this process:

- Initial configuration:
 - a. create an independent IOM handle and port for aEvents only
 - b. configure the DNx-ARINC-664 with a list of ARINC-664 receive ports
 - c. start listening for events on that list
 - d. start a thread to process asynchronous events coming from IOM
- Operation on the DNx-ARINC-664:
 - e. receive a message from the ARINC-664 bus
 - f. find the receiving port in the aEvents list
 - g. encapsulate the message data in an aEvent packet
 - h. queue the IOM-CPU to send the aEvent packet
 - i. send the aEvent packet to the PC (see Figure 2-1)
- Operation on the host PC:
 - j. wake sleeping thread (started at d) to receive the aEvent packet
 - k. unencapsulate ARINC-664 message(s) from the aEvent packet
 - l. process ARINC-664 message(s), then sleep until next packet
- Cleanup:
 - m. disable events and clear the aEvent port list

We have seen that Immediate and VMap/VMap+ function calls block the application on the PC while sending a command to the IOM, which in turn block the IOM while it polls the DNx-ARINC-664 for data for 1 to 64 receiving ARINC-664 port. We can see that these steps are no longer necessary with aEvents.

The following event modes are available:

- Send every ARINC-664 message as it is received from ARINC-664 bus.
- Accumulate received messages in a buffer of up to 12 kB, and send them when the watermark is reached, for bandwidth efficiency.
- Accumulate received messages in a buffer of 12 kB, and send them when after a timeout is reached, for bandwidth efficiency.

See the “SampleAsync664” example for how to use asynchronous events.

Note that, even though multiple multicast port handles within the same VL will be sent with aEvent mode, subscribing to send copies of this same data is neither bandwidth efficient nor necessary - one copy is always sufficient, since it can be replicated in the aEvents processor thread, so subscribe to only port handle.

2.3.4 Stop Cleanly To stop operation, call `DqAdv664Enable()` with a FALSE parameter as follows:

```
// Disable operation

int DEVN = 0;
DqAdv664Enable(hd, DEVN, FALSE);
```



This stops operation of the DNx-ARINC-664 board without changing the configuration.

2.4 UEI ARINC-664 XML Configuration

As described in Section 2.3.2, each DNx-ARINC-664 device is configured using an XML file.

UEI provides a GUI-based ARINC-664 Configurator that allows users to create and edit ARINC-664 configuration XML files. See **Figure 2-5**.

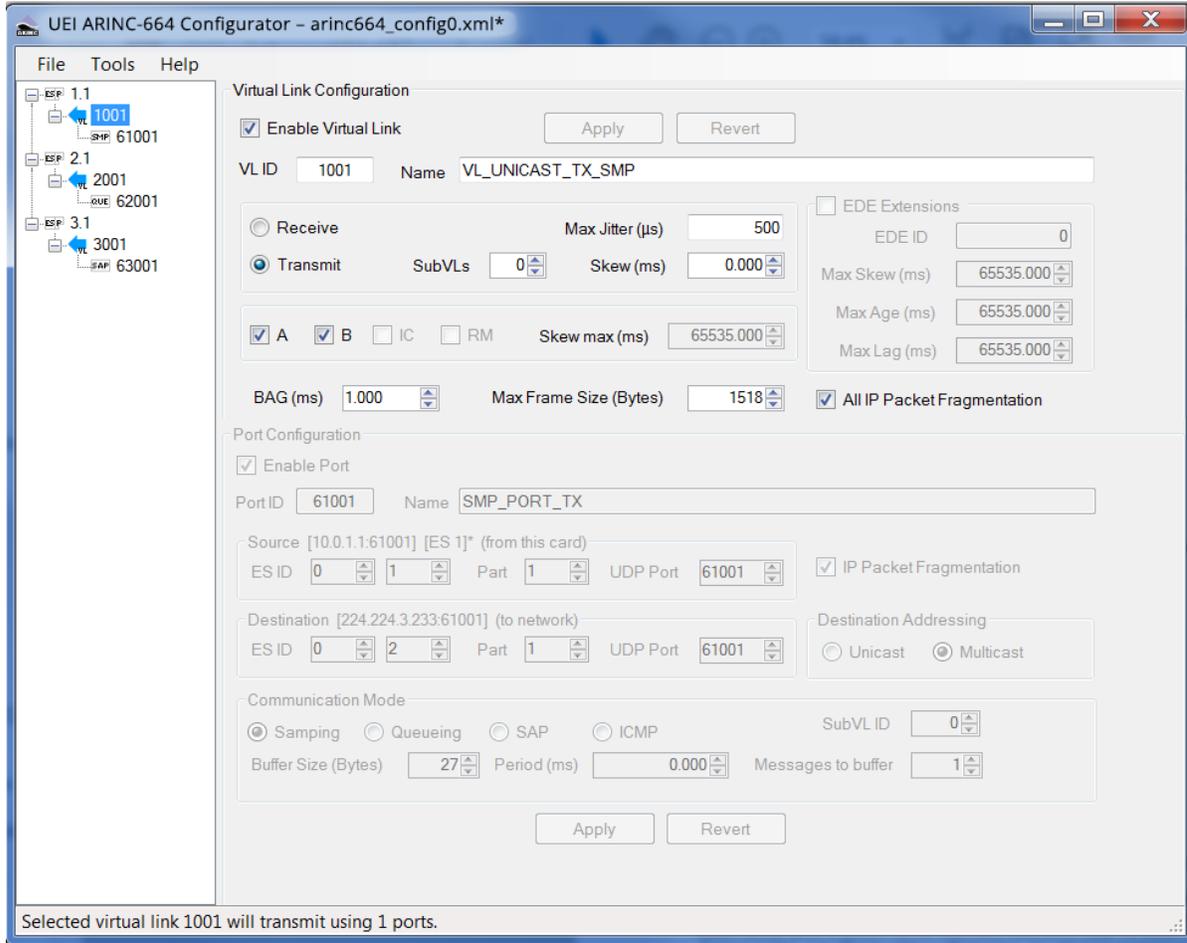


Figure 2-5 ARINC-664 Configurator

A UEI ARINC configuration XML file contains the following tags and attributes:

- An initial XML declaration: `<?xml version="1.0" encoding="utf-8" ?>`
- A root `<configuration>` tag that provides the following attributes:

Table 2-2 Summary of configuration Attributes

Configuration Attribute	Description
format	1 for expanded, 2 for compact
version	unused metadata
name	unused metadata



Table 2-2 Summary of configuration Attributes (Continued)

Configuration Attribute	Description
date	unused metadata
author	unused metadata

- One or more <VL> tags that define the following attributes:

Table 2-3 Summary of VL Attributes

VL Attribute	Description
name	metadata
vlid	16-bit virtual link
enabled	VL enabled: “yes” or “no”
direction	Data transfer direction: “read” or “write” for receive or transmit respectively
bag	BAG as 1/2/4/8/16/32/64/128 milliseconds of bandwidth allocation gap
frag_en	Enable packet fragmentation: “yes” or “no”
n_subvl	Number of subVLs: 2 to 4 possible subVLs; 0/1 to disable subVLs
network_select	Network selection: “A”, “B”, or “AB” for redundant
IC_en	Enable Integrity checking: “yes” or “no”
RM_en	Enable Redundancy Management (when network_select=“AB”): “yes” or “no”
LMax	Largest Ethernet frame: from 64 to 1518 bytes for maximum frame size
skew_max	Maximum time difference in the arrival of over redundant ports: from 0 to 65535 milliseconds for receive ports
max_jitter	Maximum allowed jitter: from 0 to 65535 milliseconds for receive ports
ICMP_VLID	16-bit VL ID of the paired VL to receive or return on
ICMP_en	Identify if there is an ICMP port on this VLID: “yes” or “no”

- For each <VL>, one or more <port> tags that define the following attributes:

Table 2-4 Summary of port Attributes

Port Attribute	Description
name	metadata
portid	metadata



Table 2-4 Summary of port Attributes (Continued)

Port Attribute	Description
enabled	Port enabled: “yes” or “no”
vlid	VLID of parent: 16-bit virtual link of parent
subvl_id	subVL ID: 1/0, 2, 3, or 4
port_type	“Sampling”, “Queuing”, “SAP”, or “ICMP”
period	Period in milliseconds for automatic retransmission in sampling ports as rounded to (period / bag * bag)
d_size	Sampling port payload data size: 1 to 1471 bytes
depth	Queue depth for Queuing/SAP/ICMP messages : 1 to 360
src_port	Source 16-bit port number
dest_port	Destination 16-bit port number
endsys_src	See Note below
endsys_dest	See Note below
part_src	See Note below
part_dest	See Note below
multicast	See Note below

NOTE:

- For the expanded XML format the endsystem and partition address are:
EndSystem address as endsys_src and endsys_dst from 0 to 65535
Partition on EndSystem as part_src and part_dst from 0 to 256
Multicast as “yes” or “no” to use a mutlicast or unicast address
- Compact format endsystem and partition addresses are represented as a multicast address 224.224.[upper 8 bits of vlid].[lower 8 bits of vlid] or unicast address 10.[upper 8 bits of endsys].[lower 8 bits of endsys].[part] for src_ip_address and dst_ip_address.

For more detail, refer to the comments for AR664_VL/PORT_CFG in section 2.3.2.



Chapter 3 Tools and Diagnostics

This chapter provides diagnostic information, procedures, and tools for troubleshooting the DNx-ARINC-664:

- Diagnostic Panel for PowerDNA Explorer (Section 3.1)
- Device RTOS Processes (Section 3.2)
- ARINC-664 Network Packet Inspection (Section 3.3)

3.1 Diagnostic Panel for PowerDNA Explorer

PowerDNA Explorer is a GUI-based diagnostic application. The following section provides information specific to the DNx-ARINC-664. Please refer to the IOM user manual for a detailed description of the tool.

On Windows based systems, PowerDNA Explorer can be accessed as follows:

- *Start » All Programs » UEI » PowerDNA Explorer*

The DNx-ARINC-664 panel in PowerDNA Explorer provides the return results of the low-level API `DqAdv664GetBusStats()` and `DqAdv664GetDeviceInfo()` function calls. Bus usage statistics appear first, followed by the current firmware version, as shown in the following image:

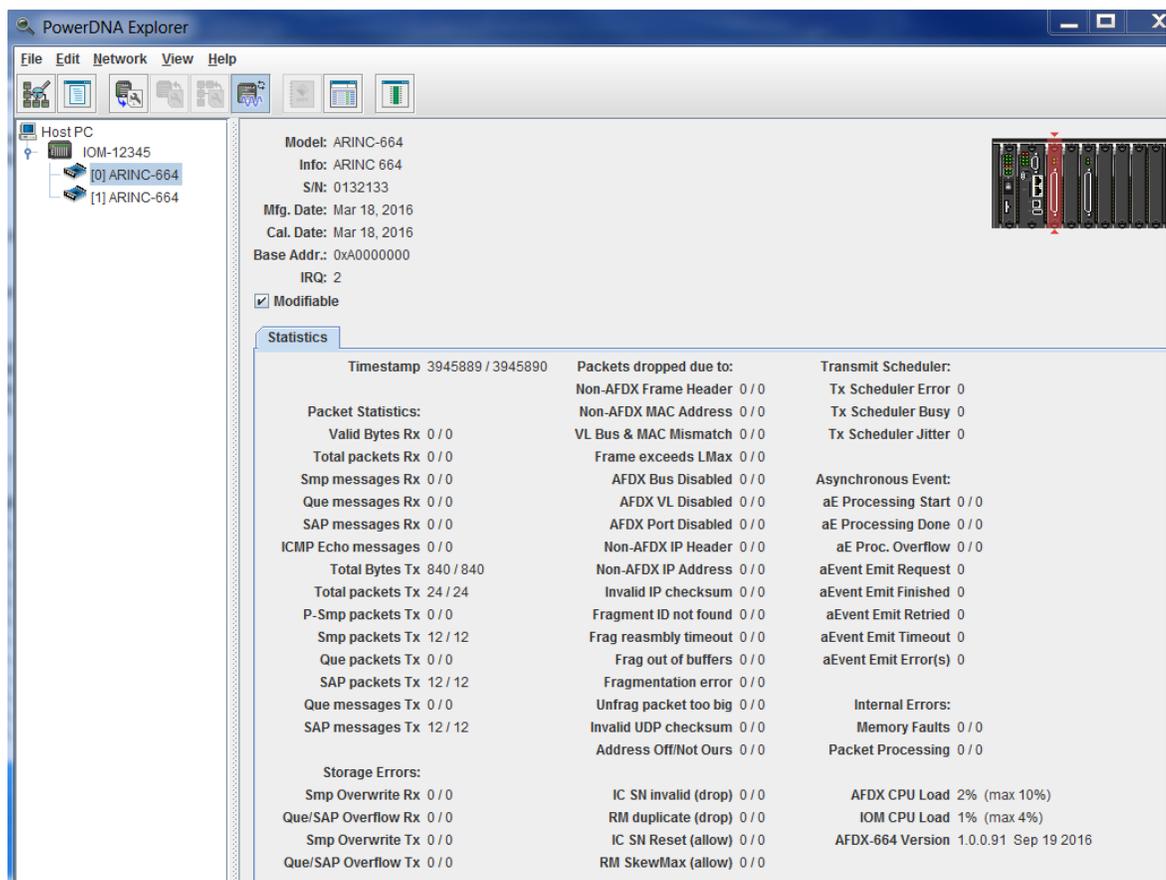


Figure 3-1 DNx-ARINC-664 Panel in PowerDNA Explorer



When **Network > Start Reading Input Data** is active,

`DqAdv664GetBusStats()` will be called a few times per second. This is useful when debugging a few ARINC-664 ports but should not be used with more than 64 active ARINC-664 ports.

PowerDNA Explorer's **View > Hardware Report** provides additional device details:

```
{Layer: Dev0, Model: ARINC-664, S/N: '0123456', Logic: 02.11.66,
Calib.: 'Mar 14, 2015', ST: '00000000', POST: '00000000', FW:
'00000000', LG: '00000000', Version: '1.0.0.88 Mar 14 2015 '}
```

This information can also be obtained using the RS-232 Serial Debug Port without using the PowerDNA Explorer diagnostic application by typing `"devtbl 1"` using the serial debug terminal.

3.2 Device RTOS Processes

This section describes the process by which the receiver stores or drops ARINC-664 messages from the time that they are received from the ARINC-664 network.

3.2.1 Reception

- Frame is received by Three-speed Ethernet Controller (TSEC) IC and subjected to loose hash-based hardware filter (if not promiscuous). Silently discard frame if VLID doesn't match Section 1.6.2.2 VLID formats as configured with `DqAdv664SetConfig`.
- Interrupt Service Routine is called for newly received network frame.
- Pre-filter: Check VLID is one of the Table of VLIDs that was configured for this card with `DqAdv664SetConfig`.
- Begin RX processing:
 - Record Timestamp
 - `pkts_rcv++` (for bus statistics retrieved with `DqAdv664GetBusStat`)
 - VL (Ethernet MAC Address) Sanity Filter:
Malformed Source/Destination Ethernet Address: `link_err++`
Unexpected Bus: `bus_not_match++`
 - IP Address Sanity Filter:
Malformed IPv4 ARINC-664 Header: `ip_hdr_err++`
Malformed IPv4 ARINC-664 Source/Destination Address: `ip_addr_err++`
 - VL-specific Filter:
Bus for VL not enabled: `bus_dis++` (done)
VL not enabled: `vl_dis++` (done)
Frame Larger than LMax: `ip_too_big++` (done)
Protocol not UDP/IP or ICMP echo: `ip_hdr_err++` (done)
UDP port filter mismatch: `port_dis++` (done)
Invalid IP Checksum: `ip_chksum++` (continue)
Invalid UDP Checksum: `udp_chksum++` (continue)



- Integrity Checking (when enabled for A, B, or both):
 Receiver Initialized or Sequence Number is 0: (continue)
 Sequence Number is not PreviousSN+1 or +2: intg_drop++ (done)
- Redundancy Management (when enabled in A&B mode):
 Receiver Initialized: (continue)
 SkewMax exceeded: rdnd_skewmax++ (continue)
 Already Received On Other Bus: rdnd_drop++ (done)
- Packet Reassembly (when fragmented):
 No fragments found or fragments were discarded: ip_frag_err++ (done)
- For all ports in receiving VL:
 - ARINC-664 Port not Enabled: port_dis++ (try next port)
 - Source MAC Address does not match EndSystem ID: (next port)
 - Source or Destination IP Address does not match: (next port)
 - Source or Destination UDP port does not match: (next port)
 - If no packets processed: udp_port_err++ (done)
- Perform Asynchronous Event processing (if enabled),
 or store to port's message buffer.
- Free network buffer.



3.3 ARINC-664 Network Packet Inspection

The following procedure can be used to inspect ARINC-664 network packets with Wireshark 1.10+. Wireshark is a free and open source network instrumentation tool.

Perform a Packet Capture

1. Prepare to capture packets
 - Bring up your capturing Ethernet Adapter's Properties (you can type "ip" in Start Menu).
 - Uncheck all boxes, (e.g., Protocol Version 4 (TCP/IP v4), etc.) to avoid injecting data.
 - Connect the Ethernet CAT5e line into the ARINC-664 network.
2. Start WireShark
3. Start a capture. In the Menu, click **Capture > Options** (Ctrl+K)
 - Select LAN connection in Capture panel.
 - Uncheck "Update list of packets in real time" in right sidebar options.
 - Click **Start**.
 - Notice in the status bar you will see the number of packets captured (e.g., Packets: 314).
4. Stop the capture when done capturing
In the Menu: **Capture > Stop** (Ctrl+E)
 - The packet capture will then begin rendering.



NOTE: For first-time configuration, you can see packets better if you perform these commands:

- View > Coloring Rules > Disable "TTL low or unexpected"
- View > Name Resolution > (Uncheck All)

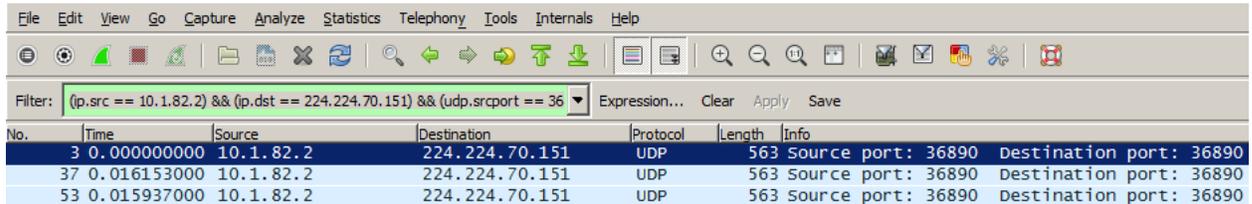


Filter a Captured Packet

To create a display filter to track an ARINC-664 connection, you can type a set of conditions to filter on into the Filter box (see image below).

For example if you have determined that there is interesting data for a particular connection, you can filter by it:

```
(ip.src == 10.1.82.2) && (ip.dst == 224.224.70.151) &&
(udp.srcport == 36890) && (udp.dstport == 36890)
```



This will allow you to scroll through packets to inspect the payload. The payload is what is delivered to the application running on a virtual flight computer (such as the Altitude display in a Display Unit).

To inspect jitter of sampling packets (difference between configured and actual retransmit period), press **Ctrl+Alt+6** or use the menu:

View > Time Display Format > Seconds Since Previous Displayed Packet

To switch back to “seconds since start of capture”, use **Ctrl+Alt+4**.

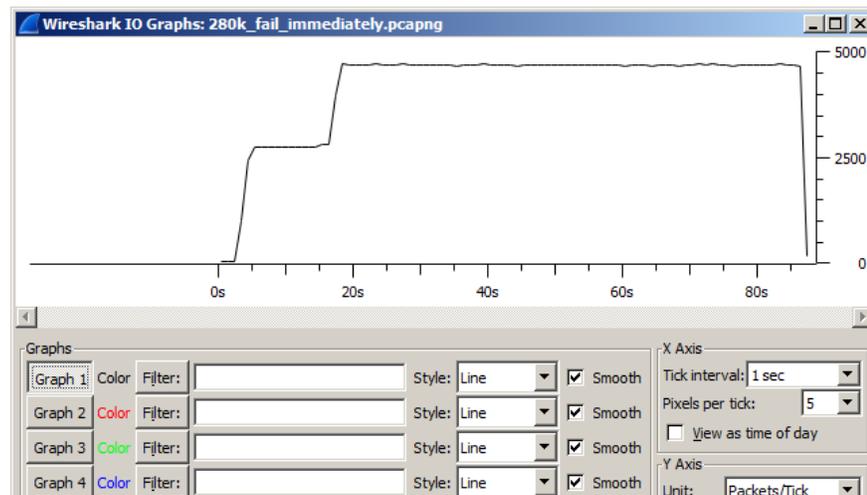
Find Interesting Data

Let’s say that you want to search for a value (e.g., the Altitude) decoded from a message on your 429 bus.

To find a particular hex value, click **Edit > Find Data > Hex Value**

Statistics:

- Statistics > Summary will show averages of traffic
- Statistics > IO Graph will show a graph of traffic across the network



Statistics > Conversations will show when all connections (between various avionics components) start communicating and for how long.



Appendix A

A.1 Accessories The following cables and STP boards are available for the DNx-ARINC-664.

DNA-CAT5E-CBL

This is a 4-conductor round unshielded twisted-pair cable with 8-pin male TIA/EIA-568 connectors on both ends.

DNA-DB9MF-CBL

This is a 9-pin serial cable with male D-sub connectors on both ends. It is used to connect to your PC's serial port or terminal console to the RS-232 port.

DNA-CBL-SYNC-10

Sync-to-sync cable used for synchronization with an external sync port.

DNA-CBL-SYNC-RJ

Sync-to-8P8C (RJ-45) cable for use with a synchronization breakout board.

