FAQ: Application Architecture

2021

UEI products support a variety of application architecture requirements, including multiprocessing, multithreading, and asynchronous events. These features help maximize the performance and speed of your application by optimizing the use of limited hardware resources. In addition, a well-designed architecture can improve code organization, readability, and maintainability.

This document goes over design considerations, best practices for communicating with UEI's input/output modules (IOMs), and answers to frequently asked questions. Before reading further, you should already be familiar with the different data acquisition (DAQ) modes offered by UEI, which are described in FAQ - Data Acquisition Modes.

1. Processes vs. Threads

Processes and threads are implemented by the operating system. Splitting an application into multiple processes and/or threads enables tasks to run concurrently and with different priorities.

A process executes a program within its own memory container, protected from other processes. A CPU with four cores could be set up to run four simultaneous processes, one on each core, so that none of the cores sit idle. Multiple processes can also run on a single core, although then the CPU would have to switch back and forth between the processes (known as context switching).

A process consists of one or more threads which share memory. The context switching time for threads is significantly shorter than for processes. However, since the threads within a process access the same memory, multithreading can cause data corruption issues if not carefully managed.

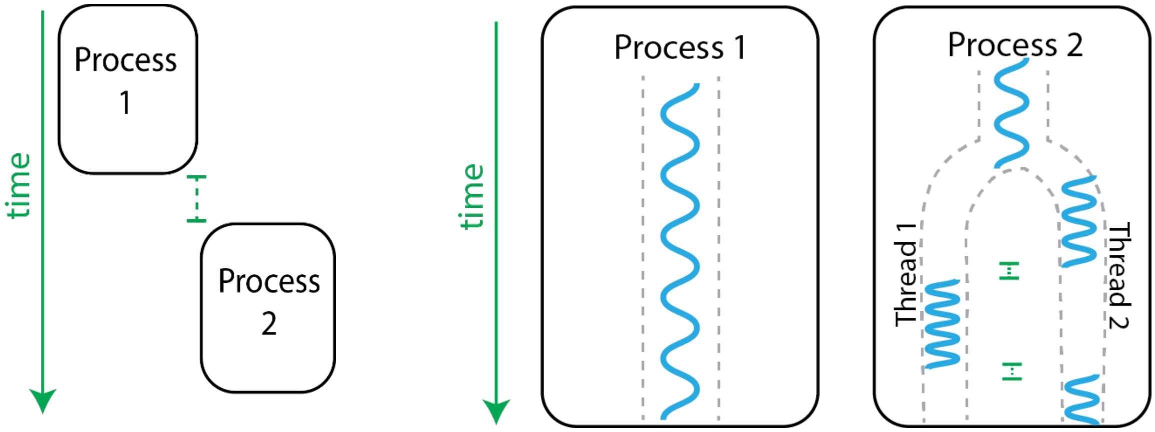


Figure 1: On the left, two processes run on a single CPU, with time for context switching. On the right, two processes run on two CPU cores. Process 1 executes a single thread, while Process 2 branches the main thread into two threads. The context switch time between threads is shorter than for processes.

1.1. Can I access multiple boards from the same process?

Yes, boards on an IOM may be grouped together and accessed by a single process. A single process may also communicate with boards across multiple IOMs.

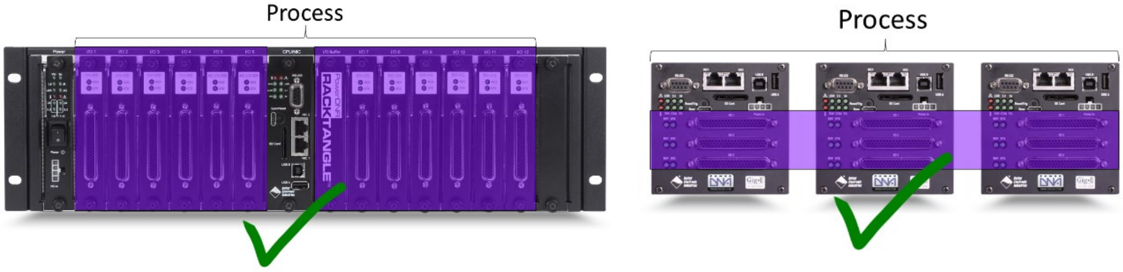


Figure 2: A process may access multiple IO boards.

1.2. Can I run multiple processes on an IOM?

The answer depends on the DAQ mode.

- The IOM firmware supports multiple Point-by-Point mode processes. However, please do not try to access the same IO board with different processes.
- We recommend running ACB, VMAP, and DMAP modes from a single process. If an ACB/VMAP/DMAP process is already running and the application tries to start a second ACB/VMAP/DMAP process, the firmware discards the configuration from the first process. It is ok to use multiple ACB/VMAP/DMAP threads within a single process, e.g. an ACB thread for one board and a DMAP thread for other boards.

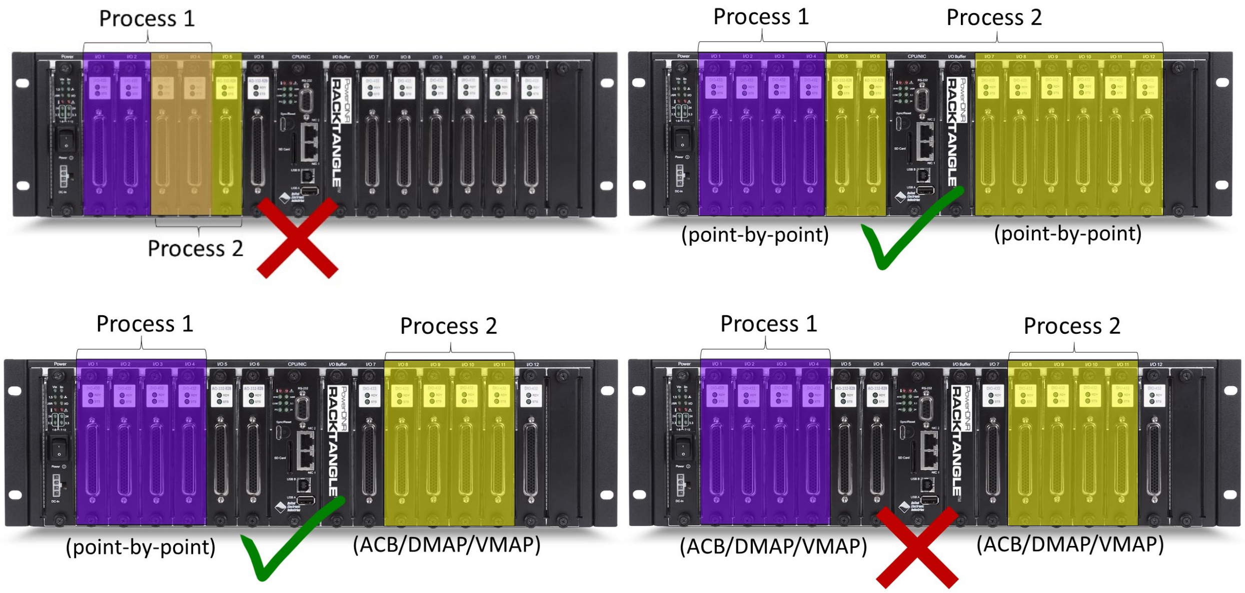


Figure 3: Two processes may not access the same IO board. Multiple Point-by-Point mode processes are allowed, but ACB/DMAP/VMAP should run from a single process.

1.3. Can I run multiple threads on an IOM?

Yes, blocks of functionality within a process may be split into threads. Perhaps each IO board gets its own thread, simplifying the code in case an IO board is replaced. UEI recommends following these best practices when accessing IO boards from threads:

- IO Boards of different types, or running at different rates, should run in dedicated threads.
- Multiple AIO/DIO boards all running at the same rate should run in the same thread.
- A messaging IO board should access all ports from the same thread.
    - If the ports run at different rates, it is possible to use one thread per port in Point-by-Point (PbP) mode only. Some boards have the ability to configure and enable ports one-by-one, while others configure/enable the board as a whole. In the latter situation, you can perform configuration in a single thread and then fork threads for the I/O part of the task.

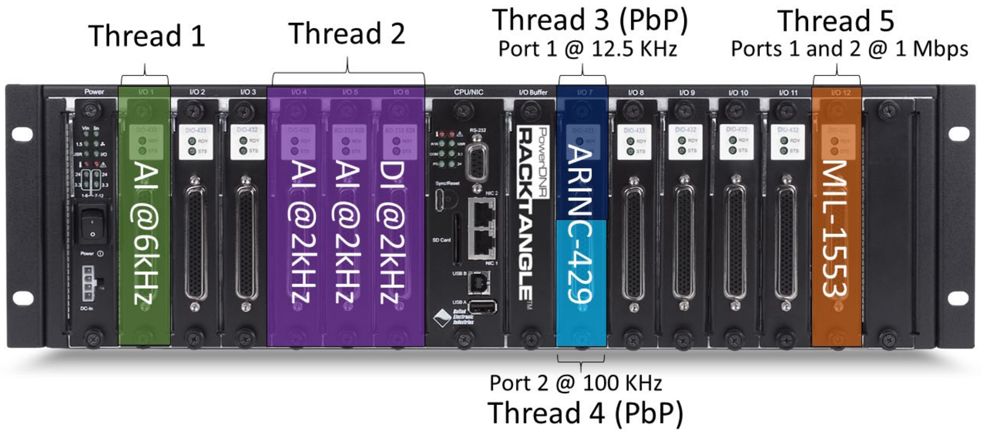


Figure 4: Example of a multithreaded application – Analog In (AI) and Digital In (DI) boards running at the same rate are controlled by a single thread (Thread 2), while an AI board at a different rate runs in another thread (Thread 1). If two ports on a Messaging IO board are set to different rates and use Point-by-Point mode, each can have its own thread (Threads 3 and 4). Two ports set to the same rate run under a single thread (Thread 5). Remember that Threads 3 and 4 must run within the same process.

Because these are guidelines rather than rules, please contact UEI Support to discuss deviations from these threading recommendations.

Data Acquisition Modes

You may select one of the following DAQ modes per IO board. Please do not try to access 1 IO board with 2 different DAQ modes.

- Point-by-Point (PbP)
- Real-time Data Map (RtVMap)
- Real-time Variable Data Map (RtVMap)
- Advanced Circular Buffer (ACB)
- Asynchronous Data Map (ADMap)
- Asynchronous Variable Map (AVMap)

Asynchronous Event Threads

In addition to the six DAQ modes listed above, UEI offers a special mode for Asynchronous Events. If the events happen infrequently, there is no reason to constantly query the IOM. Instead, an asynchronous thread can be set up to wait for data packets originated from the IOM.

While each IO board/port should only be controlled by a single DAQ thread, the board/port may have both a DAQ thread and an Asynchronous Events thread, if the board has input AND output subsystems. For example, a digital input board only has input subsystems and therefore can only have a single thread accessing it. A serial board will have input and output subsystems for each port; therefore, you may have one thread for writing data and an asynchronous thread for receiving data.

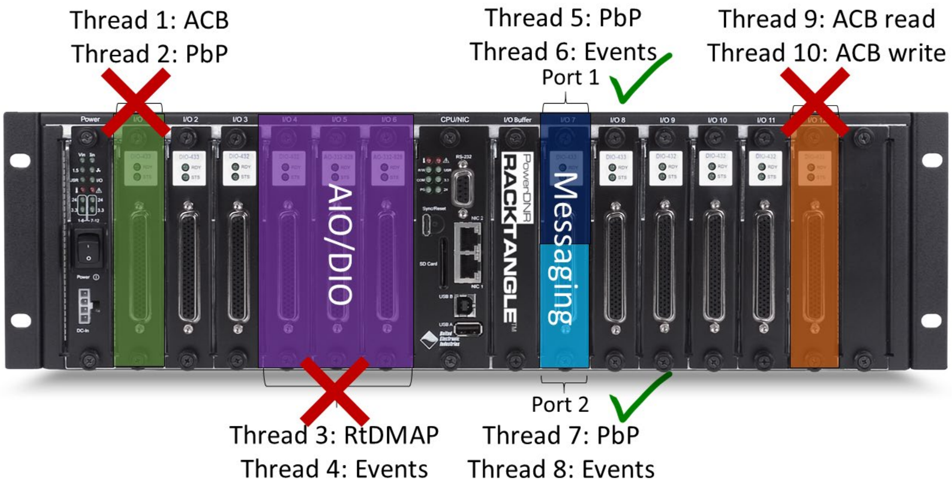


Figure 5: Avoid accessing an IO board with multiple DAQ modes (Threads 1 and 2) or multiple threads (Threads 3 and 4, Threads 9 and 10). The exception is for boards that support an additional Asynchronous Events thread. For example, a Messaging IO board with two independent ports can have up to 4 threads: 1 Point-by-Point thread and 1 events thread per port (Threads 5-8).

2. Communicating with the IOM

This section provides an overview of how to implement your application architecture. Communicating with the IOM involves the following steps:

1. **Connect to the IOM via Ethernet.**
2. **Get handle to the IOM in each process.**
3. **Get handle to the IOM for each thread (if necessary).**
4. **Send commands to specific IO boards using their Device Numbers.**
5. **Close out communications.**

Example code is included with all UEI software installations. Both low-level C examples and high-level examples are available. For more information about programming options, please refer to the "Getting Started with Your PowerDNA Application" guide.

NOTE: *Please do not mix low-level API (PowerDNA) and high-level API (Framework) in the same program.*

2.1. How can I use the Ethernet ports?

UEI systems can be deployed in a variety of configurations, including PowerDNA hosted mode and UEIPAC embedded mode.

- In PowerDNA mode, the IOM is an I/O slave under the control of a host PC via Ethernet. It is recommended for all UEI hardware to run on a separate network or virtual local area network (VLAN).
- In UEIPAC mode, the code is cross-compiled on the PC and transferred to the hardware, typically via Ethernet. The UEIPAC may then be disconnected from the PC and run as a stand-alone embedded controller or data logger.

Each UEI IOM comes with two Ethernet ports. While it may be tempting to talk to the IOM with two PCs, please use only 1 Ethernet Port and 1 PC. Rather than speed things up or add redundancy, two control PCs will cause network collisions and errors.

Why are there two Ethernet ports? The second Ethernet port may be used for Diagnostics. The "SampleDiagnostics" example code shows how to read and print out diagnostic values from an IOM.

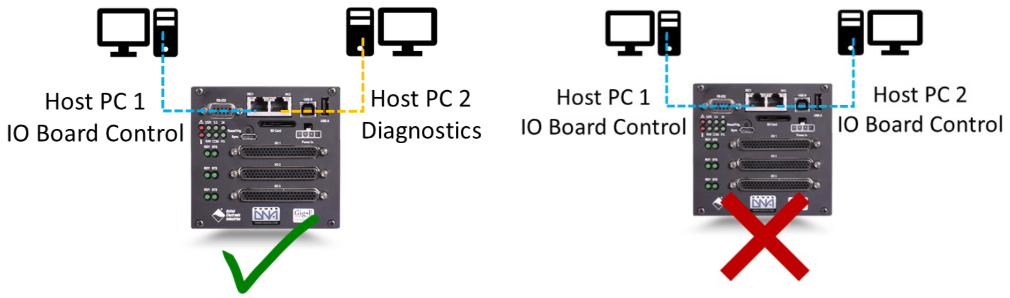


Figure 6: Use only one Ethernet port to control the IOM. The other port may be used for diagnostics or left disconnected.

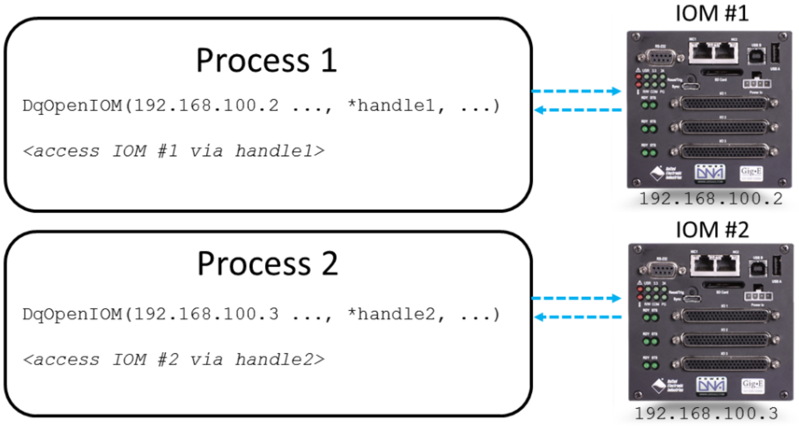
NOTE: *UEI hardware uses onboard firmware and drivers to communicate with the application running on your host PC. Each version of UEI software corresponds with a dedicated firmware version. The IOM is shipped with pre-installed firmware and a matching software installation. If you upgrade your software installation, you must also update the firmware on your Cube or RACK CPU. Instructions for updating firmware are provided in the UEI chassis user manuals.*

2.2. How do I get a handle to the IOM?

Each process needs to obtain a handle to the IOM by calling the function `DqOpenIOM()`. `DqOpenIOM` opens communications with the IOM specified by its IP address and outputs a handle.

```
DqOpenIOM(*IP ..., *handle, ...)
```

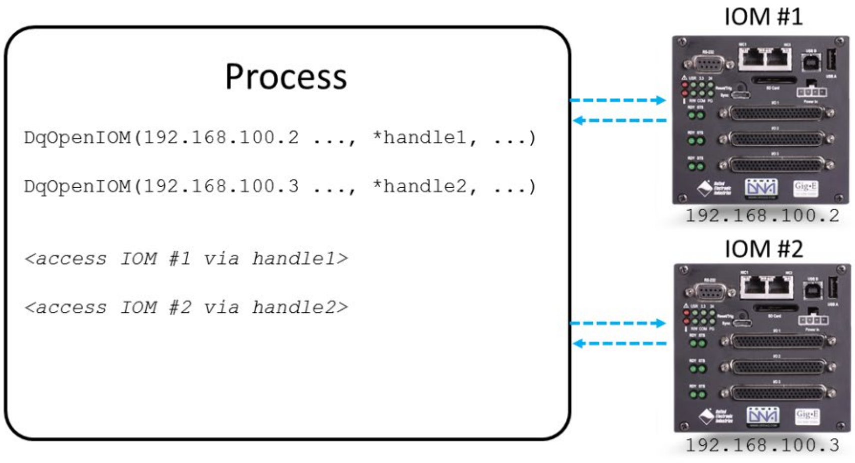
All other commands to the IOM will require the handle as an input. To learn how to determine the IOM's IP address, please see the "PowerDNA Quick Start" guide.



```
DqOpenIOM(192.168.100.2 ..., *handle1, ...)

<access IOM #1 via handle1>
```

```
DqOpenIOM(192.168.100.3 ..., *handle2, ...)

<access IOM #2 via handle2>
```

Figure 7: Before communicating with an IOM, a process must first obtain a handle to the IOM using `DqOpenIOM()`.



```
DqOpenIOM(192.168.100.2 ..., *handle1, ...)

DqOpenIOM(192.168.100.3 ..., *handle2, ...)


<access IOM #1 via handle1>

<access IOM #2 via handle2>
```

Figure 8: It is OK to call `DqOpenIOM()` multiple times in one process if each call has a unique IP address.

For more information on `DqOpenIOM()`, please see the "PowerDNA API Reference Manual."

2.3. Do I need a handle for each thread?

If the application needs to access an IOM from multiple threads, each thread *may* need to obtain a new handle to the IOM by calling the function `DqAddIOMPort()`.

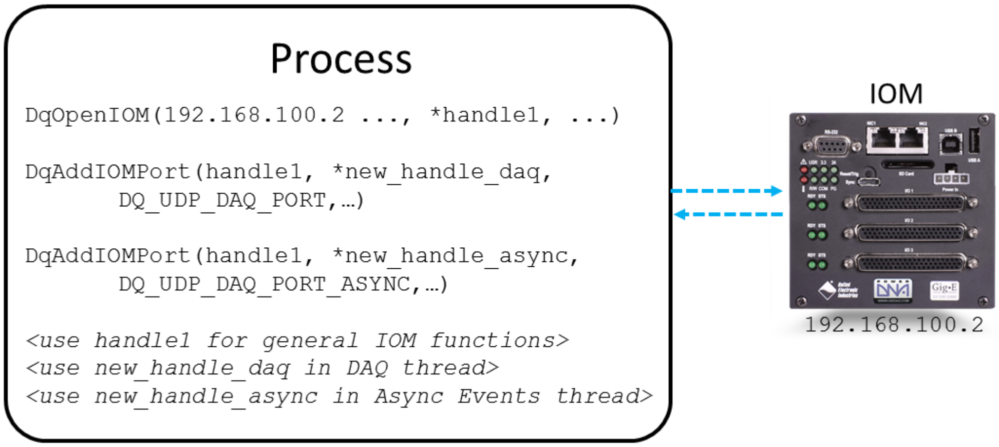
```
DqAddIOMPort(handle, *new_handle, UDP_Port ...)
```

Before adding a new port, the process should already have a `handle` to the IOM from `DqOpenIOM`. When `DqAddIOMPort` is called, the host PC creates a new local UDP port and network buffer. `DqAddIOMPort` sends a packet through `UDP_Port` to pair the new port with a new handle, returned as `new_handle`. The default `UDP_Port` is `DQ_UDP_DAQ_PORT` (6334) for normal DAQ mode threads and `DQ_UDP_DAQ_PORT_ASYNC` (6344) for Asynchronous Event threads. Allocating a dedicated port for the thread makes it safe to command the IOM from multiple threads.

The table below summarizes whether a thread within a multi-threaded process needs to call `DqAddIOMPort`. Threads running asynchronous modes always need to obtain a new handle, no matter if the hardware is in hosted or embedded mode. ACB and legacy DMAP data acquisition modes are inherently thread safe and do not need a new handle, because their UEI DQEngine programming environment automatically monitors the data streams and forwards packets to the appropriate sender/receiver thread.

Does the thread need its own handle?		
DAQ Mode	PowerDNA (host PC)	UEIPAC (embedded)
Point by Point	yes	no
ACB	no	not supported
RtDMap	yes	no
RtVMap	yes	no
ADMap	yes	yes
AVMap	yes	yes
Async Events	yes	yes

Table 1: Whether a thread needs to call `DqAddIOMPort` depends on its DAQ mode and if the application is running in hosted or embedded mode.

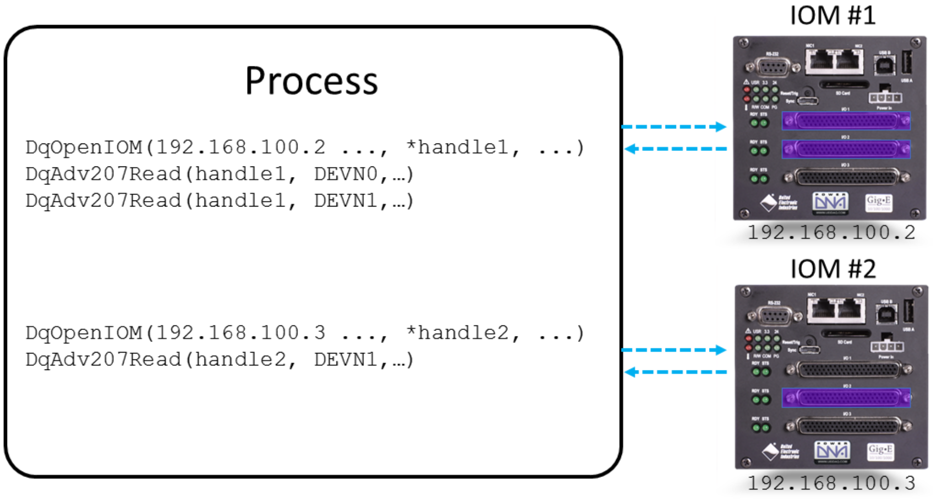


```
DqOpenIOM(192.168.100.2 ..., *handle1, ...)

DqAddIOMPort(handle1, *new_handle_daq,
       DQ_UDP_DAQ_PORT,…)

DqAddIOMPort(handle1, *new_handle_async,
       DQ_UDP_DAQ_PORT_ASYNC,…)

<use handle1 for general IOM functions>
<use new_handle_daq in DAQ thread>
<use new_handle_async in Async Events thread>
```

Figure 9: Two threads within a process obtain unique handles to the IOM through `DqAddIOMPort()`.

2.4. How do I determine a board's Device Number?

Every IO board on an IOM is identified by a device number (DEVN). All function calls to a specific board require the board's DEVN as an input. DEVNs may be determined manually or dynamically.



```
DqOpenIOM(192.168.100.2 ..., *handle1, ...)
DqAdv207Read(handle1, DEVN0,…)
DqAdv207Read(handle1, DEVN1,…)


DqOpenIOM(192.168.100.3 ..., *handle2, ...)
DqAdv207Read(handle2, DEVN1,…)
```

Figure 10: Device numbers identify the boards on an IOM.

Manual DEVN Identification

On a Cube, DEVNs start at 0 on top and increment down the stack of IO boards. On a Rack, DEVNs start at 0 on the board closest to the Power module and skip over empty slots. The slot numbers on a Rack are fixed while device numbers can move.

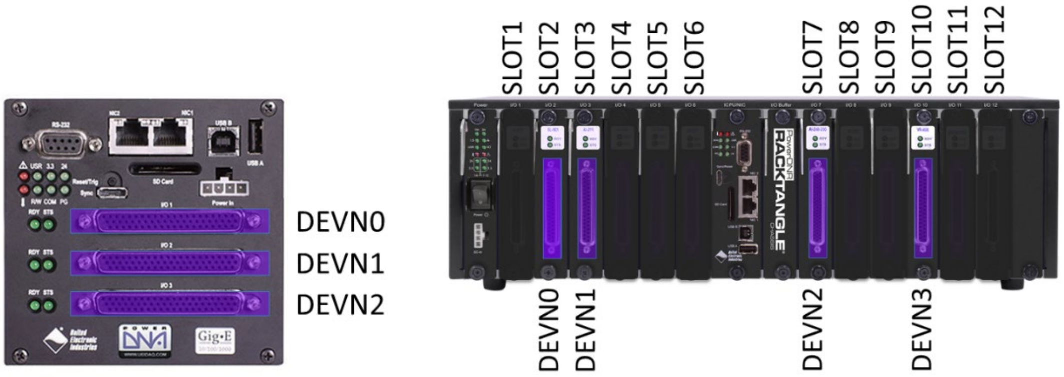


Figure 11: DEVN identification for a Cube (left) and Rack (right)

The DEVNs and slot #s for each board may also be viewed through PowerDNA Explorer.

Dynamic DEVN Identification

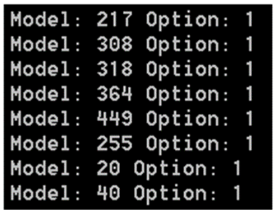
There are two ways to identify the IO boards dynamically through code:

Method 1 - Extract the DEVNs from the `DqRdCfg` structure returned by `DqOpenIOM()`

```
DqOpenIOM(*IP ..., *handle...,*DqRdCfg)
```

`DqRdCfg` includes information such as the IOM's serial number, types of IO boards, calibration dates, and more. The following code prints out the Model # and Option for each board in a Rack chassis. The output of the code is shown on the right.

```
for (i = 0; i < DQ_MAXDEVN; i++) {
     if (DQRdCfg->devmod[i]) {
         printf(" Model: %x Option: %x\n",
     DQRdCfg->devmod[i], DQRdCfg->option[i]);
     } else {
         break;
     }
 }
```



```
Model: 217 Option: 1
Model: 308 Option: 1
Model: 318 Option: 1
Model: 364 Option: 1
Model: 449 Option: 1
Model: 255 Option: 1
Model: 20 Option: 1
Model: 40 Option: 1
```

DEVN0
DEVN1
DEVN2
DEVN3
DEVN4
DEVN5
DNR-Power
CPU-Power

In a Rack, there are two power boards: input power and CPU power. The input power board (DNR-Power) is the furthest left in the chassis while the CPU power is connected to the CPU as a double slot board. The model numbers of these boards are less than 0x100. The Cube only has a single power board, with model number 0x41.

Method 2 - Call `DqGetDevnBySlot()` to return the DEVN of the device sitting in a particular slot.

```
DqGetDevnBySlot(handle, slot, *DEVN,...)
```

`DqGetDevnBySlot` is only useful with the Rack chassis. On Cubes, the slot numbers and device numbers are equivalent.

Set Board to Configuration Mode

Before sending commands to an IO board, it is good practice to use the `ChkOpsMode` macro to verify that the board is in a configurable state. If the board is in Operation mode, the macro resets the board back to Configuration mode, ready to be configured by the application.

```
ChkOpsMode(handle,DEVN,..)
```

2.5. How should I close out the application?

Stop the DAQ

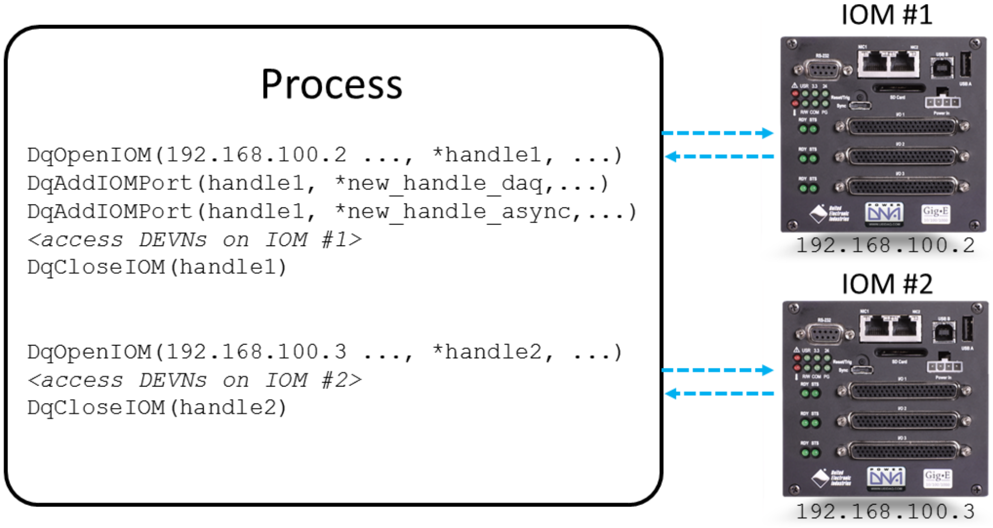
Prior to closing the IOM, ensure that the application properly stops the acquisition modes, leaving the boards in a known and configurable state. In UEI example code, this cleanup is typically performed below the `finish_up` label.

Close the IOM

Call `DqCloseIOM` to close communication with the IOM and deallocate all resources involved.

```
DqCloseIOM(handle)
```

All ports to the IOM created by DqOpenIOM and DqAddIOMPort are closed automatically.



```
DqOpenIOM(192.168.100.2 ..., *handle1, ...)
DqAddIOMPort(handle1, *new_handle_daq,...)
DqAddIOMPort(handle1, *new_handle_async,...)
<access DEVNs on IOM #1>
DqCloseIOM(handle1)


DqOpenIOM(192.168.100.3 ..., *handle2, ...)
<access DEVNs on IOM #2>
DqCloseIOM(handle2)
```

Figure 12: `DqCloseIOM` terminates and cleans up communication to the IOM.

3. Summary

- **Ethernet Ports:** Maximum of 1 PC and 1 Ethernet Port to control the IOM
- **Processes:** Maximum of 1 Process per IO board, call `DqOpenIOM`
- **DAQ Modes:** Maximum of 1 DAQ mode per IO board
- **Threads:** All DAQ modes should call `DqAddIOMPort`, except ACB and UEIPAC
    - Boards of Different Types – dedicated threads
    - Boards @Similar Rates – in the same thread
    - Boards @Different Rates – dedicated threads
    - Messaging Ports @Different Rates – in the same thread, unless using Point-by-Point mode

Underlined text is a rule. Other information constitutes guidelines, rather than rules.

Questions? Please contact UEI support at support@ueidaq.com or call 508.921.4600.