United Electronic Industries

The High-Performance Alternative

PowerDNx Protocol and Operating Modes (DAQBIOS)

for

PPCx, PPCx-1G, and PowerDNA Cubes

and

DNR-x-1G HalfRACK and RACKtangle Chassis

October 2010 Edition
PN Man-DAQBIOS Protocol 1010
Version 1.0

Information furnished in this manual is believed to be accurate and reliable. However, no responsibility is assumed for its use, or for any infringements 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 UEI's website for complete terms and conditions of sale:

http://www.ueidaq.com/company/terms.aspx

Contacting United Electronic Industries

Mailing Address:

27 Renmar Avenue

Walpole, MA 02081

U.S.A.

For a list of our distributors and partners in the US and around the world, please see http://www.ueidaq.com/partners/

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	support@ueidaq.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.

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

Table of Contents

List of Figures

Chapter 1 Host / IOM Communication

1.1 Host / IOM Communication Modes

As illustrated in **Figure 1-1**, the PowerDNA API provides four basic ways of communicating between a host and a PowerDNA IOM (cube or RACKtangle):

- DaqBIOS Command API (point-by-point simple I/O, synchronous)
- Buffered I/O in continuous (ACB) or burst (streaming) mode (asynchronous)
- Mapped I/O API (synchronous) — DMap (fixed data size) or VMap (variable data size)
- Messaging — asynchronous, buffered, messaging data format

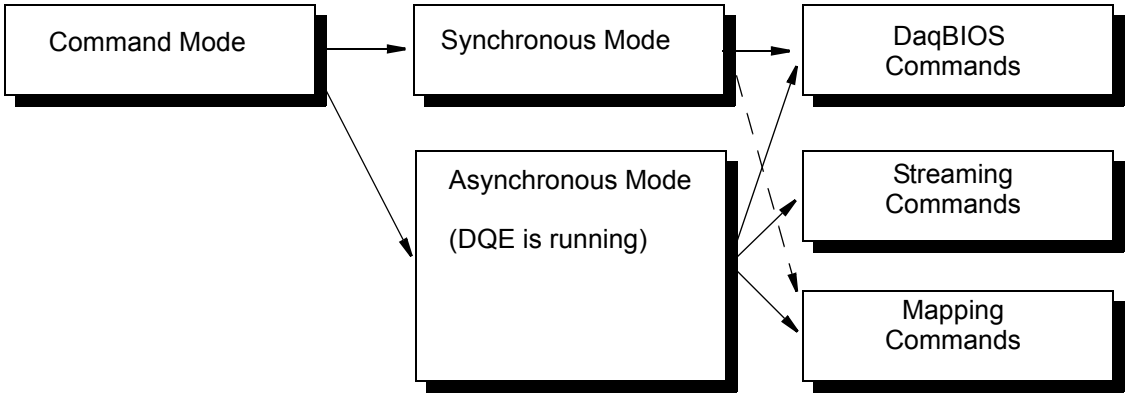


Figure 1-1. Communicating with an IOM

Note that any of the communication modes listed can be selected on a per-I/O board basis and can run independently on the same IOM. Only one API at a time can be used with each I/O board, but each IOM can have multiple I/O boards using the same or different communication modes. If DMap or VMap is selected for more than one board, all such boards are handled as a group.

Some important characteristics of the various modes are:

- In ACB mode, data is transferred in blocks between host and IOM. Each packet contains one block per I/O board configured for ACB operation. If you use multiple ACB I/O boards, data from each board will be sent in separate packets. Each of the boards can run at a different speed.
- In DMap and VMap modes, data is transferred between host and multiple DMap- or VMap-configured I/O boards on an IOM in a single packet. For DMap mode, you are limited to one data value per channel in each packet. For VMap mode, you can set the number of packets per channel in each packet. Also, all such boards must run at the same

speed, controlled by the IOM clock. Transfer of data between IOM and host is controlled by the host. For DMap, update rate of the host maps is usually set at less than half the scan rate of the IOM. For VMap, update rate is usually set to transfer at least half the FIFO of each device.

1.1.1 Synchronous vs. Asynchronous

In synchronous modes, also called single scan or simple IO mode, the host sends a request, waits for a reply, and then sends another command. This keeps the host and IOM in lock step. Error detection/correction is handled by the user.

In asynchronous modes, the host sends requests on ticks of a timebase timer, and the software automatically takes care of re-requests when a network collision or loss of a packet occurs. If you prefer, you can work in a manner similar to synchronous mode, sending request after request and processing packets yourself. However, we recommend that you use asynchronous for streaming or data mapping and design your application accordingly.

Asynchronous mode is inherently soft-real-time because collisions on the network cannot be predicted and, therefore, cannot be avoided.

All three APIs (synchronous, buffered, mapped) can be used to communicate with the same IOM, but not at the same time on one I/O board. Once a device on the IOM is switched to an asynchronous mode, you should not issue synchronous commands to that board so as to avoid interfering with any device configuration or timing set up for asynchronous operation.

1.2 Buffered I/O

Buffered I/O modes use temporary intermediate storage to compensate for varying data transfer rates between host and IOM or devices. The two main asynchronous buffered modes are called Advanced Circular Buffer (ACB) and Burst Mode.

1.2.1 Advanced Circular Buffer (ACB)

As shown in **Figure 1-2**, the Advanced Circular Buffer Mode uses a circular buffer divided into frames. The DaqBIOS engine (DQE) stores data at a known location (the “head”) and reads it at another (the “tail). When a read or write crosses a frame boundary, the DQE triggers an event.

ACB mode also uses another packet ring buffer for temporary and sequential storage of received packets. When the application detects a missing packet, it requests retransmission of the missing packet and uses the packet ring buffer to place the packet in its proper sequence before writing it to the ACB.

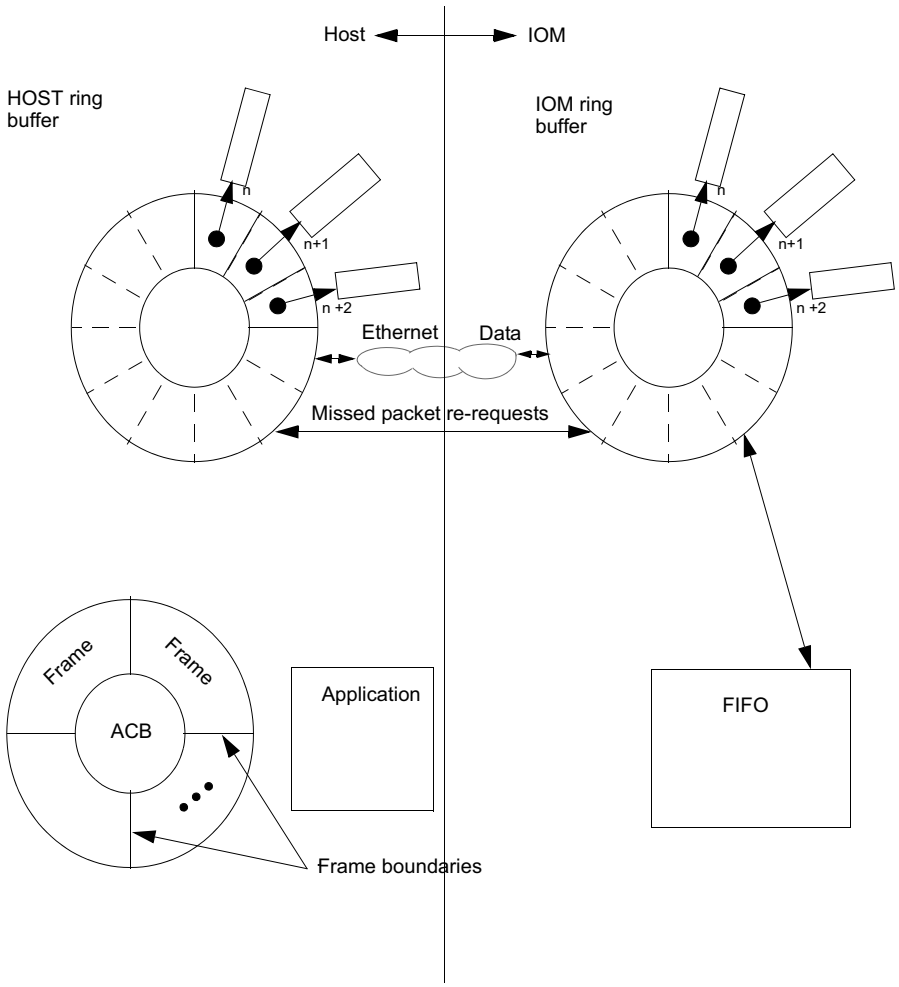


Figure 1-2. Host / IOM Communication in ACB Mode (with DQE)

Once an acquisition is started, DQE stores data into the buffer at a known point (called the head), while the application generally reads data at another position (known as the tail). Both operations occur asynchronously and can run at different rates. However, you can synchronize them either by timer notification or by triggering a DQE event.

To issue a notification to the user application upon receipt of a specific sample or when incoming data reaches a scan-count boundary, DQE segments the buffer into frames. Whenever incoming (or outgoing) data crosses a frame boundary, DQE sends an event to the application. If multi-channel acquisition is performed, the frame size should be a multiple of the scan size to keep pointer arithmetic from becoming unnecessarily complex.

With the ACB, three modes of operation are possible, which differ in the actions taken when the end of the buffer is reached or when the buffer head catches up with the tail.

• In **Single Buffer** mode, acquisition stops when DQE reaches the end of the buffer. The user application can access the buffer and process data during

acquisition or wait until the buffer is full. This approach is appropriate when you are not acquiring data in a continuous stream.

• In **Circular Buffer** mode, the head and tail each wrap to the buffer start when they reach the end. If the head catches up to the tail pointer, the buffer is considered full and acquisition stops. This mode is useful in applications that must acquire data with no loss of sample data. Data acquisition continues until either a predefined trigger condition occurs or the application stops DQE. If the application can't keep up with the acquisition process and the buffer overflows, the driver halts acquisition and reports an error condition.

• **Recycled** mode resembles Circular Buffer mode except that when the head catches up with the tail pointer, it doesn't stop but instead overwrites the oldest scans with the new incoming scans. As the buffer fills up, DQE is free to recycle frames, automatically incrementing the buffer tail. This buffer-space recycling occurs whether or not the application reads the data. In this mode, a buffer overflow never occurs. It is best suited for applications that monitor acquired signals at periodic intervals. The task might require that the system digitize signals at a high rate, but not process every sample. Also, an application might need only the latest block of samples.

When the buffer is used for output, the user should fill at least two frames before starting output. Every time a frame becomes empty and ready to accept new data, the DQE triggers an event to the application.

While the ACB may seem a departure from the single and double-buffer schemes you see in most other data acquisition systems, it's actually a superset of them. In Single Buffer mode, the ACB behaves like a single buffer. If configured as a Circular Buffer with two frames, it behaves as a double buffer. With multiple frames, the ACB can function in algorithms designed for buffer queues. The only limitation, which results in more efficient performance, is that the logical buffers in the queues cannot be dynamically allocated or freed and that their order is fixed.

The Ethernet UDP protocol used to transfer data is connectionless and unreliable. Older packets may come first and new packets may never arrive. The ACB assumes that the data comes sequentially without gaps between scans. To accommodate the sequential nature of a data stream with the packet nature of Ethernet, DQE implements an additional intermediate buffer – called the Packet Ring Buffer (PRB), which should not be confused with the separate ACB buffer.

The PRB is a non-contiguous ring buffer intended for data loss recovery. FIFO devices on the IOM send their data to the host in sequentially numbered packets (using the dqCounter field of the DaqBIOS command header). These numbers vary from 0x1 to 0xFFFF and then wrap around (skipping 0). Such numbering allows DQE to notice when a packet is missing — detected whenever a higher-numbered than expected packet is received. (In **Figure 1-2**, if the last packet number was n and we've just received one numbered n+2, we know that the packet n+1 is missing.) Since the receiving buffer is non-contiguous, we just put the newly arrived packet into the buffer, which was bound to receive it anyway, and send a specific request for the missing one. When it finally arrives, we just put it in its proper place and copy all data into the contiguous ACB in

correct order.

A thread transfers data from the ring buffer into the ACB when contiguous chunks of data become available. The data request routine, (`DqGetACB-Scans()`, also performs additional transfers if a chunk of contiguous data is available at the moment of execution.

1.2.2 Burst Mode

Burst Mode is a streaming mode in which data is sent or received continuously for a specific time duration or until an event such as timer event, buffer full, or buffer empty occurs.

1.3 Message Mode (Msg Protocol)

With messaging devices (serial, CAN, ARINC interfaces), the data is a stream of bytes logically divided into frames, messages, strings, etc. Two characteristics of messaging devices make DMap protocol inefficient, if not impossible, for handling messages and thus generate a need for another protocol specifically designed for messaging. These characteristics are:

1. Since the data is a stream, losing part of the data may change the meaning of the message.
2. Unlike digital or analog data, the timing of data availability depends on the external stream of messages — you cannot predict when and how much data will become available, and whether or not receiving/transmitting errors may exist on the bus.

Messaging layers, therefore, are supported by the **Msg Protocol**, which shares the same buffering mechanism as the ACB protocol. The Msg protocol buffer receives packets and delays releasing newer packets to the user application until it re-requests and receives all the packets in the message stream. Although this protocol does provide a gapless stream of messages, it is not suited for realtime operation because some deadlines may be missed.

Message mode operates in much the same way as ACB mode. The IOM, of course, must have a layer that supports a messaging protocol, such as a CAN-503 layer. When messages are received by the messaging layer, they are stored in the FIFO. As with the streaming version of ACB mode, a messaging layer in Operation mode sends packets (containing the received messages) to the host automatically, without the host having to send a command to request them. When the host receives the message packets, it puts them into a Receiving Message Queue, which is similar to an ACB, and signals an event, which alerts the client program. The client program can then retrieve the messages and process them as needed.

There is also a Sending Message Queue on the host side, into which the client program can insert outgoing messages. These messages are taken from the queue by the reader thread and sent to the IOM. The IOM then transmits the message on the network interface of the layer.

1.3.1 IOM/Host Data Transfer

When the messaging layer receives a message, the message is stored in the FIFO of the layer. When running in Operation mode, the layer checks the FIFO at regular intervals and transmits any as yet unsent messages to the host.

A DQFIFO structure in a packet sent from the IOM to the host may contain one

or more messages in its data field. The data field consists of a 16-bit value indicating the size of the next message block, followed by the message block itself, followed by another size value and message block, etc. A size field of 0 terminates the sequence. See the **Figure 1-3** for an illustration.

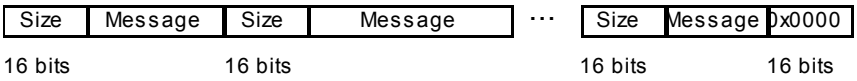


Figure 1-3 Data Field of a RDFIFO Packet Containing Messages

This same format is used to transfer outgoing messages from the host to the IOM for transmission on the network. The host sends a WRFIFO command whose data field holds one or more messages stored the same way.

The format of each message block is specific to the layer type, as described below.

1.3.2 CAN-503 Data Transfer

There are two relevant pieces of information contained in a CAN network protocol packet: the identifier and the message data itself. The identifier is either 11 or 29 bits long, depending on whether it is a standard packet or an extended packet. The data can be 0 to 8 bytes long. In addition, the CAN-503 layer has four network interfaces, which are analogous to channels on other layers. The message data coming from the FIFO of the layer thus has to include the following three pieces of data: the ID, the message data, and the channel that received it. Messages sent to the IOM from the host must also include this information. The message block for a CAN-503 layer is illustrated in **Figure 1-4**.

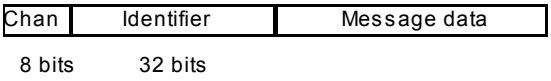


Figure 1-4. Message Block for CAN messages in FIFO

The first byte indicates the channel (network interface), the next four bytes contain the identifier, and the remaining bytes contain the message data. Recall that the size of this block is stored in the 16 bits immediately preceding it, where it appears in an RDFIFO response packet.

1.3.3 PDNALib Structures

The PDNALib requires several data structures to implement Message mode, as described below.

1.3.3.1 Message Struct

The DqMessage struct holds a message. It contains the channel number, the address of the intended recipient, the address of the sender, and the message contents. Like other structures defined in the PDNALib, the message content field is declared last, as a byte array of unspecified size, so that an instance of the struct can be allocated via malloc to any size based on the desired message data size. The address field size is 16 bytes, which is the address size of IPv6. This allows PDNA to support any existing messaging protocol.

1.3.3.2 Message Queue

In PDNALib, the BCB structure can be any of three types: ACB, DMAP, or a third variant called Message Queue (or MSGQ). For each messaging layer installed, the user should create two message queues: one to hold messages to

be sent, and one to hold received messages. A Message Queue is similar to an ACB, except that instead of being implemented as a flat byte array, it is a linked list of pointers to DqMessage structures. The sending callback function for Message mode takes DqMessage structures from the sending message queue and converts them to DQ commands, which are then sent to the proper layer on the IOM. As messages are received by the receiving thread from the IOM, the receiving callback function converts the messages to DqMessage structures, and then stores them in the receiving message queue. In this case, an event is triggered with the `DQ_eDataAvailable` flag set. When the client program gets the event, it can call `DqMsgRecvMessage` to get the message and remove it from the receiving message queue.

Two message queue BCBs must be created for interacting with a message layer: one for sending messages, and one for receiving them. One of two constants must be passed to DqMsgInitOps to indicate which direction the BCB is being initialized for.

The DQBCB structure is now able to contain a message queue instead of an ACB or DMAP.

For more detailed information, refer to the PowerDNA Reference Manual API.

1.3.4 Error Recovery

If network problems prevent an occasional message packet from successfully being sent to the IOM or host, PDNALib will attempt to recover by retransmitting or re-requesting the lost packet.

1.3.5 Other Messaging Types

Other types of messages such as SL-501/508, ARINC-429, and MIL-1553 are handled in a manner similar to that of the CAN-503.

1.4 Mapped I/O

The basic benefit from using I/O mapping is increased speed and throughput. By maintaining duplicate maps of I/O data in both host and the IOM, both processors can access their own data map(s) as needed, without having to communicate across the network. Communication between host and IOM only has to keep the two maps up to date with each other. UEI offers two types of data mapped I/O: Direct Data Mapping (fixed data size), called DMap, and Variable Data Mapping (variable data size) called VMap, which are described below.

1.4.1 Fixed-Size Data Mapping (DMap)

Fixed-size data mapping allocates defined-size (maximum of one packet) areas of input and output data that are continually maintained as mirrors of each other. The following diagram illustrates the structure of DMap operation.

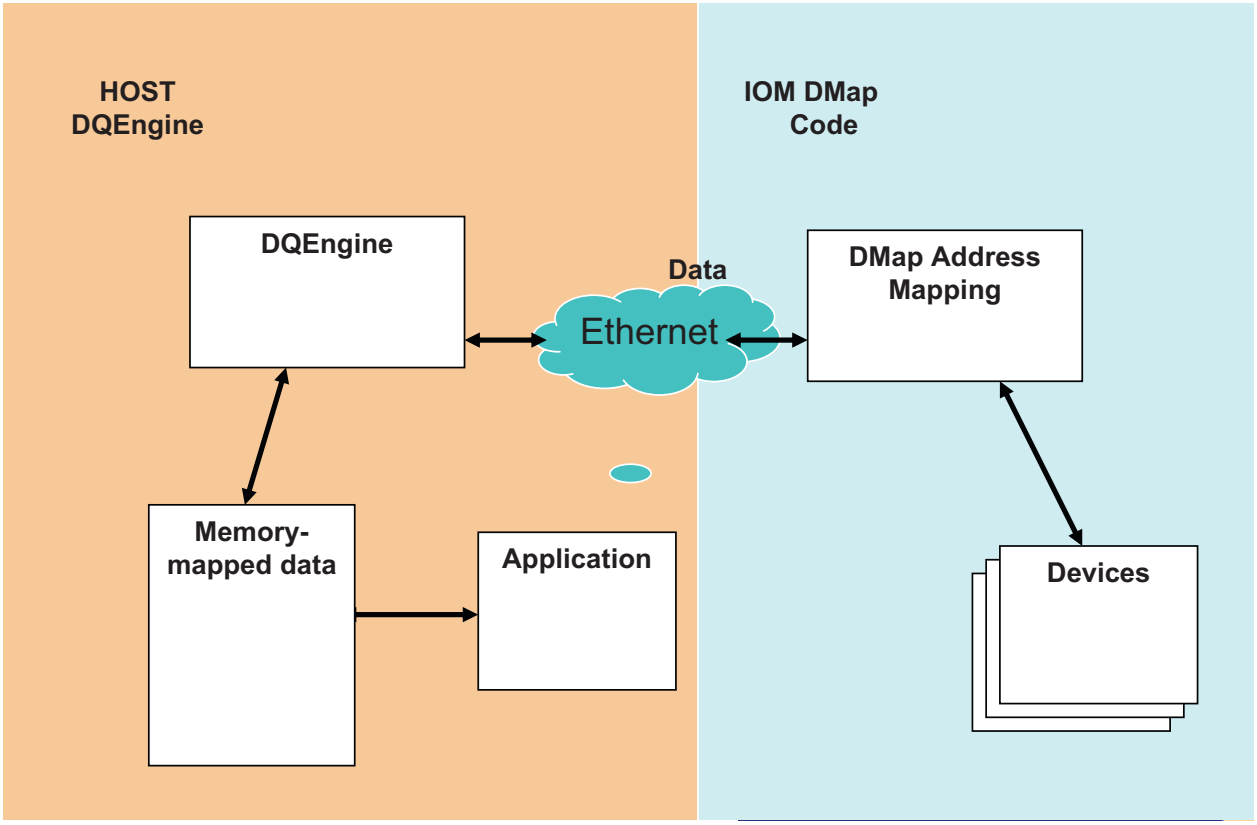


Figure 1-5. Host / IOM Communication in DMap Mode

Each DMap is associated with a device (layer) or group of similar devices in an IOM and each has its own input and output maps. A DMap can store data either in raw or engineering units (volts by default) and can be a maximum of 510 bytes in size (equal to one packet of data). Maps are therefore fully updated as each packet is received from the network.

As indicated in the diagram, input data is acquired by the IOM layers under control of the IOM clock, stored in IOM memory in an area called DMap, and then transferred over the network to the host. In the host, it is stored in the host DMap where it can be accessed by the host application software as needed. Output data is transferred in a similar manner from host to IOM. Packets are transmitted in both directions at a rate determined by the host. The rate is set

fast enough to provide a fresh input reading with every reply packet and is typically set at a rate less than half the IOM scan rate (Nyquist rule). The output runs at a rate capable of updating outputs before the next portion of data arrives.

The major attributes of DMap Mode are:

- Each fixed size data map holds a snapshot of simultaneous data for all layers in an IOM that are configured for DMap mode.
- A DMap packet delivers output data from host to IOM: IOM returns most recent input data as a reply.
- Reply is guaranteed within 250 us (133 us with gig-E networks)
- All DMap-configured layers in an IOM are inherently synchronized
- Data is synchronized across multiple IOMs, based on host requests
- All packets are sequentially numbered; the application can re-request lost packets.

1.4.2 Variable-size Data Mapping (VMap)

VMap is another type of mapped I/O that offers variable size data maps. VMap, therefore, is useful for installations in which the size of data to be transferred is unpredictable, such as in messaging or data streaming applications, or when communication bandwidth utilization can be improved by varying the packet size.

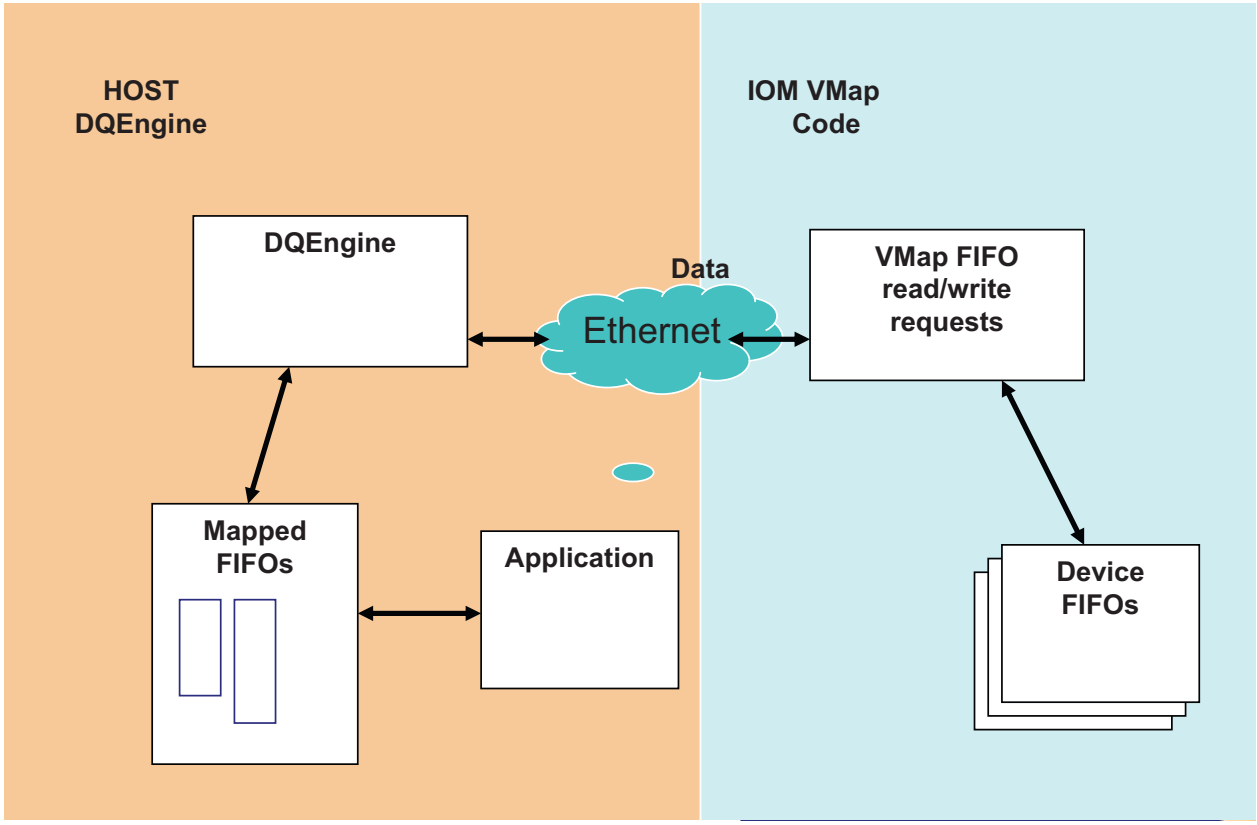


Figure 1-6. Host / IOM Communication in VMap Mode (with DQE)

At high level, VMap is very similar to DMap. A user must create VMap with output and input buffers and add channels/layers of interest to it. As with DMap, DQEngine supports multiple VMaps that can operate at different rates derived from the main DQEngine update period. Unlike DMap, however, VMap packets have additional fields.

First of all, there is a flag field, which is used to guarantee continuity of messaging data. Second, an output buffer adds a pair of fields for each channel in the map at its header. The first field provides the IOM with information on how much data is to be transmitted for that channel and the second field defines the maximum size of data to be received from that channel. Offsets of the output data in the buffer should match the size of the data in the buffer header.

An input packet also contains a flag field as well as the number of bytes actually written, actually received, and (optionally) the number of bytes available in the receive FIFO and the room available in the transmit FIFO. This feature allows flexibility in allocating packet slices for different channels. Each time packets are exchanged between host and IOM, the user application can select different sizes for outgoing and incoming data, taking into consideration the amount of

data required to be sent and the size of data accumulated in the receiving FIFO. If you don't use a channel at this time, you should set the "size to send" and the "size to receive" to zero. The header has a fixed width set up before starting VMap operation. The user cannot change the header size on the fly even if the channel is no longer in use.

VMap also has a function that returns the VMap ID to the user for use in multiple IOM installations. Since packets from multiple IOMs may be received by the host out of time sequence, this function gives the host the information necessary to call the right VMap processing routine for that packet.

The packet counter (dqCounter in the DQPKT header) and the flags field work hand-in-hand to synchronize the user application with the DQ Engine.

Table 1-1 lists functions specific to VMap mechanisms. For more detailed information, refer to the PowerDNA API Reference Manual

1.5 Choosing the Right Layers, Operating System, and Mode

Choosing the right communication mode for your data collection system can significantly improve performance of your system and help meet your design goals. The PowerDNA system offers several choices to meet the needs of your particular application. One of them is sure to meet your particular requirements. Note that you can select different modes for each layer (but only one mode per layer) and that all DMap- or VMap-configured layers are handled as multi-layer groups.

1.5.1 Attributes of Modes

The various modes and their attributes are described below.

Simple I/O Mode (Single Scan)

The major attribute of point-by-point mode is its simplicity and straightforward operation. Requests are sent back and forth between host and IOM in sequence. Error detection/correction is handled by the host.

This mode is supported in Windows XP, Linux, Real-time, UEIPAC, and QNX operating systems.

ACB Mode

Each subsystem of each layer is handled as a separate stream of data. Every data point is guaranteed delivery, which inherently synchronizes data from different layers. ACB data can be synchronized with other data by using time-stamps and/or the SyncX interface between Cubes. In ACB mode, devices can be clocked from external sources. This mode is currently supported in Windows XP and QNX operating systems.

Burst Mode (ACB sub-mode)

Stream-to-memory improves performance by storing data into RAM first and then transferring data on a stop trigger. Stream to memory makes 64MB RAM available for temporary storage (equal to 4 seconds of data from four AI-205 layers). Cannot work continuously (limited by size of memory).Can stream data on change of state of digital input. This mode is currently supported in Windows XP and QNX operating systems.

Messaging Mode

Uses an ACB to transfer messages.Messages can be grouped together in a single packet to improve performance. Messages can be sent upon receiving a specified amount of data or upon timeout. This mode is currently supported in Windows XP, UEIPAC, and QNX operating systems.

DMap Mode --Fixed size of data map(s)

Can be used with both soft and hard real-time systems. With a non-RT OS, the DQE handles lost packet detection/correction and guarantees message continuity. When the real-time set of PowerDNA functions is used on soft or hard real-time OSs, the application software must handle error conditions.

Single DMap for all DMap-configured layers in Cube or Rack.

A single DMap packet delivers output data for all DMap layers; IOM returns most recent input data as a reply. Reply is guaranteed within 250us (133 us for Gig-E systems). Data is synchronized across multiple IOMs based on host requests, guaranteeing that data is synchronized within DMap timebase. Notifies application if packet is lost and no recovery is available via DQE. All packets are numbered sequentially; custom application can re-request a lost packet. This mode is currently supported in Windows XP, Linux, Real-time, UEIPAC. and QNX operating systems.

VMap Mode -- Variable size of data map(s)

Single VMap for all VMap-configured layers in IOM (IOM can be partitioned into multiple VMaps as needed). VMap packet delivers output data and returns input data plus number of samples and number now available.VMap packet can be resized dynamically to optimize bandwidth use. VMap can be used for AIn/DIn streaming when soon-to-be-released support is available. VMap has built-in mechanism to inform about lost packets. All VMap packets are sequentially numbered to maintain message integrity. If packet is lost between host and IOM, IOM will re-output (input) the packet. If lost between IOM and host, IOM will resend the packet. This mode is currently supported in Windows XP, Linux, Real-time, UEIPAC. and QNX operating systems.

1.5.2 Application Requirements

Each application has a particular set of requirements, several of which may be opposing. For example, a pure **data acquisition application** may require that all data be delivered without any gaps, but may accept slight delays in delivery of the data. A **control system application**, however, usually requires that all data be delivered on time, but that a few missing data points can be tolerated. In such applications, meeting time deadlines is more important than having gapless data. In **complex applications**, the usual requirement is that all data be delivered in as short a time as possible.

The typical tolerances for delay in data delivery for various types of data collection systems are:

- Data Acquisition System — 1 to 2 seconds
- Control System — 0.5-10 ms

- Complex Application — 10 ms

The tolerances listed above and the bandwidth of the signals being measured affect the choice of communication mode, scan rate, and type of operating system. Some important attributes of desktop and real-time operating systems that influence these choices are:

Desktop OS

- Windows XP: 10 ms soft real-time
- Linux 2.6: 10 ms soft real-time (1 ms with pre-emptive patch)
- All communication modes are supported
- Real-time OS
- Real-time Linux, Windows RTX: 250us hard real-time control loops
- DMap, VMap, and Single Scan are supported
- ACB, Msg, and M3 modes are not supported

Mode / OS Support

In making the choice of communication mode for your application, you need to verify that a particular mode is supported by your selected layers, operating system, and also by the OS environment you are working with. **Table 1-1** lists the current state of support offered by UEI for various operating systems. **Table 1-1** shows current UEI support for type of OS environment. **Table 1-3** shows current UEI support for types of Analog Input Layers. **Table 1-5** shows current UEI support for types of DIO and Analog Output Layers. **Table 1-6** shows current UEI support for types of Messaging Layers.

Table 1-1. Mode Support by Operating System

Operating System	Single Scan	ACB	DMap	VMap	Messaging
Windows XP/Linux	Y	Y	Y	Y	Y
Real-time	Y	N	Y	Y	N
UEIPAC	Y	N	Y	Y	N
QNX	Y	Y	Y	Y	Y

.

Table 1-2. Mode Support by Operating System Environment

OS Environment	Single Scan	ACB	DMap	VMap	Messaging
PDNALib	Y	Y	Y	Y	Y
Framework	Y	Y	Y	N	Y
3rd party drivers	Y	Y	Y	N	Y
PDNALib running under a Real-time OS	Y	N	Y	Y	N

.

Table 1-3. Mode Support by Analog Input Layer

Layer Model No.	Single Scan	ACB	DMap	VMap	Messaging
AI-201	Y	Y	Y	N	--
AI-207/208	Y	Y	Y	N	--
AI-205	Y	Y	Y	N	--
AI-211	Y	Y	Y	N	--
AI-217	Y	Y	Y	N	--
AI-224	Y	Y	Y	N	--
AI-225	Y	Y	Y	N	--

Table 1-4. Mode Support by Analog Output Layer

Layer Model No.	Single Scan	ACB	DMap	VMap	Messaging
AO-308	Y	Y	Y	N	--
AO-332	Y	Y	Y	N	--
AO-333	Y	Y	Y	N	--

Table 1-5. Mode Support by Digital I/O Layer

Layer Model No.	Single Scan	ACB	DMap	VMap	Messaging
DIO-40X	Y	Y	Y	N	--
DIO-416	Y	N	Y	N	--
DIO-432/433	Y	N	Y	N	--
DIO-448	Y	N	Y	N	--

Table 1-6. Mode Support by Messaging Layer

Layer Model No.	Single Scan	ACB	DMap	VMap	Messaging
SI-501 / 508	Y	N	N	Y	Y
CAN-503	Y	N	N	Y	Y
429-566	Y	N	N	Y	N
CT-601 / 604	Y	N	Y	N	Y

1.5.3 Selecting the Right Mode for Your Application

To select the communication mode best suited to meet your needs, consider the following selection criteria for each mode:

ACB application requirements (typical)

- Acquire and store/display data
- Continuous data > 100 Hz, gapless
- Data stream faster than 10 kB/s
- Timing accuracy better than 1/(data rate) seconds
- Delay between acquisition and delivery is non-critical (0.1s – 1s)
- IOM controls timing
- External trigger/clock is required

Messaging application requirements (typical)

- Send, receive, store a stream of messages
- Guaranteed message delivery
- Maximum communication bus loads (serial, CAN, ARINC)
- Non-critical delay of delivery (within 0.1s-1s)
- Receive data based on number of bytes, messages, content, timeout

DMap application requirements (typical)

- Control and simulation applications
- Host controls timing of data transfers, minimizes response time
- No network collisions allowed
- Permits scan rate of 100-500 Hz on non-realtime, 4 kHz on realtime OS
- Multiple IOM configuration OK
- VMap application requirements (typical)
- Control and simulation applications
- Variable length messages or Real time data size larger than one scan
- Host controls timing of data transfers
- Maximizes IOM performance and bandwidth, minimizes response time
- Advanced features: message scheduler, frame delays and repetitions

1.5.3.1 Selection Procedure

The general procedure for selecting the communication mode for your system is as follows:

STEP 1: First, define the primary goals of your data collection system.

- Is it a control or data acquisition application?
- What are the signal types, levels, and bandwidths?
- Which is more important – gapless data or timely response?

- Can the application run on a real-time OS?

STEP 2: Choose the I/O layers for your system and select operating parameters for each.

- Signal Type – In/Out, Analog Voltage/Current, Digital Logic Level, Frequency, PWM, Strain, Message (SL, CAN, ARINC, 1553)
- Signal Level
- Bandwidth
- Timing Control (simultaneous or not, Int/ext sync, etc.)

STEP 3: Determine timing requirements and tolerance for gaps in data.

- Data Acquisition — a 1 to 2 second delay is usually acceptable
- Control — 1 0.5 to 10 ms control loop period is typical
- Complex Application — 10 ms control loop data delay is acceptable
- Real-time or Non-Real-Time — can missed deadlines be tolerated?

STEP 4: Select applicable Operating System

- Desktop OS
- Windows XP: 10 ms soft real-time
- Linux 2.6: 10 ms soft real-time (1 ms with pre-emptive patch)
- All communication modes are supported
- Real-time OS
- Real-time Linux, Windows RTX: 250us hard real-time control loops
- DMap, VMap, and Single Scan are supported
- ACB, Msg modes not supported

STEP 5: Verify availability of UEI support for your choices (layers, parameters, data processing, OSs, OS environments, RT/nonRT, messages vs. non-message communication modes). Modify choices as needed.

- See Table 1-2 to Table 1-6 starting on page 14

STEP 6: Based on factors listed above, choose Host/IOM communication mode and select optimum parameters.

Chapter 2 How DaqBIOS Protocol Works

2.1 DaqBIOS Packet Structure

The DaqBIOS (DQ) protocol relies on the Ethernet protocol for transfer. Current implementation of the IOM firmware allows exchanging DaqBIOS packets over raw Ethernet packets and over UDP packets, but library implementation under Microsoft Windows [tm] does not have an option of using raw Ethernet packets.

Ethernet header (14 bytes)	IP header (20 bytes)	UDP header (8 bytes)	DQ header (8 bytes)	DQ data (6-514)	Ethernet CRC (4 bytes)

Figure 2-1. DaqBIOS Packet Over UDP Packet

Ethernet header (14 bytes)	DQ header (16 bytes)	DQ data (34-542)	Ethernet CRC (4 bytes)

Figure 2-2. DaqBIOS Packet Over Raw Ethernet Packet

The DaqBIOS protocol relies on the simple concept of acknowledging every packet sent from the host to the IOM.

The DaqBIOS packet header has following fields:

```
typedef struct {
    uint32 dqProlog;      /* const 0xBABAFACA */
    uint16 dqTStamp;      /* 16-bit timestamp */
    uint16 dqCounter;     /* Retry counter + bitfields */
    uint32 dqCommand;     /* DaqBIOS command */
    uint32 rqId;          /* Request ID - sent from host, mirrored */
    uint8  dqData[];      /* Data – 0 to 514 bytes */
} DQPKT, * pDQPKT;
```

dqProlog is always 0xBABAFACA for revision 2 of the DQ-TS protocol. The DQ-VT protocol available earlier is no longer supported in R2. Instead, we use flow-control and error-correction protocols. The only exception is when you can send a packet with 0xBABAFAC2 as a prolog. In this case, the IOM replies with a proper Prolog and protocol version supported in dqTStamp.

dqTStamp is a field used for time synchronization between the IOM and the host.

dqCounter is used for flow-control between the host and the IOM. The counter starts from one and continues up to 65535, then wraps around.

dqCommand is used to specify the command to be executed when sent from the host to the IOM. The host replies with the command executed and with any error flags set. If the IOM processes the command successfully, it replies with the requested command and the DQREPLY (0x1000) flag. If the host sends a command with a DQNOREPLY (0x2000) flag, the IOM does not send a reply packet.

The following errors located in the upper 16 bits of dqCommand are sent in dqCommand field:

```
/* Masks to extract DQERR_... from command code */
#define DQERR_MASK      0xFFFF0000
#define DQNOERR_MASK    0x0000FFFF

/* The first nybble indicates how the next three nybbles should be interpreted
*/
#define DQERR_NYBMASK   0xF0000000 /* general error/status mask */
#define DQERR_MULTFAIL  0x80000000 /* high bit - multiple bits indicate error/
status */
#define DQERR_SINGFAIL  0x90000000 /* low bit in first nybble - single error/
status */

#define DQERR_BITS     0x0FFF0000 /* error/status bits or value extracted from
here */

/* multiple errors - inclusive or-ed with dqCommand -- high bit set */
#define DQERR_GENFAIL   0xF0000000 /* general error/status mask */
#define DQERR_OVRFLW    0x80010000 /* Data extraction too slow - data overflow
*/
#define DQERR_STARTED   0x80020000 /* Start trigger is received */
#define DQERR_STOPPED   0x80020000 /* Stop trigger is received */

/* single errors/status - not inclusive or-ed bit 0x10000000 set */
#define DQERR_EXEC      0x90010000 /* exception on command execution */
#define DQERR_NOMORE    0x90020000 /* no more data is available */
#define DQERR_MOREDATA  0x90030000 /* more data is available */
#define DQERR_TOOOLD    0x90040000 /* request is too old (RDFIFO) */
#define DQERR_INVREQ    0x90050000 /* Invalid request number (RDFIFO) */
#define DQERR_NIMP      0x90060000 /* DQ not implemented or unknown command */

/*
** The following is reuse of the previous code
** in a different direction: host->IOM
** It means that there was no reply to one
** of the previous packets of the same type
** Made especially for RDALL, WRRD and RDFIFO
** commands.
*/
#define DQERR_OPS       0x90070000 /* IOM is in operation state */
#define DQERR_PARAM     0x90080000 /* Device cannot complete request
                                    /* with specified parameters */

/* network errors */
#define DQERR_RCV       0x90090000 /* packet receive error */
#define DQERR_SND       0x900A0000 /* packet send error */
```

rqId – request ID. Every time the host sends a packet to IOM, it is accompanied with a new request ID. The Request ID serves to specify what request the reply belongs to when request/reply pairs are overlapped. RqId is used under the control of DQE only.

In synchronous operating mode, commands are sent and replies are received. The following picture depicts how the host and the IOM exchange packets under the DaqBIOS protocol:

2.2 DaqBIOS Protocol Versions

To recognize what version of the DaqBIOS protocol the PowerDNA cube supports, the host should send a command with dqProlog set to 0xBABAFAC2. The IOM will reply with the proper prolog and the DaqBIOS protocol version in the dqTStamp field and the firmware version in the next four bytes. This subprotocol allows the host to recognize what version of the firmware is running on the PowerDNA cube and what version of protocol it supports.

2.3 Host and IOM Data Representation

Data on the IOM as well as in the network packets are represented in big-endian format. Data on the PC platform are rendered in little-endian format. Thus, to ensure proper data representation, the user should convert data from network format to host format and back.

2.3.1 Soft and Hard Real-time

We address real-time performance as soft-real-time when timing deadlines are achieved almost every time. However, soft-real-time cannot guarantee meeting a deadline in all instances. The majority of general-purpose OSs (Microsoft Windows, Linux, etc.) are soft-real-time with better or worse probability of missing a deadline.

Hard-real-time performance guarantees that no one deadline is missed. Hard-real-time OSs have specially designed schedulers that preempt any ongoing operation when real-time code has to be executed. QNX and RTLinux are examples of hard-real-time OSs.

2.3.1.1 Implementation Details

Hard real-time response is achievable only under control of hard-real-time OSs (QNX, for example) or general-purpose OSs with real-time extensions (RTLinux, RTAI Linux.) Real-time OSs are capable of sending DaqBIOS commands to the host without missing deadlines (using DQE). This avoids network collisions completely. Two sets of commands are available for real-time operations: DaqBIOS commands and data mapping commands. Streaming cannot be made real-time because its timing cannot be controlled from the host side.

If streaming is required under a real-time system, you can implement streaming in FIFO mode rather than streaming mode. FIFO mode assumes that the host sends a request to retrieve data from the IOM side every now and then. This way, the real-time application is responsible for retrieving data on time.

2.3.1.2 Immediate and Pending Commands

The firmware processes some commands immediately in the network interrupt vector. Other commands are scheduled and executed by firmware in the pending command thread. A vast majority of DaqBIOS commands are immediate commands. See the PowerDNA API Reference Manual for details. Firmware running on a CM-1 layer sends replies within 200-400µs. Commands that include waiting for some hardware events to happen are implemented as pending commands. They include IOCTL calls, setting/getting/saving parameters, and receiving layer capabilities information. The time for pending command execution varies and the user should adjust the timeout prior to calling these commands appropriately.

2.3.2 DaqBIOS & Network Security

The PowerDNA Cube may be connected to the Internet, posing virtually no risk to the network it is hosted on. Several features make the PowerDNA Cube next to invulnerable for external attack, in descending order:

1. The PowerDNA Cube has only one UDP open port. By default, this port is 6334 – falling in the IANA unassigned port range 6323-6342. Default secu-

rity hole scanners will either skip UDP scanning, or skip scans of this range, expecting no useful protocols to run in this range.

2. The only protocol running on the cube is DaqBIOS – an unpublished protocol with no known exploits. If UDP port 6334 is discovered, it is unusable by anyone who does not understand the protocol.
3. Commands over the network that involve a change to the IOM memory or settings require a passwordAny command that changes internal state of the cube requires user password to be supplied. The password is stored in the encoded NVRAM area of the RTC chip. Any command that changes non-volatile memory requires a super-user password. The password is supplied over DQ protocol.
4. To prevent disruption of the experiment, the cube has the option to be locked onto an IP/port pair. For compatibility, locking/unlocking is disabled by default. When the locking option is enabled and the host PC establishes communication with the cube, the cube locks on to the host's IP/port pair and will listen for commands only from the locked host – until the host unlocks/releases the cube. Other PCs can only request cube configuration and status requests (e.g., IOM_25431 with AI-201 layer in slot 0 is currently in Locked state).

Finally, note that the PowerDNA Cube has no known exploitable daemons (e.g., Ms-IIS for http, ftp, etc.)

Chapter 3 DaqBIOS Engine

The DaqBIOS Engine (DQE) is organized as a PowerDNA shared library with which a user application is linked. It is a set of functions and data structures, implementing the DaqBIOS data acquisition protocol. DQE provides all functions necessary to interact with IOMs over the network.

DQE functions are executed within the user process; however, DQE may create additional execution threads for its purposes. Different user applications can use DQE simultaneously. Every process gets its own copy of DQE. DQE implements interlock mechanisms, preventing using of a single IOM by two processes and a single layer in exclusive modes.

DQE is used to simplify PowerDNA programming and shift data contingency and buffering responsibility from a user application to the library.

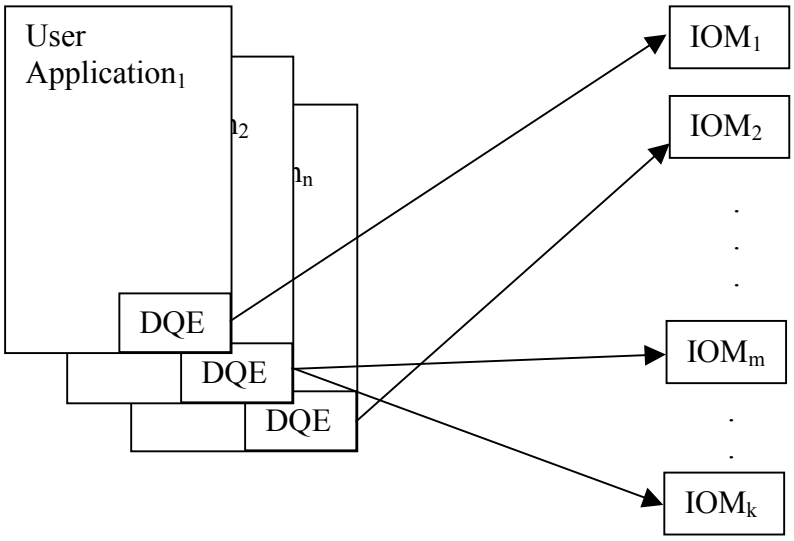


Figure 3-1. User Application/DQE/IOM Interaction.

In the above figure, note that one user application may interact with more than one IOM. The converse is not true.

3.1 Basic Architecture

DaqBIOS Engine consists of the following parts:

- **Sending thread/periodic task (multimedia timer callback under Windows)**
  This piece of code periodically wakes up and checks the *command queue* (CQ) of each IOM accessed by the process. It sends one or more commands per IOM per execution cycle and marks it as “waiting for response” so that it isn’t sent the next time. See also *command queue entry* below. There is a single sending thread in every DQE.

- **Receiving Thread**
  There is exactly one receiving thread per each IOM. This thread listens to the IOM, receives packets, and routes them to the input buffers according to the IOM's *command queue*. When a packet arrives from the IOM, the receiving thread looks up the corresponding entry in the command queue and then relocates the packet to the ring buffer. If there is no corresponding CQ entry, the packet is discarded. If there is a call-back associated with the entry, the receiving thread calls it with the specified parameter.
- **IOM Table**
  The IOM table is a static array inside the library and is common to all processes. It contains information about all active IOMs being contacted from this host. It includes the list of layers and their options, the processes that are working with them (one process per IOM), and some additional control information. The IOM table access is often made from a critical section.
- **Command Queue**
  There is exactly one command queue per IOM. It is a double-linked list that keeps the descriptions (also called command queue entries) of all commands to be sent and all replies to be received to/from the corresponding IOM. The entries are parsed with the *sending thread* and later used by the *receiving thread*. They are put into the queue by `DqSendPkt()` and other DQE calls. The results (after the packets arrive) are used by `Directivity()` calls or DQE callbacks, specified in the command queue entry.
- **Buffer Control Block**
  This structure contains control information about Advanced Circular Buffer (ACB) or Data Map (DMap), such as device, subsystem, transfer list, expected byte rate, update period, etc.
- **Reader and Writer Threads**
  Reader and writer threads provide transfer of data to and from the packet ring buffer to the ACB or DMap. They are responsible for calling proper data conversion routine depending on the layer type and data format selected. They are also responsible for error correction.
- **Advanced Circular Buffer, Data Ma**p
  These are the data exchangers between the user application and FIFO devices (for ACB) or groups of snapshot devices (for DMap) on IOM.

3.2 Threads and Function

Every instance of DQE has one *sending* and one *receiving* thread. When a process allocates an ACB or a DMap, DQE starts two additional threads. One of them is called *writer* thread and another one *reader* thread. The purpose of these threads is to transfer data from the ACB to the ring buffer for output and from the ring buffer to the ACB for input. The sending or receiving thread wakes the threads up when data needs to be transferred to/from the ring buffer.

3.3 IOM Data Retrieval and Data Conversion

The reader and writer threads call a conversion routine that converts data from the raw format represented in the ring buffer into a floating point representation of volts or other engineering units. If conversion parameters (offset and coefficient) weren't supplied upon creation of ACB or DMap, the data conversion routine converts raw data into native representation – Volts.

Chapter 4 Real-time Operation with an IOM

This section discusses how to perform data mapping and streaming under control of a real-time operating system. The reason for making a separate chapter for real time operation is that writing real-time code can be done more efficiently without using the DQE. Therefore, this section discusses programming of streaming and data mapping operations at low-level.

4.1 Simple I/O

Simple I/O mode, which is commonly associated with lower speed systems, may also be used for real-time applications with a real-time operating system. The key requirement is not speed of operation but rather that all timing be deterministic and that no time deadline be missed.

4.2 Real-time Data Mapping (RtDmap)

Direct data mapping is a mechanism that allows you to create areas of input and output data that mirror data values on the input and output lines of networked IOMs. The following diagram illustrates the structure of DMap operation.

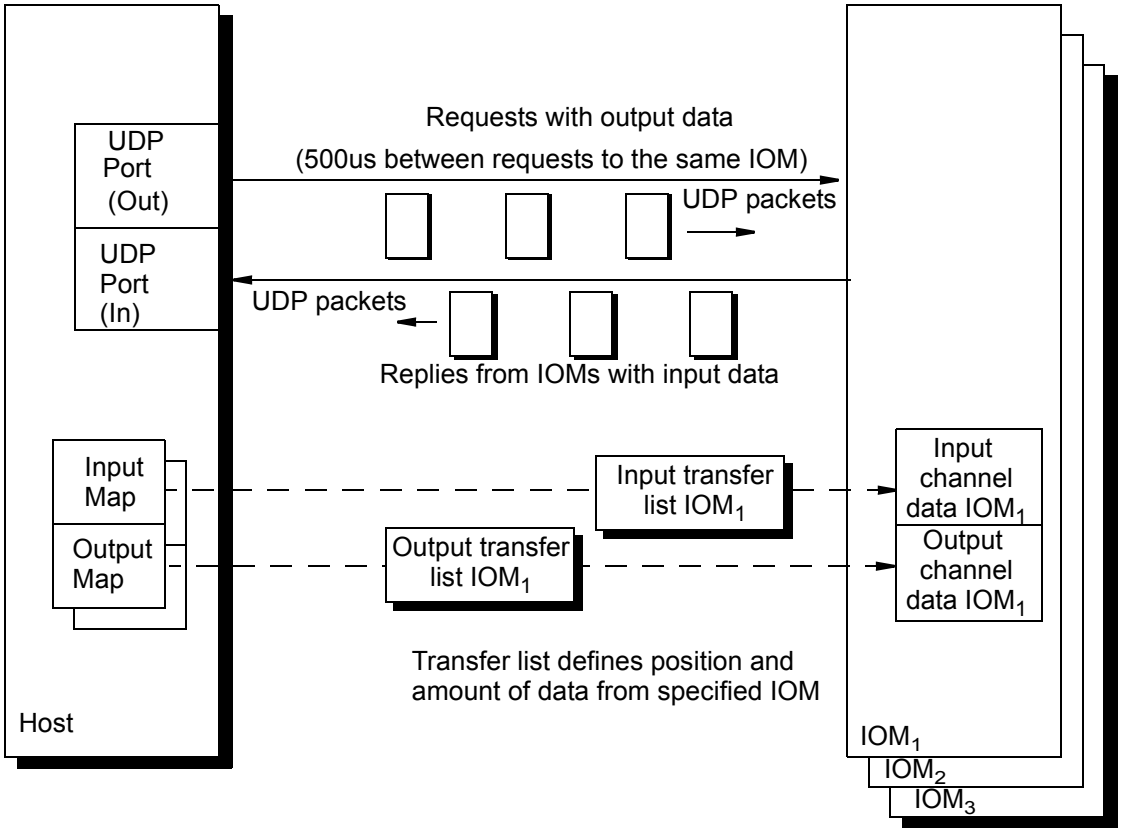


Figure 4-1. DMap Operation

Every DMap has its input and output maps and can work with a single multi-module IOM. Two DMaps can work with the same IOM, but must address different I/O boards (devices) within the IOM.

The maximum size of a DMap is limited to the size of a single packet – 510 bytes, which means that a DMap can be updated by receiving the data contained within a single new packet. Also, DMap allows representation of data either in raw or engineering units (volts by default).

In DMap mode, I/O devices perform at a rate sufficient to update input points fast enough to provide a fresh input reading with every reply packet. The output runs at a rate capable of updating outputs before the next portion of data arrives. Therefore, DMap mode meets the requirements of “hard” real-time operation.

4.2.1 Data Replication over the Network

DMap can be used for input data replication across a local area network if workstation NICs are set into promiscuous mode and receive all reply packets from the UDP interface. DMap can also be used in homogenous networks of IOMs in which IOMs exchange data between each other.

4.2.2 RtDmap Functional Description

The RtDmap API, described in this section, gives easy access to DMap operation without requiring use of the DQEngine. For more detailed information, refer to the PowerDNA Reference Manual.

Operation is as follows:

At each tick of the IOM clock, the IOM firmware scans the configured channels and stores the result in an area of memory called the DMap.

The host PC keeps its own copy of the DMap and synchronizes it periodically with the IOM’s version of the DMap. The rate at which the host transfers packets is controlled by the host and is usually set at a rate less than half the scan rate of the IOM clock.

This mode is very useful when the host computer runs a real-time operating system because it ensures that the host refreshes its DMap at deterministic intervals (hard real-time). It optimizes network transfer by packing all channels from multiple I/O boards into a single UDP packet, thus reducing the network overhead.

The standard (non-real-time) low-level API (**DqDmap***** functions) use the DqEngine (DQE) to refresh the DMap at a given rate and to retry a DMap refresh request if, for some reason, a packet is lost. Use of the DQE is necessary on desktop-oriented operating systems to ensure that the DMap is refreshed periodically, but is not required (and not recommended) for use with hard real-time operating systems.

The following is a list of the real time data mapping functions, with short descriptions of each. (Note that each of these functions does not use DQE.

Table 4-1. RtDMap API Functions

Function	Description
DqRtDmapInit	Initializes the specified IOM to operate in DMAP mode at the specified refresh rate.
DqRtDmapAddChannel	Adds one or more channels to the DMAP.
DqRtDmapGetInputMap	Gets a pointer to the beginning of the input data map allocated for the specified device
DqRtDmapGetInputMapSize	Gets the size in bytes of the input map allocated for the specified device.
DqRtDmapGetOutputMap	Gets a pointer to the beginning of the output data map allocated for the specified device.
DqRtDmapGetOutputMapSize	Gets the size in bytes of the output map allocated for the specified device.
DqRtDmapReadScaledData	Reads and scales the data stored in the input map for the specified device. **Note**: The data read is the data transferred by the last call to DqRtDmapRefresh(). This function should only be used with devices that acquire analog input data such as the AI-2xx series layers.
DqRtDmapReadRawData16	This function reads raw data from the specified device as 16-bit integers. **Note**: The data read is the data transferred by the last call to DqRtDmapRefresh(). This function should only be used with devices that acquire 16-bit wide digital data such as the AI-4xx series layers.
DqRtDmapReadRawData32	This function reads raw data from the specified device as 32-bit integers. **Note**: The data read is the data transferred by the last call to DqRtDmapRefresh(). This function should only be used with devices that acquire 32-bit wide digital data such as the DIO-4xx series layers.
DqRtDmapWriteScaledData	This function writes scaled data to the output map of the specified device. **Note**: The data written is actually transferred to the device on the next call to DqRtDmapRefresh(). This function should only be used with devices that generate analog data such as the AI-3xx series layers.

Table 4-1. RtDMap API Functions (Cont.)

Function	Description
DqRtDmapWriteRawData16	This function writes 16-bit wide raw data to the specified device. **Note**: The data written is actually transferred to the device on the next call to **DqRtDmapRefresh()**. This function should only be used with devices that generate 16-bit wide digital data such as the DIO-4xx series layers.
DqRtDmapWriteRawData32	This function reads raw data from the specified device as 32-bit integers. **Note**: The data written is actually transferred to the device on the next call to **DqRtDmapRefresh()**. This function should only be used with devices that acquire 32-bit wide digital data such as the AI-4xx series layers.
DqRtDmapStart	This function starts operation and the IOM updates its internal representation of the map at the rate specified in **DqRtDmapCreate.**
DqRtDmapStop	This function stops operation and the IOM stops updating its internal representation of the data map.
DqRtDmapRefresh	This function refreshes the host's version of the map by downloading the IOM's map. **Note**: The IOM automatically refreshes its version of the data map at the rate specified in **DqRtDMapInit()**. This function needs to be called periodically (a real-time OS is necessary) to synchronize the host and IOM data maps.
DqRtDmapRefreshOutputs	This function refreshes the host's version of the map by downloading the IOM's map. **Note**: The IOM automatically refreshes its version of the data map at the rate specified in **DqRtDMapInit()**. This function needs to be called periodically (a real-time OS is necessary) to synchronize the host and IOM data maps.
DqRtDmapRefreshInputs	This function refreshes the host's version of the map by downloading the IOM's map. **Note**: The IOM automatically refreshes its version of the data map at the rate specified in **DqRtDMapInit().** This function needs to be called periodically (a real-time OS is necessary) to synchronize the host and IOM data maps.
DqRtDmapClose	This function frees all resources on the specified IOM allocated by the DMAP operation.

4.2.3 RtDmap Typical Program Structure

The following is a quick tutorial on use of the RtDmap API (with error handling omitted):

1. Initialize the DMAP to refresh at 1000 Hz.
   DqRtDmapInit(handle, &dmapid, 1000.0);
2. Add channel 0 from the first input subsystem of device 1.
   chentry = 0;
   DqRtDmapAddChannel(handle, dmapid, 1, DQ_SS0IN, &chentry, 1);
3. Add channel 1 from the first output subsystem of device 3.
   chentry = 1;
   DqRtDmapAddChannel(handle, dmapid, 3, DQ_SS0OUT, &chentry, 1);
4. Start all devices that have channels configured in the DMAP.
   DqRtDmapStart(handle);
5. Update the value(s) to be output to device 3.
   outdata[0] = 5.0;
   DqRtDmapWriteScaledData(handle, dmapid, 3, outdata, 1);
6. Synchronize the DMAP with all devices.
   DqRtDmapRefresh(handle, dmapid);
7. Retrieve the data acquired by device 1.
   DqRtDmapReadScaledData(handle, dmapid, 1, indata, 1);
8. Stop the devices and free all resources.
   DqRtDmapStop(handle, dmapid);
   DqRtDmapClose(handle, dmapid);

4.3 Real-time Variable-size Data Mapping (RtVmap)

This feature is similar to RealTime DMap operation (see “Real-time Data Mapping (RtDmap)” on page 24) except that the size of the data transfer is variable.

The RtVmap API, like the RtDmap API, gives easy access to the VMap operating mode without needing the DqEngine.

VMap is a protocol developed for *control* applications in which the ability to get immediate real-time data may be more important than receiving a continuous gapless flow of the data. VMap is also well-suited for many real-time messaging applications, as described below.

Messaging layers are normally supported by the Msg protocol, which shares the same buffering mechanism as the ACB protocol. The Msg protocol buffer receives packets and delays releasing newer packets to the user application until it re-requests and receives all the packets in the previous message stream. Although this protocol does provide a gapless stream of messages, it is not suited for real-time operation because timing is not deterministic.

VMap, however, can provide a real-time alternative to the Msg protocol for messaging devices — at the expense of restricting the ability to recover lost packets. It shifts the decision about whether or not to recover the lost packet to the

user application. A set of hard real-time VMap functions is listed below in **Table 4-2**.

At high level, VMap is very similar to DMap. A user creates a VMap with output and input buffers and add channels/layers of interest to it. VMap packets also have additional fields. First of all, there is a flag field required to guarantee continuity of messaging data. Second, an output buffer adds a pair of fields for each channel in the map at its header. The first field provides the IOM with information on how much data is to be transmitted for that channel; the second field defines the maximum size of data to be received from that channel. The offsets of the output data in the buffer should be in agreement with the size of the data in the buffer header.

An input packet also contains a flag field as well as the number of bytes actually written, actually received plus (optionally) the number of bytes available in the receive FIFO, and the room available in the transmit FIFO. This feature allows flexibility in allocating packet slices for various channels. Each time packets are exchanged between host and IOM, the user application can select different sizes for outgoing and incoming data, taking into consideration the amount of data required to be sent and the size of data accumulated in the receiving FIFO. If you don't use a channel at this time, you should set *size to send* and *size to receive* to zero. The header has a fixed width set up before starting VMap operation; the header size cannot be changed on the fly even if the channel is no longer in use.

Note that VMap has a function that returns the VMap ID to the user for use in systems that have multiple IOMs. Since packets from multiple IOMs may be received by the host out of time sequence, this function gives the host the information necessary to call the right VMap processing routine for that packet.

Table 4-2 is a list of the real-time variable data mapping functions, with short descriptions of each. Refer to the PowerDNA Reference Manual API for more detailed information.

Table 4-2 . RtVmap API Functions

Function	Description
DqRtVmapInit	Initializes the specified IOM to operate in VMap mode at the specified refresh rate.
DqRtVmapAddChannel	This function adds a channel to <vmapid> VMap. The function adds an entry to the transfer list. Channels with an SSx_IN subsystem are added to the transfer list; channels with an SSx_OUT subsystem are added to the output transfer list. Channel in <cl> should be defined in the standard way including channel number, gain, differential, and timestamp flags. Configuration <flags> for the input subsystem can include DQ_VMAP_FIFO_STATUS to report back the number of samples in the input FIFO waiting to be requested (after output packets are processed). Configuration <flags> for the output system can include DQ_VMAP_FIFO_STATUS to report back the number of samples that can still be written into the output FIFO before it becomes full (after all transmitted bytes have been written). Note that this flag adds a uint16 word to the standard header for an input packet, thus inceasing te size of the header and decreasing the size available for data. <clSize> specifies the maximum number of array entries. The Output VMap buffer, which transfers data from host to IOM, has the structure shown in Table 4-3 on page 33.

Table 4-2 . RtVmap API Functions (Cont.)

Function (Cont.)	Description
DqRtVmapAddChannel (cont.)	The total length of the buffer cannot exceed the size available in the UDP packet minus the combined size of the DQPKT and DQQRRD headers. The output buffer of VMap contains information to be written to the channel output FIFOs of the messaging layer (as well as theanalog or digital layers equipped with hardware FIFOs). It also specifies the number of bytes to read from the same channel, if any. Data for or from the channel should be assembled in accordance with the message structure of that layer. Flags are used to make data ready and to acknowledge packet execution. This feature arises because VMap relies on continuous data flow compatible with messaging layers as well as continuous acquisition and output and thus must ensure continuuty of data. In other words, no message can be sent or received twice. The Input VMap buffer, which transfers data from IOM to host, has the structure shown in Table 4-4 on page 34. The Input VMap buffer contains information showing how much data was actually retrievded from the channel FIFO and how much of the data in the output buffer has been written to that channel. The header size cannot be changed after **DqRtVmapStart()** is called. In other words, after a channel is added using **DqRtVmapAddChannel()**, the header size increases by one in the output packet and by one or two (if **DQ_VMAP_FIFO_STATUS** is set) uint16 words in the input packet. The header allocation cannot be changed until the current VMap is destroyed and a new one is created. If youwould like to send zero bytes for that channel or receive zero byttes froma a channel, VMap fills the appropriate header field with 0. **Note:** Each call to **DqRtVmapAddChannel()** adds one or more transfer list entries. Ther indices are zero-origin, sequential, and cumulative. For example, if one adds five channels in the first call to this function, the transfer list index of the last channel is 4. For the next call, the last channel will have transfer list index equal to 9.
DqRtVmapStart	This function sets up all parameters needed for operation – channel list and clock; transfers and finalizes the transfer list. The function also parses the transfer list and stores offsets of the headers for each transfer list entry. If clocked devices (AIn/AOut) are used, the function programs devices at the rate specified in **DqRtDmapInit**.

Table 4-2 . RtVmap API Functions (Cont.)

Function (Cont.)	Description
DqRtVmapStop	This function stops operation and the IOM stops updating its internal representation of the data map.
DqRtVmapClose	This function destroys the <**vmapid**> VMap.
DqRtVmapRefresh	This function refreshes the host version of the map by downloading the IOM map. Use the **DQ_VMAP_REREQUEST** flag if you want to re-request the failed transaction instead of performing a new one. In such case, the dqCounter in the **DQPKT** header will not be incremented by the host and the IOM will not output/input a new message if the IOM already processed it (reply packet lost). Instead, the IOM will reply with a copy of the previous packet. If the IOM never received the packet, it will process it in the normal way. **Note:** The IOM automatically refreshes its version of the data map at the rate specified in **DqRtVMapInit()**. This function should be called periodically (a real time OS is required) to synchronize the host and IOM data maps).
DqRtVmapRefreshOutputs	This function refreshes the host version of the map by downloading the IOM map. Use **DQ_VMAP_REREQUEST** flag if you want to re-request the failed transaction instead of performing a new one. **Note**: This function needs to be called periodically (real-time OS is required) to synchronize host and IOM data.
DqRtVmapRefreshInputs	This function refreshes the host version of the map by downloading the IOM map. **Note**: This function needs to be called periodically (a real-time OS is necessary) to synchronize the host and IOM data maps.
DqRtVmapGetInputPtr	This function gets the pointer to the beginning of the input data allocated for the specified entry. **Note**: This function can be called only after packet is received.
DqRtVmapGetOutputPtr	This function gets the pointer to the beginning of the output data allocated for the specified entry. **Note:** This function can be called only after transmission size for all channels is written.
DqRtVmapGetInputMap	Get pointer to the beginning of the input data map allocated for the specified device. **Note:** This fuunction can be called only after a packet is received, because the actual positions of the input data in the packet for each transfer list entry depend on the number of bytes actually retrieved from the input FIFO. If the number of bytes retrieved is less than requested, VMap will not waste the space in the packet, but rather will pack it to decrease transmission time.

Table 4-2 . RtVmap API Functions (Cont.)

Function (Cont.)	Description
DqRtVmapGetOutputMap	This function gets the pointer to the beginning of the output data map allocated for the specified entry. **Note:** This function can be called only after transmission size for all channels is written. Actual offsets of the data for each channel in the output packet depend on the size of the data stored in the packet header. Thus, this function makes sense only if all data is placed into the packet.
DqRtVmapAddOutputData	This function copies data into the output packet and returns the number of bytes left in the packet. **Note:** This function modifies the output packet.This function must be called *before* **DqRtVmapRefresh()**.
DqRtVmapRqInputDataSz	This function requests the number of bytes to receive in the input packet. It returns the number of bytes left in the buffer, the actual size requested, and the pointer to the location where the data will be stored. **Note:** This function modifies the output packet.This function must be called *before* **DqRtVmapRefresh().**
DqRtVmapGetInputData	This function copies data from the input packet and returns the number of bytes copied and the size available in the input FIFO. **Note:** This function must be called *after* **DqRtVmapRefresh()**.
DqRtVmapGetOutputDataSz	This function examines the input packet and returns the number of bytes copied from the output packet to the output FIFO and (optionally) how much room is available in the output FIFO. **Note:** This function must be called *after* **DqRtVmapRefresh()**.

Table 4-3. Output VMap Buffer

Size	Flags (uint16)
Size to write to Ch0 (uint16)	Size to write to ChN (uint16)
•	•
•	•
•	•.
Size to read from Ch0 (uint16)	Size to read from ChN (uint16)
Data for Ch0 (of specified size)	
•	
•	
•	
Data for ChN (of specified size)	

Table 4-4. Input VMap Buffer

Size	Flags (uint16)
No. of bytes retrieved from Ch0 (uint16)	No. of bytes remaining in Ch0 (uint16, optional)
• • •	• • •.
No. of bytes retrieved from ChN (uint16)	No. of bytes remaining in ChN (uint16, optional)
No. of bytes written to Ch0 (uint16)	No. of bytes that can be written to Ch0 (uint16, optional)
• • •	
No. of bytes written to ChN (uint16 optional)	No. of bytes that can be written to ChN (uint16, optional)
Data from Ch0 (of specified retrieved size)	
• • •	
Data from ChN (of specified retrieved size)	

4.3.1 RtVmap Typical Program Structure

The following is a short tutorial example that uses the RtVmap API (handling of error codes is omitted):

1. Initialize the VMAP to refresh at 1000 Hz:
   DqRtVmapInit(handle,&vmapid,1000.0);
2. Configure device input output ports using the appropriate DqAdv*** function. For example, the following configures an ARINC-429 device (DEVN) input and output ports 0 to run at 100kbps with no parity and no SDI filtering.
   DqAdv566SetMode(handle, DEVN, DQ_SS0OUT, 0, DQ_AR_RATEHIGH | DQ_PARITY_OFF);
   DqAdv566SetMode(handle, DEVN, DQ_SS0IN, 0, DQ_AR_RATEHIGH|DQ_PARITY_OFF|DQ_AR_SDI_DISABLED);
3. Add input port 0 to VMAP, set flag to retrieve the status of the input FIFO after each transfer:
   chentry = 0;
   flag = DQ_VMAP_FIFO_STATUS;
   DqRtVmapAddChannel(handle, vmapid, DEVN, DQ_SS0IN, &chentry, &flag, 1);
4. Add output port 0 to VMAP, set flag to retrieve the status of the output FIFO after each transfer.
   chentry = 0;

```
flag = DQ_VMAP_FIFO_STATUS;
DqRtDmapAddChannel(handle, vmapid, DEVN, DQ_SS0OUT, &chentry,
&flag, 1);
```

5. Enable ARINC-429 ports.
   `DqAdv566Enable(handle, DEVN, TRUE);`
6. Start all devices that have channels configured in the VMAP.
   `DqRtVmapStart(handle, vmapid);`
7. Prepare ARINC word to send through port 0 and update VMAP.
   ```
   uint32 arincWord = DqAdv566BuildPacket(data, label, ssm, sdi, parity);
   DqRtVmapAddOutputData(handle, vmapid, 0, sizeof(uint32), &accepted,
   (uint8*)&arincWord);
   ```
8. Specify that we wish to receive up to MAX_WORDS words received by port 0.
   ```
   DqRtVmapRqInputDataSz(handle, vmapid, 0,
   MAX_WORDS*sizeof(uint32), &rx_act_size, NULL);
   ```
9. Synchronize the VMAP with all devices.
   `DqRtVmapRefresh(handle, vmapid, 0);`
10. Retrieve the data received by port 0.
    ```
    uint32 recvWords[MAX_WORDS];
    DqRtVmapGetInputData(handle, vmapid, 0, MAX_WORDS*sizeof(uint32),
    &rx_data_size, &rx_avl_size, (uint8*)recvWords);
    ```
11. We can also check how much data was actually transmitted during the last refresh.
    ```
    DqRtVmapGetOutputDataSz(handle, vmapid, 0, &tx_data_size,
    &tx_avl_size);
    ```
12. Stop the devices and free all resources.
    ```
    DqRtVmapStop(handle, vmapid);
    DqRtVmapClose(handle, vmapid);
    ```

Chapter 5 Asynchronous Real-time Operation with an IOM

The PowerDNA DaqBIOS works on a request/response scheme in which the host requests certain actions to be performed by the cube. The cube performs these actions and returns the status and/or data.

Thus, request and response packet pairs should be matched. This is accomplished in the same way that each `DqAdv…()` command and `DqRtVmapRefresh()` command sends a request and waits until a timeout occurs or a response is received. If a response is received from a command other than the one that requested it, this response packet is discarded.

Thus, when a program has multiple threads, it is important to safeguard integrity of the request/response interface. This is normally done in one of two ways:

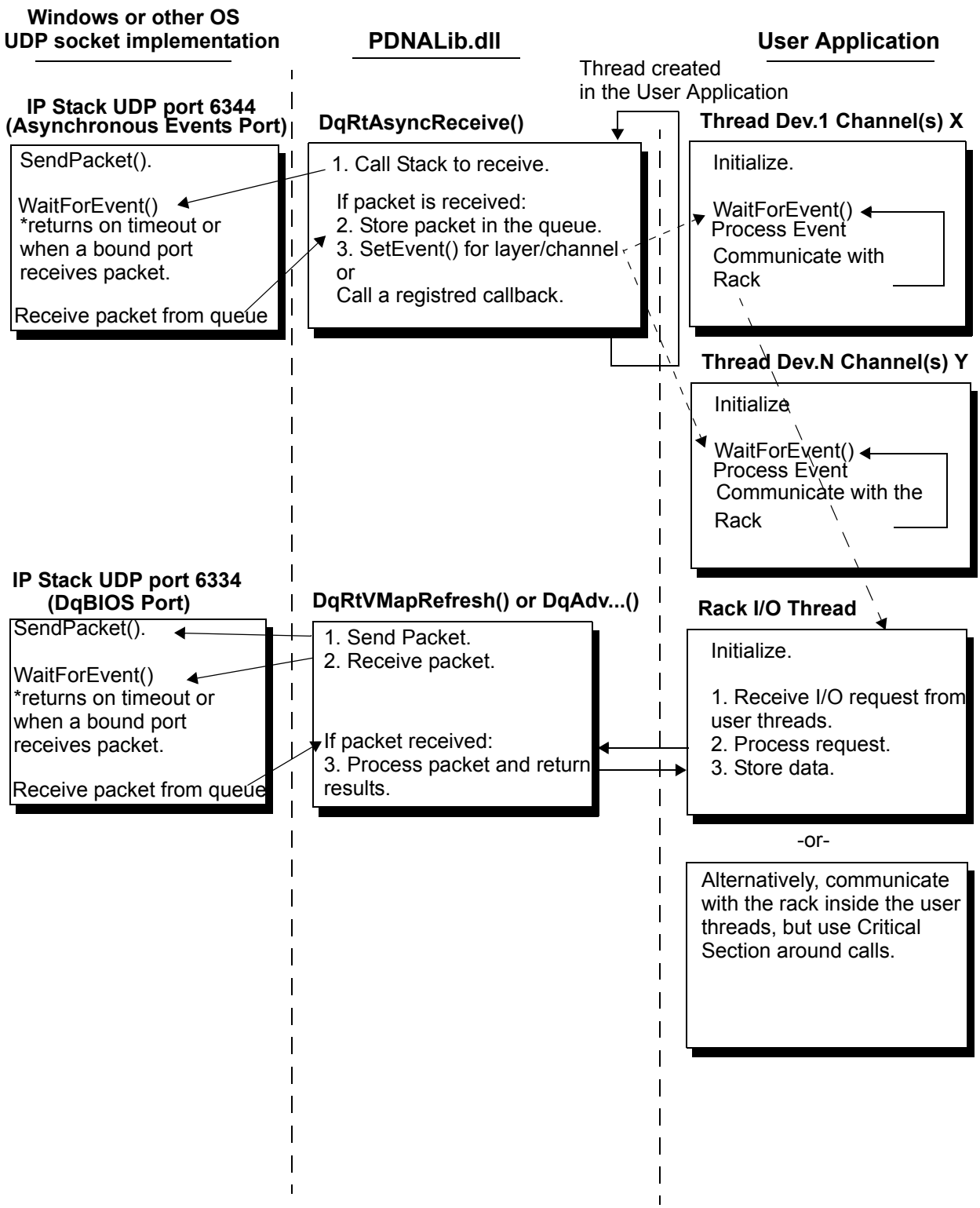
1. Use a separate thread to communicate with the cube and use *events* (or other host OS primitives like semaphores or messages) to exchange requests and results with this thread.
2. Use OS-provided primitives to ensure that no `DqAdv…()` or `DqRt…()` call can happen while another thread is waiting for a return packet in the PDNALib.dll

When using asynchronous events, it is also important to have a single thread responsible for receiving asynchronous event packets and dispatch them to appropriate user application threads. This is accomplished by creating a thread in the user application, initializing asynchronous operation (i.e., what layers, channels, and event types are involved), creating and assigning event primitives for each event, and then calling `DqAsyncReceive()` in the loop.

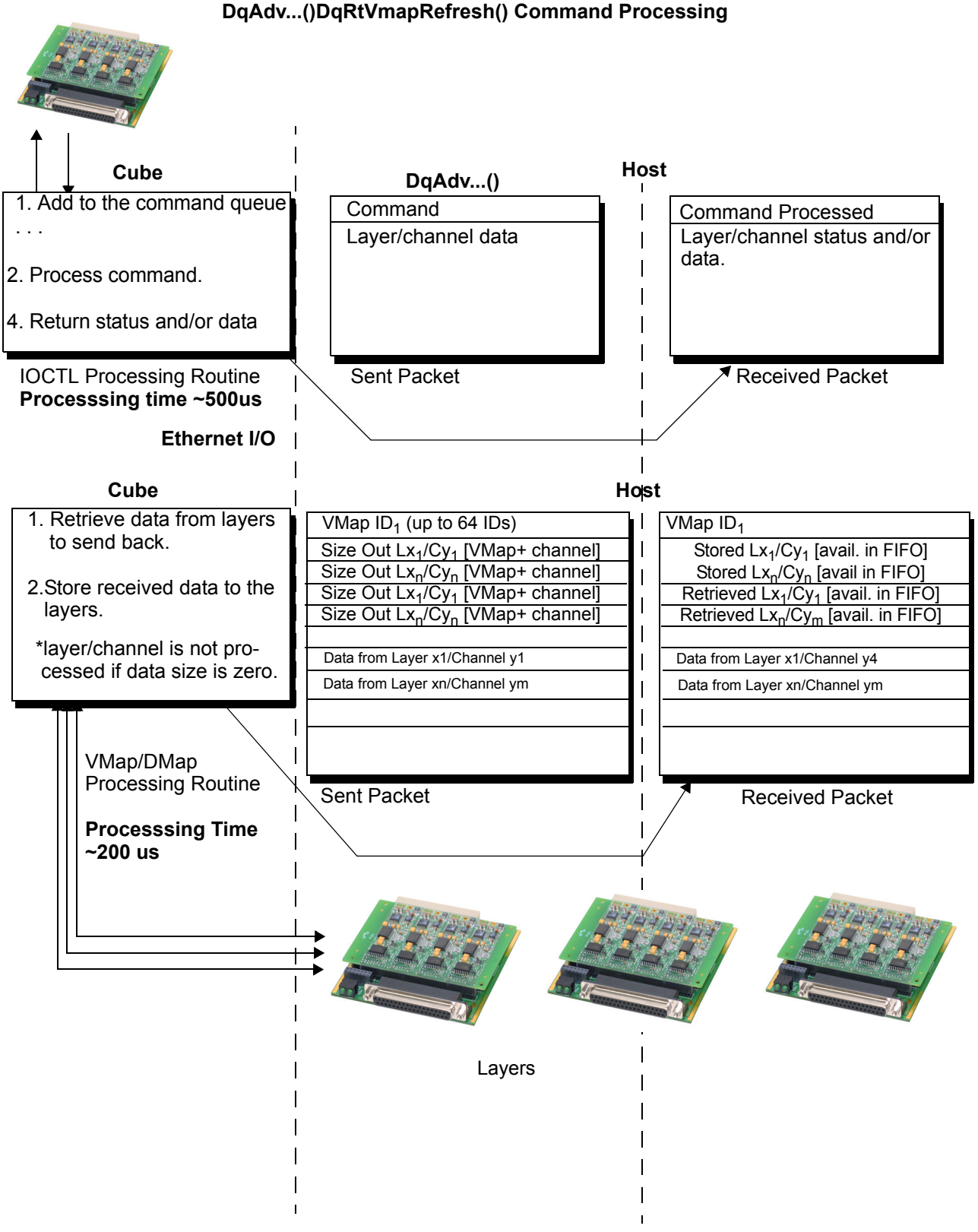
`DqAsyncReceive()` waits for the socket until a asynchronous packet is received (or timeout occurs), then stores it in the queue, and sets an appropriate event. Then, the thread continues its execution and calls `DqAsyncReceive()` again.The OS is responsible for delivering an event to the user threads dedicated to processing it and re-scheduling threads accordingly to assigned priorities and available resources.

Refer to the following figures for a more detailed description o the Application Design and Command Processing.

Multithreaded Application Design with Asynchronous Events



DqAdv...()DqRtVmapRefresh() Command Processing



Index