# PowerDNA API Reference Manual
# Release 5.0

September 2, 2022 Edition

# 1 Introduction

This document is intended to serve as a reference guide to those who wish to program a PowerDNA system. PowerDNA is the umbrella name that describes a real-time distributed I/O system with exceptional flexibility and performance. It consists of PowerDNA Cubes and RACKs (also known as I/O Modules, or IOMs) that are distributed throughout a process or large piece of equipment, and a dedicated Ethernet interface card plugged into a host computer. A host system can communicate with IOMs over conventional Ethernet, using copper or fiber optic cables. If hard real-time response is required, the host computer should have a real-time OS running. To achieve hard real-time performance on commercial Ethernet cabling, a Central Controller comes with firmware that implements UEI’s patent-pending DaqBIOS protocol.

Each PowerDNA chassis provides several slots or “layers” that hold a variety of analog and digital I/O function modules. For full details on PowerDAQ hardware, including PowerDNA chassis and the various I/O Layers you can select, please go to www.PowerDNA.com

This document gives further details about the features and functionality of various system components. It also provides an overview of all API functions that a designer employs to create a user application.

In broad terms, a user application uses function calls from the PowerDAQ API shared library. These functions in turn use DaqBIOS commands that run the DaqBIOS Engine, the firmware that allow the host or Central Controller to communicate with the PowerDNA cubes and racks, sending operating commands and retrieving data.

Although it is not always stated, all layers have an option level. The option level appears as a dash and a number after the model number. For instance AI-217-803 is an AI-217 layer with option level 803. If the option level is not shown then one may assume that all option levels are being referenced. The option level for a base model layer is always -1.

# 2 Five ways to communicate with IOM

The PowerDNA API provides five ways of communicating with PowerDNA cubes and racks:

1. DaqBIOS Command API – so-called low-level commands (point-by-point synchronous)
2. Buffered I/O in continuous (asynchronous) – see DqAcb functions
3. Mapped I/O (synchronous) – see DqRtDmap (fixed data size) and DqRtVmap (variable data size) functions
4. Messaging I/O (asynchronous) – see DqMsg functions
5. Async Event / Mapped I/O (event-driven) – see DqRtAsync (event notification, asynchronous) and aDMap/aVMap (clocked, timer, or FIFO watermark event-driven) functions

All five APIs can be used to communicate with a single IOM, but not at the same time. Once your system is switched into one of asynchronous modes, it is recommended that you not issue synchronous commands so as to avoid interfering with the PowerDNA cube or rack configuration and timing set up for asynchronous mode.

## DaqBIOS Command API

The DaqBIOS Command API is a direct copy of DaqBIOS commands to the IOM. Every DaqBIOS command has its equivalent in the PowerDNA (PDNALib) library.

Any code using the PowerDNA Library must include the file, PDNA.h, which itself includes all necessary additional header files.

## 2.1 Pre-defined Types and Error Codes

### 2.1.1 Pre-defined Types

These types are defined for use within the library.

`DAQLIB` is omitted from this document for the sake of readability. Under Microsoft Windows, it is used for functions that are exported from the DAQLib dll for the user.

```
#define DAQLIB __declspec(dllexport) __stdcall
```

The following types keep the API call format similar across wide range of operating systems.

```
typedef unsigned long    u32;
typedef unsigned short   u16;
typedef unsigned char    u8;

typedef unsigned long    uint32;
typedef unsigned short   uint16;
typedef unsigned char    uint8;

typedef long             int32;
typedef short            int16;
typedef char             int8;

typedef unsigned long    DWORD_PTR;
typedef LPTSTR char*;
```

### 2.1.2 Devices and subsystems

DaqBIOS defines a maximum of 16 layers (devices) inside an IOM and eight subsystems per layer.

```
#define DQ_MAXDEVN    16   // sixteen layers on the bus maximum
#define DQ_MAXSS       8   // maximum number of subsystems (four input and four output)
#define DQ_SS0IN       0   // Subsystem 0 input (main and often only device SS)
#define DQ_SS0OUT      1   // Subsystem 0 output (main and often only device SS)
#define DQ_SS1IN       2   // Subsystem 1 input
#define DQ_SS1OUT      3   // Subsystem 1 output
#define DQ_SS2IN       4   // Subsystem 2 input
#define DQ_SS2OUT      5   // Subsystem 2 output
#define DQ_SS3IN       6   // Subsystem 3 input
#define DQ_SS3OUT      7   // Subsystem 3 output
```

The layer number is `uint8`. Only the four lower bits are used to signify the layer number. The upper four bits have a special purpose. One of them is the `DQ_LASTDEV` (0x80) flag, which is used to mark the last command entry in a DaqBIOS packet. The PowerDNA library handles use of `DQ_LASTDEV`, so that a user never has to deal with it.

## *2.1.3 DaqBIOS Packet Structures*

DaqBIOS protocol runs on top of UDP protocol. DaqBIOS packet has the following structure:

```
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 */
} DQPKT, * pDQPKT;
```

The maximum size of a DaqBIOS packet (including this header) is limited to 1472 bytes.

## *2.1.4 Error Codes*

The following error codes are generated in firmware and can be returned from the IOM in the `dqCommand` field of a DaqBIOS packet:

```
/* 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   0x80040000 /* 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 */
#define DQERR_ACCESS    0x90070000 /* password is not cleared - access denied */
#define DQERR_LOCKED    0x90080000 /* cube is locked */
```

```
/*
** The following is reuse of the previous code
** in the 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 */
```

These codes are or-ed with command reply value (`dqCommand | DQREPLY`).

Another set of error codes can be generated on the host side:

```
#define DQ_NOERROR               0     // no error encountered
#define DQ_SUCCESS               1     // success
#define DQ_WAIT_ENDED            2     // asynchronous event or VMap/DMap timed out
#define DQ_DATA_NOTREADY         3     // success,new data has not yet settled on the IOM side
#define DQ_DATA_NOTEXIT          4     // success but new data does not exist
#define DQ_DEV_STARTED           5     // start trigger is received
#define DQ_DEV_STOPPED           6     // stop trigger is received

#define DQ_ILLEGAL_ENTRY        (-1)   // illegal entry in parameters
#define DQ_ILLEGAL_HANDLE       (-2)   // illegal IOM handle (index)
#define DQ_SOCK_LIB_ERROR       (-3)   // socket error
#define DQ_TIMEOUT_ERROR        (-4)   // command returns upon timeout
#define DQ_SEND_ERROR           (-5)   // packet sending error
#define DQ_RECV_ERROR           (-6)   // packet receiving error
#define DQ_IOM_ERROR            (-7)   // IOM reports an unrecoverable error
#define DQ_PKT_TOOLONG          (-8)   // too much data to fit a packet
#define DQ_ILLEGAL_PKTSIZE      (-9)   // packet size too small or too large
#define DQ_INIT_ERROR           (-10)  // IOM initialization error
#define DQ_BAD_PARAMETER        (-11)  // Invalid parameter passed
#define DQ_BAD_DEVN             (-12)  // incorrect DEVN
#define DQ_NOT_IMPLEMENTED      (-13)  // not implemented yet
#define DQ_NO_MEMORY            (-14)  // not enough memory
#define DQ_NOT_ENOUGH_ROOM      (-15)  // not enough room in the packet/structure
#define DQ_DEVICE_BUSY          (-16)  // somebody else uses this device
#define DQ_EVENT_ERROR          (-17)  // event handling error
#define DQ_BAD_CONFIG           (-18)  // bad configuration reported by DQCMD_RDSTS
#define DQ_DATA_ERROR           (-19)  // layer returned invalid data
#define DQ_DEVICE_NOTREADY      (-20)  // device is not ready
#define DQ_CALIBRATION_ERROR    (-21)  // error while performing calibration
#define DQ_WRONG_DMAP           (-22)  // requested and received dmapid's do not match
#define DQ_DATA_NOT_AVAILABLE   (-23)  // requested data is not available
#define DQ_FIFO_OVERFLOW        (-24)  // device FIFO overflowed
#define DQ_ILLEGAL_INDEX        (-25)  // illegal index supplied
#define DQ_DIO_LINE_NOT_EXIST   (-26)  // the specified dio line doesn't exist
#define DQ_WRONG_PKT_COUNTER    (-27)  // incorrect packet counter was received
#define DQ_ASYNC_OUT_REREQST    (-28)  // last asynchronous output packet is re-requested
#define DQ_PROTOCOL_MISMATCH    (-29)  // protocol version mismatch (DAQLib / IOM CPU)
#define DQ_CMD_NOTALLOWED       (-30)  // command is not allowed either because the device
                                       //      is in op or shutdown mode
#define DQ_CMD_ACCESSDENIED     (-31)  // access denied due to invalid authentication or
                                       //      hardware prevention mechanism
#define DQ_DEVLOCKED            (-32)  // access denied due to IOM locked to another host
```

```
#define DQ_BAD_PARAMETER_0    (-200)  // first parameter on command line was invalid
#define DQ_BAD_PARAMETER_1    (-201)  // second parameter on command line was invalid
#define DQ_BAD_PARAMETER_2    (-202)  // third parameter on command line was invalid
Et cetera...

#define DQ_ERR_NEGATIVE_RETURN 0x80000000   // invert return code
```

As a general rule, you should check the return code from function calls for a negative number, which indicates that an error has occurred.

## 2.2 Auxiliary Functions

### 2.2.1 DqTranslateError

**Syntax:**

`char *DqTranslateError(int error)`

**Input:**

| | |
|---|---|
| `int error` | Error code (negative) returned by previous DaqBIOS API call |

**Output:**

None

**Return:**

Pointer to ASCIIZ string describing this error (`const char`)

**Description:**

Converts error code into string

**Note:**

Windows implementation returns `LPTSTR` type

### 2.2.2 DqInitDAQLib

**Syntax:**

`int DqInitDAQLib(void)`

**Input:**

None

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_SOCK_LIB_ERROR` | Windows `WSAStartup()` error |
| `DQ_INIT_ERROR` | Error allocating memory |
| `DQ_SUCCESS` | Command processed successfully |

**Description:**

Loads the socket libraries and initializes table structures

**Notes:**

This function should be called at the beginning of your program to allow the library to allocate required resources and data structures after it is loaded.

### 2.2.3 DqCleanUpDAQLib

**Syntax:**

```
void DqCleanUpDAQLib(void)
```

**Input:**

None

**Output:**

None

**Return:**

None

**Description:**

Closes socket libraries and deallocates table structures.

**Note:**

This function should be called at the end of your program, to allow the library to release resources and clean up its allocated structures before it is unloaded. This function should not be called under Windows, because Windows will automatically call it when unloading the DLL.

### 2.2.4 DqGetLibVersion

**Syntax:**

```
uint32 DqGetLibVersion(void)
```

**Input:**

None

**Output:**

None

**Return:**

Version of PDNA Library

**Description:**

Library version returns as `0xMMNNSS`, where `M` is major version, `N` is minor version and `S` is release subversion.

**Note:**

Versions with the same minors and different subversions have compatible APIs

### 2.2.5 DqOpenIOM

**Syntax:**

```
int  DqOpenIOM(char *IP, uint16 UDP_Port, uint32 mTimeOut, int
*handle, pDQRDCFG *pDqCfg);
```

**Input:**

| | |
|---|---|
| `char *IP` | IP Address in Decimal dot Notation |
| `uint16 UDP_Port` | UDP port for use by the IOM |
| `uint32 mTimeOut` | Timeout Duration in milliseconds |
| `int *handle` | pointer to store descriptor of the IOM being opened |
| `pDQRDCFG *pDqCfg` | pointer to store pointer to `DQCMD_ECHO` results, or NULL if not required |

**Output:**

| | |
|---|---|
| `int *handle` | descriptor of the IOM being opened |
| `pDQRDCFG *pDqCfg` | pointer to stored `DQRDCFG` structure resulting from `DQCMD_ECHO` command or NULL upon error |

**Return:**

| | |
|---|---|
| `DQ_SOCK_LIB_ERROR` | error opening socket or connecting to server |
| `DQ_NO_MEMORY` | memory allocation error or exceeded maximum table size |
| `DQ_IOM_ERROR` | IOM reports command execution error |
| `DQ_SEND_ERROR` | unable to send a packet to the IOM |
| `DQ_TIMEOUT_ERROR` | IOM reply wasn't received within timeout period |
| `DQ_SUCCESS` | if the IOM was successfully opened |

**Description:**

This function opens communications with the IOM. If the IOM is successfully opened, it allows function calls from this process. Normally, this function opens the descriptor of the network interface and sends the `DQCMD_ECHO` command to the specified IP address to retrieve the IOM serial number, layer types, model, calibration dates, etc. Upon retrieval of the `DQRDCFG` information, it stores it in the IOM table. If the `DQCMD_ECHO` command fails, the function returns a NULL in `pDqCfg`. If the `DQRDCFG` information is not desired, pass NULL as `pDqCfg`.

The returning packet payload has the following structure:

```
/* device configuration data */
typedef struct {
    uint32 model;              /* IOM model */
    uint32 ipaddr;             /* ip address */
    uint32 sernum;             /* serial number */
    uint32 caldate;            /* calibration date */
    uint32 mfgdate;            /* manufacturing date */
    uint16 devmod[DQ_MAXDEVN]; /* up to 16 installed layers */
    uint16 option[DQ_MAXDEVN]; /* and their option parameters */
} DQRDCFG, *pDQRDCFG;
```

Calibration and manufacturing dates are represented as `0xMMDDYYYY` in `uint32`, where `MM` is month, `DD` is day of the month and `YYYY` is year.

**Note:**

- If the configuration from the IOM wasn't requested or if the function returned a configuration retrieval error, DQE will attempt to retrieve IOM configuration when `Dq…InitOps()` is called.
- To access the same IOM from multiple threads, please see `DqAddIOMPort()`.

### *2.2.6 DqCloseIOM*

**Syntax:**

`void DqCloseIOM(int Iom)`

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM Descriptor |
| `DQ_SUCCESS` | successful completion |

**Description:**

The function closes communication with the IOM and de-allocates all resources involved.

**Note:**

None

## 2.2.7 DqAddIOMPort

**Syntax:**

```
int DAQLIB DqAddIOMPort(int handle, int* new_handle, uint16 UDP_Port,
uint32 mTimeOut)
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `uint16 UDP_Port` | UDP port used for `DqOpenIOM` call (i.e. `DQ_UDP_DAQ_PORT` or `DQ_UDP_DAQ_PORT_ASYNC`) |
| `uint32 mTimeOut` | milliseconds to wait for the reply packet |

**Output:**

| | |
|---|---|
| `int* new_handle` | descriptor of the newly created IOM handle |

**Return:**

| | |
|---|---|
| `DQ_SOCK_LIB_ERROR` | error opening socket or connecting to server |
| `DQ_NO_MEMORY` | memory allocation error or exceeded maximum table size |
| `DQ_IOM_ERROR` | IOM reports command execution error |
| `DQ_SEND_ERROR` | unable to send a packet to the IOM |
| `DQ_TIMEOUT_ERROR` | IOM reply wasn't received within timeout period |
| `DQ_SUCCESS` | if the IOM was successfully opened |

**Description:**

The `DqAddIOMPort()` function is used if the user needs to access the same IOM from multiple threads when using the following data acquisition modes: Point by Point, Realtime VMap (RTVmap), Realtime DMap (RTDMap), asynchronous VMap/DMap and asynchronous events modes. `DqAddIOMPort` creates a separate handle that is required for a separate command-reply stream. It is particularly useful when multiple asynchronous VMaps/DMaps are used or when the application is written to wait for different events in the different threads. Note that ACB and legacy DMAP data acquisition modes are inherently thread safe and do not need to use `DqAddIOMPort` because they use UEI's DQEngine, a programming environment that takes care of handling streams of data and performing error correction.

DAQLib on the host PC (OS) creates a new local UDP port and network buffer. A command packet is sent to the IOM on `UDP_Port` to pair the local UDP port with a new handle, returned as `new_handle`.

The default UDP_Port for normal IOM communications is DQ_UDP_DAQ_PORT (6334). The default UDP_Port for asynchronous events is DQ_UDP_DAQ_PORT_ASYNC (6344). The port is listed as “udp” through the serial debug console’s “show” command.

When the `new_handle` is used in any Dq… function, the packet is sent to the UDP port specified by the `UDP_Port` parameter. Replies are received on the new local UDP port, which is different than the local port and network buffer allocated by `DqOpenIOM()`. Thus, it becomes safe to call functions from multiple threads without waiting for the IOM lock to reply or risking unexpected replies.

Additional ports are closed automatically when `DqCloseIOM()` is called. Note that `DqAddIOMPort()`only needs to be used on the **UEIPAC** for asynchronous modes (async events, asynchronous VMAP and asynchronous DMAP).

**Notes:**

- Up to 63 additional ports can be opened per IOM; more requests return a `DQ_NO_MEMORY` error.
- `DqAddIOMPort()` is not necessary when using multiple processes. For multiple processes, you must call `DqOpenIOM()`in each process.

### 2.2.8 DqGetDevnBySlot

**Syntax:**

```
int DqGetDevnBySlot(int Iom, uint32 Slot, uint32* devn, uint32*
serial, uint32* address, uint16* model)
```

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint32 Slot` | The DNR chassis slot to query |

**Output:**

| | |
|---|---|
| `uint32* devn` | Device number of the device sitting in the specified slot |
| `uint32* serial` | Serial number of the device sitting in the specified slot |
| `uint32* address` | Address of the device sitting in the specified slot |
| `uint16* model` | Model number of the device sitting in the specified slot |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM Descriptor |
| `DQ_BAD_PARAMETER` | The specified slot is empty or invalid |
| `DQ_SUCCESS` | successful completion |

**Description:**

The function returns the ID number as well as other parameters for the device inserted in a given slot.

**Note:**

This function is only useful with PowerDNR chassis. On PowerDNA I/O modules the device number is always the position of the device in the stack.

## *2.2.9 DqGetDevnBySerial*

**Syntax:**

```
int DqGetDevnBySerial(int Iom, uint32 Serial, uint32* devn,
uint32* slot, uint32* address, uint16* model)
```

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint32 Serial` | The serial number of the device to query |

**Output:**

| | |
|---|---|
| `uint32* devn` | Device number of the device matching the serial number |
| `uint32* slot` | Slot of the device matching the serial number |
| `uint32* address` | Address of the device matching the serial number |
| `uint16* model` | Model number of the device matching the serial number |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM Descriptor |
| `DQ_BAD_PARAMETER` | The specified serial number was not found |
| `DQ_SUCCESS` | successful completion |

**Description:**

The function returns the ID number as well as other parameters for the device with a given serial number.

**Note:**

None

### *2.2.10 DqSetPacketSize*

**Syntax:**

```
int DqSetPacketSize(int Iom, uint32 MinPktSize, uint32
MaxPktSize)
```

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint32 MinPktSize` | minimum size of UDP packet to send[1] |
| `uint32 MaxPktSize` | maximum size of UDP packet to send[2] |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid Descriptor |
| `DQ_ILLEGAL_PKTSIZE` | invalid value for `MinPktSize` or `MaxPktSize` |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_SUCCESS` | successful completion |

**Description:**

The function sets up the minimum and maximum allowed sizes of UDP DaqBIOS packets.

**Note:**

For non-fragmented DaqBIOS packets, the maximum payload size (including the DQPKT header) is limited to 1472 bytes.

For legacy systems using the ColdFire processor, the maximum size of a DaqBIOS packet (including the DQPKT header) is limited to 530 bytes.

### *2.2.11 DqSetTimeout*

**Syntax:**

```
int DqSetTimeout(int Iom, pDQE pDqe, int Timeoutms)
```

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM |
| `pDQE pDqe` | DQ Engine pointer, or NULL if not using DQE |
| `int Timeoutms` | timeout value in milliseconds |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid Descriptor |
| `DQ_BAD_PARAMETER` | `Timeoutms` is negative |
| `DQ_SUCCESS` | command processed successfully |

---

[1] Setting minimum size is possibly not useful.

[2] Limit maximum size for faster response.

**Description:**

Changes the maximum allowed wait time for a reply packet (IOM must be open).

**Note:**

If (`pDqe` != NULL), the function sets the total timeout for every DQ command handled by DQE. To calculate reply timeout, the total timeout is divided by number of retries allowed. If `pDqe` is not specified, the function sets the timeout that applies only to the specified IOM.

### 2.2.12 DqReadSrec

**Syntax:**

```
int DqReadSrec(char *filename, int relative, uint32 size, char
*buffer, uint32 *bytes)
```

**Command:**

DAQLib

**Input:**

| | |
|---|---|
| `char *filename` | file name of .S19 S-Record file |
| `int relative` | TRUE to honor addresses read from the file, FALSE to read data sequentially and ignore addresses |
| `uint32 size` | size of user buffer |
| `char *buffer` | pointer to user buffer |
| `uint32 *bytes` | pointer to receive number of bytes written to `buffer` |

**Output:**

| | |
|---|---|
| `char *buffer` | raw data from .S19 S-Record file |
| `uint32 *bytes` | number of bytes written to `buffer` |

**Returns:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | if any parameter is NULL |
| `DQ_SUCCESS` | successful completion |

**Description:**

Reads S-Record (.S19) file and converts it to binary data.

If `relative` is `TRUE`: This function considers the first address it encounters as the beginning of the buffer, and all other addresses relative to the beginning of the buffer. Any addresses outside the range the buffer can hold will be silently dropped.

If `relative` is `FALSE`: This function will read all data sequentially into the buffer, despite the addresses indicated in the S-Records.

**Note:**

None

### *2.2.13 DqGetLastStatus*

**Syntax:**

```
int DqGetLastStatus(int iom, uint32 *data, uint32 *size)
```

**Input:**

| | |
|---|---|
| `int iom` | Handle to the IOM |
| `uint32 *data` | buffer to copy status data into |
| `uint32 *size` | size of buffer (number of 32-bit values) |

**Output:**

| | |
|---|---|
| `uint32 *size` | returns amount of data copied into `data` parameter, in 32-bit chunks |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor |
| `DQ_BAD_PARAMETER` | `data` or `size` parameter was NULL |
| `DQ_NOT_ENOUGH_ROOM` | size passed in is too small to hold the data |
| `DQ_RECV_ERROR` | no status data has been received yet |
| `DQ_SUCCESS` | successful completion |

**Description:**

Gets the last status data retrieved via the heartbeat.

**Note:**

None.

### *2.2.14 DqReadAIChannel*

**Syntax:**

```
int DqReadAIChannel(int hd, int devn, int samplesz, uint32
CLSize, uint32 *cl, uint8 *data)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM |
| `int devn` | device number |
| `int samplesz` | size of one sample, in bytes |
| `uint32 CLSize` | channel list size |
| `uint32 *cl` | channel list |
| `uint8 *data` | array to store data (should be of `CL` size times sample size) |

**Output:**

| | |
|---|---|
| `uint8 *data` | received data |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function reads data from an analog input device.

**Note:**

Include `DQ_INSTMODE_NTOH` flag in `samplesz` parameter to automatically convert data from network byte order to host byte order.

### 2.2.15 DqWriteAOChannel

**Syntax:**

```
int DqWriteAOChannel(int hd, int devn, int samplesz, uint32
CLSize, uint32* cl, uint8* data)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM |
| `int devn` | device number |
| `int samplesz` | size of one sample, in bytes |
| `uint32 CLSize` | channel list size |
| `uint32 *cl` | channel list |
| `uint8 *data` | array of data (should be of `CL` size times sample size) |

**Output:**

| | |
|---|---|
| `uint8 *data` | received data |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function writes data to an analog output device.

**Note:**

Include `DQ_INSTMODE_NTOH` flag in `samplesz` parameter to automatically convert data from host byte order to network byte order.

### *2.2.16 DqCmdEcho*

**Syntax:**

`int DqCmdEcho(int Iom, pDQRDCFG pDQRdCfg)`

**Command:**

`DQCMD_ECHO` (0x104), Echo

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQRDCFG pDQRdCfg` | pointer to `DQRDCFG` structure allocated by calling process |

**Output:**

| | |
|---|---|
| `pDQRDCFG pDQRdCfg` | pointer to populated `DQRDCFG` structure allocated by calling process |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_IOM_ERROR` | IOM reports command execution error |
| `DQ_SEND_ERROR` | cannot send packet |
| `DQ_TIMEOUT_ERROR` | IOM reply wasn't received within timeout period |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

Specified IOM replies back to the host with configuration data.

The returning packet payload has the following structure:

```
/* device configuration data */
typedef struct {
    uint32 model;              /* IOM model */
    uint32 ipaddr;             /* ip address */
    uint32 sernum;             /* serial number */
    uint32 caldate;            /* calibration date */
    uint32 mfgdate;            /* manufacturing date */
    uint16 devmod[DQ_MAXDEVN]; /* up to 16 installed layers */
    uint16 option[DQ_MAXDEVN]; /* and their option parameters */
} DQRDCFG, *pDQRDCFG;
```

Calibration and manufacturing dates are represented as `0xMMDDYYYY` in uint32, where `MM` is month, `DD` is day of the month and `YYYY` is year.

**Note:**

This is a safe command in any mode.

### *2.2.17 DqCmdCheckAlive*

**Syntax:**

`int DqCmdCheckAlive(int Iom, uint32* response)`

**Command:**

`DQCMD_HRTBEAT` (0x101), Heartbeat

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint32* response` | Pointer to content of scratchpad 0 register |

**Output:**

| | |
|---|---|
| `uint32* response` | Content of scratchpad register: 0 on first API call after bootup; subsequent calls return the CPU timestamp |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_IOM_ERROR` | IOM reports command execution error |
| `DQ_SEND_ERROR` | cannot send packet |
| `DQ_TIMEOUT_ERROR` | IOM reply wasn't received within timeout period |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This is a lowest-CPU-load command possible for heartbeat service to verify that the chassis is alive. The API performs the following functions:

1. Reads current timestamp from CPU board
2. Retrieves contents from scratchpad register 0 (which is initialized as 0 upon reboot), and returns value retrieved as `response`
3. Stores timestamp read in step 1 into scratchpad 0 register

**Note:**

The `DqCmdWriteVal()` API can also be used to write the scratchpad 0 register (0xA00E0010 in PPC): do not use `DqCmdCheckAlive`() in conjunction with directly writing scratchpad 0 using the `DqCmdWriteVal()` API.

### *2.2.18 DqCheckForCriticalError*

**Syntax:**

`int DqCheckForCriticalError(int Iom, int error, uint32 flags, uint32* criticality)`

**Command:**

`DQCMD_HRTBEAT` (0x101), Heartbeat

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `int error` | Error flag received from Dq..() call |
| `uint32 flags` | DQ_CHECK_ALIVE to ping the cube on error, 0 - simply check |

**Output:**

| | |
|---|---|
| `uint32* criticality` | how critical the error is (or NULL to ignore) |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_IOM_ERROR` | IOM reports command execution error |
| `DQ_SEND_ERROR` | cannot send packet |
| `DQ_TIMEOUT_ERROR` | IOM reply wasn't received within timeout period |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

When timeout occurs it is not a critical error (network may lose a packet now and then). Two timeouts in a row signal about more severe error.

DQ_TIMEOUT_ERROR could happen in three cases:

1. Normal behavior of the network - a request or response packet is lost due to the transmission error (since Ethernet protocol does not have acknowledge feature built into it the transmitter doesn't know that the packet was lost unless there is a babbling on the line). This is rare and not critical error

2. Misconfigured switch. For example, packet storm protection on some Cisco switches is set to remove similar packets to the same port if they come in less than predefined milliseconds apart. In this case I normally log into the switch and disable all advanced traffic monitoring features. This is not critical error

3. Network is losing packets in droves or dead or misconfigured or cube firmware became irresponsive. This is critical error and software needs to know about it immediately

In cases (1) and (2) network loses a single packet very infrequently. In (3) requests are not replied in blocks (in worst case - until cube is reset).

If the flag DQ_CHECK_ALIVE is set `DqCheckForCriticalError()`pings IOM with `DqCmdCheckAlive()` to see whether IOM is responding to the requests.

The function returns actual <error> for any criticality level above DQ_CHK_ERROR_IGNORE.

Criticality levels are:
DQ_CHK_ERROR_NONE == 0 no error
DQ_CHK_ERROR_IGNORE == 1 upon a single timeout, function returns non-negative number
DQ_CHK_ERROR_NONCRITICAL == 2 error in function call parameters (and an actual error code)
DQ_CHK_ERROR_CRITICAL == 3 execution cannot continue (and an actual error code)

The following error codes are deemed non-critical (i.e. execution can continue, but parameters needs to be adjusted):

```
DQ_ILLEGAL_ENTRY:          // illegal entry in parameters
DQ_BAD_DEVN:               // incorrect DEVN
DQ_PKT_TOOLONG:            // too much data to fit a packet
DQ_INIT_ERROR:             // IOM initialization error
DQ_BAD_PARAMETER:          // Invalid parameter passed
DQ_NO_MEMORY:              // not enough memory
DQ_NOT_ENOUGH_ROOM:        // not enough room in the packet/structure
DQ_DEVICE_BUSY:            // somebody else uses this device
```

```
DQ_ILLEGAL_PKTSIZE:       // packet size too small or too large
DQ_NOT_IMPLEMENTED:       // not implemented yet
DQ_DEVICE_NOTREADY:       // device is not ready
DQ_CALIBRATION_ERROR:     // error while performing calibration
DQ_WRONG_DMAP:            // requested and received dmapid's do not match
DQ_DATA_NOT_AVAILABLE:    // requested data is not available
DQ_FIFO_OVERFLOW:         // device FIFO overflowed
DQ_ILLEGAL_INDEX:         // illegal index supplied
DQ_DIO_LINE_NOT_EXIST:    // the specified dio line doesn't exist
DQ_WRONG_PKT_COUNTER:     // incorrect packet counter was received
DQ_EVENT_ERROR:           // event handling error
DQ_BAD_CONFIG:            // bad configuration reported by DQCMD_RDSTS
DQ_DATA_ERROR:            // layer returned invalid data
```

Critical errors are:

```
DQ_ILLEGAL_HANDLE:        // illegal IOM handle (index)
DQ_SOCK_LIB_ERROR:        // socket error
DQ_SEND_ERROR:            // packet sending error
DQ_RECV_ERROR:            // packet receiving error
DQ_IOM_ERROR:             // IOM reports an unrecoverable error
```

### *2.2.19 DqCmdReset*

**Syntax:**

`int DqCmdReset(int Iom)`

**Command:**

`DQCMD_RST` (0x110), Reset Device

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | cannot send packet |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

Resets the device specified. After this command, a device considers itself "initialized" and switches to Init mode.

**Note:**

This command cannot be called in Operation mode. A device does not return from reset immediately. Device becomes unavailable for network operations until reboot (approximately 5-10 seconds.)

### *2.2.20 DqCmdHwReset*

**Syntax:**

`int DqCmdHwReset(int Iom)`

**Command:**

`DQCMD_RST` (0x110), Reset Device

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | cannot send packet |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

Performs full hardware reset on the IOM specified. After this command, the IOM considers itself "initialized" and switches to Init mode.

**Note:**

This command cannot be called in Operation mode. A device does not return from reset immediately. A device becomes unavailable for network operation until reboot occurs (approximately 5-10 seconds.)

### *2.2.21 DqCmdWriteVal*

**Syntax:**

`int DqCmdWriteVal(int Iom, uint32 Address, uint32 *Value)`

**Command:**

`DQCMD_WRVAL` (0x114), Write Value to the Device

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint32 Address` | IOM memory map address to write to |
| `uint32 *Value` | pointer to 32-bit value to write to `Address` |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function writes `Value` to a physical `Address` on IOM.

**Note:**

- This low-level function can be dangerous because all IOM address space, including layer registers, is accessible.
- This function is password-protected with user level password.
- If using `DqCmdWriteVal()`to write the scratchpad 0 register (0xA00E0010 in PPC), do not use `DqCmdCheckAlive`(), which also accesses scratchpad register 0.

### *2.2.22 DqCmdReadVal*

**Syntax:**

`int DqCmdReadVal(int Iom, uint32 Address, uint32 *Value)`

**Command:**

`DQCMD_RDVAL` (0x118), Read Value from the Device

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint32 Address` | IOM memory map address to read from |
| `uint32 *Value` | pointer to buffer to store the value received from the device |

**Output:**

| | |
|---|---|
| `uint32 *Value` | value received from the device |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

Function reads `Value` from a physical `Address` on IOM.

**Note:**

This low-level function can be dangerous because all IOM address space is accessible. Therefore, I/O layer triggers may be initiated when certain memory is read.

This function is password-protected with user level password

### *2.2.23 DqCmdWriteRead32*

**Syntax:**

```
int DqCmdWriteRead32(int Iom, pDQWRRD32 wrrdcmd, uint32*
data_wr, uint32* data_rd)
```

**Command:**

`DQCMD_WRRD32` (0x115), Write and read values to the layer

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQWRRD32 wrrdcmd` | device and address/size information to write/read data |
| `uint32* data_wr` | data to be written if wrrdcmd->size_w > 0 |
| `uint32* data_rd` | data read from the device if wrrdcmd->size_r > 0 |

**Output:**

| | |
|---|---|
| `pDQWRRD32 +`<br>`uint32* data_rd` | value received from the device |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

Performs a write followed by a read (or read followed by a write) into the 32-bit address space of the selected layer. This function is safer to use than DqCmdReadVal, DqCmdReadMultipleValues, DqCmdWriteVal, or DqCmdWriteMultipleValues because it directly addresses the layer and reduces the ability to accidentally read or write into critical system areas of the IOM.

This structure is used to read and write multiple addresses from and to IOM memory:

```
/* DQCMD_WRRD32 */
#define DQWRRD32_DONOT_INC  (1L<<1) // do not increment address, useful to read FIFOs
#define DQWRRD32_READ_1ST   (1L<<0) // first read then write, default is the opposite

typedef struct {
    uint16 dev;         // device DEVN to access
    uint16 read_1st;    // read before write if TRUE
    uint16 addr_w;      // address to write
    uint16 size_w;      // size to write (in uint32s). Must be lower than 368.
    uint16 addr_r;      // address to read
    uint16 size_r;      // size to read (in uint32s). Must be lower than 368.
    uint32 data[];      // data
} DQWRRD32, *pDQWRRD32;
```

**Note:**

<read_1st> field can contain two behavior modificators:
#define DQWRRD32_DONOT_INC (1L<<1)
#define DQWRRD32_READ_1ST (1L<<0)
If read or write is not required in this function call simply pass zero as <size_rd> or <size_wr>.

### 2.2.24 *DqCmdWriteMultipleValues*

**Syntax:**

`int DqCmdWriteMultipleValues(int Iom, pDQWRVALM DQWrMultVal)`

**Command:**

`DQCMD_WRVALM` (0x11C), Write Multiple Values to the Device

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQWRVALM DQWrMultVal` | pointer to the structure that defines what and where to write |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function writes defined values to defined addresses in `DQWrMultVal`.

You have to fill all fields of this structure to ensure correct operation.

This structure is used to read and write multiple addresses from and to IOM memory:

```
/* DQCMD_WRVALM structure */
typedef struct {
    uint32  addr;      // address to write value
    int16   increment; // address increment/decrement after each write
    uint8   size;      // size of operand and four lower bits of increment
    uint8   count;     // number of values to write
    uint8   data[];
} DQWRVALM, *pDQWRVALM;
```

**Note:**

The maximum count is limited by the packet size. This low-level function can be dangerous to use because all IOM address space is accessible. I/O layer triggers may be initiated when certain memory is written.

This function is password-protected with user level password.

### 2.2.25 DqCmdReadMultipleValues

**Syntax:**

```
int DqCmdReadMultipleValues(int Iom, pwDQRDVALM pDQRdMultVal,
uint8 *Data)
```

**Command:**

`DQCMD_RDVALM` (0x120), Read Multiple Values from the Device

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pwDQRDVALM pDQRdMultVal` | pointer to the structure defining what and where to read |
| `uint8 *Data` | pointer to store data received from the PowerDNA cube |

**Output:**

| | |
|---|---|
| `uint8 *Data` | data received from the PowerDNA cube |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function reads multiple values from a physical address on IOM. You have to initialize `pwDQRDVALM` structure before calling it.

```
/* DQCMD_RDVALM structure */
typedef struct {
    uint32  addr;
    int16   increment;
    uint8   size;
    uint8   count;
} wDQRDVALM, *pwDQRDVALM;
```

**Note:**

The maximum count is limited by the packet size. This low-level function can be dangerous to use because all IOM address space is accessible. I/O layer triggers may be initiated when certain memory is read.

This function is password-protected with user level password

### 2.2.26 DqCmdSetCfg

**Syntax:**

```
int DqCmdSetCfg(int Iom, DQSETCFG pDQSetCfg[], uint32 *Status,
uint32 *entries)
```

**Command:**

`DQCMD_SETCFG` (0x124), Set Configuration for the Device.

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `DQSETCFG pDQSetCfg[]` | Array of configuration structures |
| `uint32 *Status` | Array to store status information from all configured devices |
| `uint32 *entries` | Number of configuration structures passed in `pDQSetCfg`. |

**Output:**

| | |
|---|---|
| `uint32 *Status` | Array of status values for the devices |
| `uint32 *entries` | Returns number of entries actually processed |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function sets up configuration of the PowerDNA cube. You must supply an array of `DQSETCFG` structures.

The following structure is used in `DQCMD_SETCFG` command to set up configuration of the layer. Multiple structures in the packet are allowed. The last device number must be or-ed with `LASTDEV` (0x80).

```
/* DQCMD_SETCFG */
typedef struct {
    uint8 dev;  // device
    uint8 ss;   // subsystems
    uint32 cfg;
} DQSETCFG, *pDQSETCFG;
```

Configuration flags are divided into a standard part (common for all layer types) and a layer-specific part. Common flags are:

```
// Standard part (lower 16 bits) of layer configuration word
//
#define DQ_LN_RAW32      (1L<<18)  // copy timestamp along with the data
#define DQ_LN_MAPPED     (1L<<15)  // For WRRD (DQDMAP) devices
#define DQ_LN_STREAMING  (1L<<14)  // For RDFIFO devices - stream the FIFO data automatically
                                   // For WRFIFO - do NOT send reply to WRFIFO unless needed
#define DQ_LN_RECYCLE    (1L<<13)  // if there is no data taken/available overwrite/reuse data
#define DQ_LN_GETRAW     (1L<<12)  // force layer to return raw unconverted data
#define DQ_LN_TMREN      (1L<<11)  // enable layer periodic timer
#define DQ_LN_IRQEN      (1L<<10)  // enable layer irqs
#define DQ_LN_PTRIGEDGE1 (1L<<9)   // stop trigger edge MSB
#define DQ_LN_PTRIGEDGE0 (1L<<8)   // stop trigger edge:  00 - software, 10 - external
#define DQ_LN_STRIGEDGE1 (1L<<7)   // start trigger edge MSB
#define DQ_LN_STRIGEDGE0 (1L<<6)   // start trigger edge: 00 - software, 10 - external
#define DQ_LN_CVCKSRC1   (1L<<5)   // CV clock source MSB
#define DQ_LN_CVCKSRC0   (1L<<4)   // CV clock source 0 - SW, 01 - internal, 10 - exterrnal
#define DQ_LN_CLCKSRC1   (1L<<3)   // CL clock source MSB
#define DQ_LN_CLCKSRC0   (1L<<2)   // CL clock source 0 - SW, 01 - internal, 10 - external
#define DQ_LN_ACTIVE     (1L<<1)   // "ACT" LED status
#define DQ_LN_ENABLED    (1L<<0)   // enable operations
```

Please refer to the layer-specific section in the user manual for layer-specific configuration flags.

**Note:**

None

### *2.2.27 DqCmdReadStatus*

**Syntax:**

`int DqCmdReadStatus(int Iom, uint8 *DeviceNum, uint32 *Entries, uint32 *Status, uint32 *StatusSize)`

**Command:**

`DQCMD_RDSTS` (0x118), Read Device Status

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint8 *DeviceNum` | Array of layer numbers to retrieve status from |
| `uint32 *Entries` | Number of entries in `DeviceNum` array |
| `uint32 *Status` | Pointer to buffer to store values received from the device |
| `uint32 *StatusSize` | Size of buffer, in 32-bit chunks. |

**Output:**

| | |
|---|---|
| `uint32 *Entries` | Actual number of entries requested |
| `uint32 *Status` | Array of device status requested |
| `uint32 *StatusSize` | Number of 32-bit values copied into `Status` |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_NOT_ENOUGH_ROOM` | `StatusSize` is too small to hold the data |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function returns 128 bits of status *per subsystem* for each device specified. The following constants are defined for analyzing the returned status: ( refer to powerdna.h for greater detail )

The 4th uint32 (STS_LOGIC) contains the layer's logic status (not implemented for logic below version 10.00):

```
#define STS_LOGIC_DC_OOR          (1UL<<0)    // DC/DC out of range (IOM also)
#define STS_LOGIC_DC_FAILED       (1UL<<1)    // DC/DC failed (IOM also)
#define STS_LOGIC_TRIG_START      (1UL<<2)    // Trigger event started (IOM also)
#define STS_LOGIC_TRIG_STOP       (1UL<<3)    // Trigger event stopped (IOM also)
#define STS_LOGIC_CLO_NOT_RUNNING (1UL<<4)    // Output channel list not running
#define STS_LOGIC_CLI_NOT_RUNNING (1UL<<5)    // Input channel list not running
#define STS_LOGIC_CVCLK_CLO_ERR   (1UL<<6)    // CV clock error for CLO
#define STS_LOGIC_CVCLK_CLI_ERR   (1UL<<7)    // CV clock error for CLI
#define STS_LOGIC_CLCLK_CLO_ERR   (1UL<<8)    // CL clock error for CLO
#define STS_LOGIC_CLCLK_CLI_ERR   (1UL<<9)    // CL clock error for CLI
```

The first four error flags, `DC_OOR`, `DC_FAILED`, `TRIG_START`, and `TRIG_STOP`, are used in the IOM status as well.

The 3rd uint32 (STS_FW)contains the firmware status:

```
#define STS_FW_CLK_OOR           (1UL<<0)    // Clock out of range (IOM also)
#define STS_FW_SYNC_ERR          (1UL<<1)    // Synchronization interface error (IOM also)
#define STS_FW_CHNL_ERR          (1UL<<2)    // Channel list is incorrect
#define STS_FW_BUF_SCANS_PER_INT (1UL<<3)    // Buf setting error: scans/interrupt
#define STS_FW_BUF_SAMPS_PER_PKT (1UL<<4)    // Buf setting error: samples/packet
#define STS_FW_BUF_RING_SZ       (1UL<<5)    // Buf setting error: FW buffer ring size
#define STS_FW_BUF_PREFUF_SZ     (1UL<<6)    // Buf setting error: Pre-buffering size
#define STS_FW_BAD_CONFIG        (1UL<<7)    // Layer cannot operate in current config
#define STS_FW_BUF_OVER          (1UL<<8)    // Firmware buffer overrun
#define STS_FW_BUF_UNDER         (1UL<<9)    // Firmware buffer underrun
#define STS_FW_LYR_FIFO_OVER     (1UL<<10)   // Layer FIFO overrun
#define STS_FW_LYR_FIFO_UNDER    (1UL<<11)   // Layer FIFO underrun
#define STS_FW_EEPROM_FAIL       (1UL<<12)   // Layer EEPROM failed
#define STS_FW_GENERAL_FAIL      (1UL<<13)   // Layer general failure
#define STS_FW_OPER_MODE         (1UL<<14)   // Layer is in operation mode
#define STS_FW_FIR_GAIN_ERR      (1UL<<15)   // Sum of fir coeffs is not correct
#define STS_FW_OUT_FAIL          (1UL<<16)   // Output CB tripped or over-current
#define STS_FW_IO_FAIL           (1UL<<17)   // Messaging I/O failed (5xx layers)
#define STS_FW_NO_MEMORY         (1UL<<18)   // Error with memory allocation
#define STS_FW_BAD_OPER          (1UL<<19)   // Operation wasn't performed properly
#define STS_FW_LAYER_ERR         (1UL<<20)   // Layer returned unexpected value
#define STS_FW_OVERLOAD          (1UL<<21)   // CPU or Ethernet is starved out of time or told to free
the wire
```

Clock out of range (`STS_FW_CLK_OOR`) means that the call to the DqCmdSetClock() function specified a clock rate that is not possible to achieve with that layer, taking channel/gain list into consideration. For example, an AI-207 layer has a multiplexed front-end with a maximum aggregate sampling rate of 16kS/s using the gain of 1. Thus, if a user specifies a sampling rate 2kS/s for 16 channels, this bit is going to be set (also, DqDmapInitOps() or else DqAcbInitOps() will return an error).

A synchronization interface error (`STS_FW_SYNC_ERR)` indicates that the Sync interface was not configured in a fashion that can provide proper clock signals.

A channel list incorrect error (`STS_FW_CHNL_ERR`) is set when one or more entries in the channel list array supplied to the layer is incorrect for that particular layer.

The flags `STS_FW_BUF_SCANS_PER_INT`, `STS_FW_BUF_SAMPS_PER_PKT`, `STS_FW_BUF_RING_SZ`, `STS_FW_BUF_PREFUF_SZ` set in ACB or Burst mode signify that the requested buffer settings cannot be accepted in the current configuration. The reason may be an incorrect parameter for that particular layer or an inability to allocate a proper buffer or function compatible with other layers in the stack.

The flag `STS_FW_BAD_CONFIG` is set when a layer configuration is improper for this layer or a combination of layers.

The flags `STS_FW_BUF_OVER` and `STS_FW_BUF_UNDER` are set for input and output subsystems in situations in which an internal buffer becomes full and there is no room to store more data (and one or more samples are lost) or when an output buffer becomes empty (output stopped). These flags make sense in ACB mode only.

The flags `STS_FW_LYR_FIFO_OVER` and `STS_FW_LYR_FIFO_UNDER` are similar to previous flags, but report underrun/overrun status of FIFO circuitry located on each layer.

The flag `STS_FW_EEPROM_FAIL` is set when the layer $E^2$PROM 32-bit cyclic redundancy code (CRC32) calculated during the initialization stage does not match the CRC32 stored in the last four bytes of the 2048-byte $E^2$PROM chip.

The flag `STS_FW_GENERAL_FAIL` reports a general layer failure of unknown source detected by the firmware.

The flag `STS_FW_OPER_MODE` is set when a layer is in operating mode. The layer enters operating mode for DMap, ACB, Msg, M3, and Burst operations.

The first two error flags, `CLK_OOR` and `SYNC_ERR`, are used in the IOM status as well.

The 2nd uint32 of the *IOM status* (STS_POST) *only* contains flags that indicate errors discovered during POST (not implemented for firmware revisions 3.4 and below):

```
#define STS_POST_MEMERR        (1L<<0)    // Memory test failed
#define STS_POST_EEPROM_CHK    (1L<<1)    // E2PROM read failed
#define STS_POST_LAYER_FAILED  (1L<<2)    // Layer failure
#define STS_POST_FLASH_FAILED  (1L<<3)    // Flash checksum error
#define STS_POST_SDCARD_FAILED (1L<<4)    // SD Card is not present
#define STS_POST_DC24          (1L<<5)    // DC->24 layer failed
#define STS_POST_DCCORE        (1L<<6)    // Core voltages problem
```

Note that when either the `EEPROM_CHK` or `LAYER_FAILED` flag is set, the `STS_FW_EEPROM_FAIL` or `STS_FW_GENERAL_FAIL` flag, respectively, will also be set in the appropriate layer's status.

The 1st uint32 of the *IOM status* (STS_STATE) contains the operating mode of the layer and the 8 least significant bits of the timestamp counter. The operating mode is encoded into bits 3..0 per the following defines:

#define DQ_IOMODE_INIT (1L) // device is being initialized
#define DQ_IOMODE_CFG (2L) // device is in configuration mode, pt. by pt.
#define DQ_IOMODE_OPS (4L) // device is in operation mode, ACB, DMAP, etc
#define DQ_IOMODE_SD (8L) // device is in shutdown mode

Bits 15..8 of STS_STATE reflect the value of the timestamp counter when the `DqCmdReadStatus` command was performed. This is used to verify that the timestamp system is 'alive', as we expect the value to be constantly changing.

**Note:**

1. Use device |= 0x80 to indicate that this is the last device in the list
2. IOM reads the status registers of the specified devices only
3. The DqVT packet can contain more entries, but the number of devices per IOM is limited to 8.
4. Use DeviceNum entry of 0x7F to get the status of the IOM itself
5. Use DeviceNum entry of 0x7E to get the status of IOM followed sequentially by all devices present; this must be the only value used, and the result is always 128 bits per device, regardless of subsystem count. Each error bit for a device represents the aggregate for all subsystems of that device; in other words, an error bit will be set if at least one subsystem has that error bit set. You can then call `DqCmdReadStatus` with the specific device number to find out which subsystem had the error.

### 2.2.28 DqCmdWriteChannel

**Syntax:**

```
int DqCmdWriteChannel(int Iom, pDQWRCHNL DQChannelData, uint32
DataBufLen)
```

**Command:**

`DQCMD_WRCHNL` (0x12C), Write Channel

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQWRCHNL DQChannelData` | Array of structures that define device, subsystem, channel, and data |
| `uint32 DataBufLen` | Combined size in bytes of all structures contained in `DQChannelData` |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| Other values | low level IOM status |

**Description:**

This function writes channels specified in `DQWRCHNL` structures.

The IOM firmware scans incoming data and passes subsystem and channel information to the appropriate device driver.

The `data` field should be typecast to a data size appropriate to the device. For example, for an AO-302, the `data` field should be typecast to `uint16`. See user manual for a complete list.

`DQCMD_WRCHNL` transfers data in the following format:

```
/* DQCMD_WRCHNL */
typedef struct {
    uint8 dev;
    uint8 ss;
    uint8 chnl;
    uint32 data;
} DQWRCHNL, *pDQWRCHNL;
```

**Note:**

Use `dev |= DQ_LASTDEV` to indicate the last device in the list.

A channel may be a relative position in the programmed channel list but not an actual channel number, depending on the hardware.

Since DMap mode is provided, there is only marginal use for this function,.

### 2.2.29 DqCmdReadChannel

**Syntax:**

`int DqCmdReadChannel(int Iom, pDQRDCHNL pDQChannelData, void *Data, uint32 *DataBufLen)`

**Command:**

`DQCMD_RDCHNL` (0x130), Set Channel List

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQRDCHNL DQChannelData` | Array of structures that defines device and channel |
| `void *Data` | pointer to buffer to store received data |
| `uint32 *DataBufLen` | size of the data buffer, bytes |

**Output:**

| | |
|---|---|
| `void *Data` | received data |
| `uint32 *DataBufLen` | returned size of the data in buffer, bytes |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Timeout duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function reads channels specified in `DQRDCHNL` structures.

The IOM firmware scans incoming data and passes subsystem and channel information to the appropriate device driver. Firmware relies on user to set up `DQ_LASTDEV` for the last device structure in the packet and proper size of data (accordingly to the layer type). For example, an AI-201 returns `uint16` for every channel requested, and an AI-205 and an AI-225 return `uint32`. See user manual for a complete list.

Neither firmware nor the library performs network-to-host data conversion. The user is responsible for converting data according to the layer types involved.

`DQCMD_RDCHNL` transfers data in the following format:

```
/* DQCMD_RDCHNL */
typedef struct {
    uint8 dev;
    uint8 ss;
    uint8 chnl;
} DQRDCHNL, *pDQRDCHNL;
```

**Note:**

Use device |= 0x80 to indicate that this is the last device in the list.

Since DMap mode is provided, there is only marginal use for this function. This function interface is prone to user errors.

### *2.2.30 DqCmdSetClock*

**Syntax:**

```
int DqCmdSetClock(int Iom, pDQSETCLK pDQSetClk, float
*CloseFreq, uint32 *entries)
```

**Command:**

`DQCMD_SETCLK` (0x134), Set Clock Rate

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQSETCLK pDQSetClk` | clock settings (pointer to an array of structures) |
| `float *CloseFreq` | pointer to uninitialized variable (user allocates buffer) |
| `uint32 *entries` | number of entries in `pDQSetClk` array |

**Output:**

| | |
|---|---|
| `float *CloseFreq` | array of actual frequencies, Hz |
| `uint32 *entries` | pointer to actual number of entries set |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function programs layer clocks, using the `DQSETCLK` structure.

The firmware selects the closest possible speed (due to a division remainder of base frequency divided by requested rate). The function returns the actual rate in `CloseFreq` array.

`DQCMD_SETCLK` uses the following structure. Multiple entries are allowed.

```
/* DQCMD_SETCLK */
typedef struct {
      uint8 dev;      // device number
      uint8 ss;       // channel/subsystem/CVCL
      uint8 clocksel; // select clock/trigger settings
      float frq;
} DQSETCLK, *pDQSETCLK;
```

`clocksel` must contain one of the following values:

```
// Clock identifiers
#define DQ_LN_CLKID_DUTY1          (1L<<7)    // Duty cycle of TMR1 (0 is a single pulse)
#define DQ_LN_CLKID_DUTY0          (1L<<6)    // Duty cycle of TMR0 (0 is a single pulse)
#define DQ_LN_CLKID_TMR1           (1L<<5)    // TMR1 (burst clock)
#define DQ_LN_CLKID_TMR0           (1L<<4)    // TMR0 (conversion base clock)
#define DQ_LN_CLKID_CVIN           (1L<<3)    // CV Input SS clock
#define DQ_LN_CLKID_CVOUT          (1L<<2)    // CV Output SS clock
#define DQ_LN_CLKID_CLIN           (1L<<1)    // CV Input SS clock
#define DQ_LN_CLKID_CLOUT          (1L<<0)    // CV Output SS clock
```

```
Note: when the following clocksel parameters are used, *CloseFreq will return 0.0 .
These defines allow the user to use an alternate clock source for the TMR0 and TMR1 timers
The frq parameter becomes a divider value for the TMR involved. Requires logic rev 02.11.1a
or greater.

#define DQ_LN_CLKID_SRC_TMR0_66M    (0x50)     // 66MHz (default)
#define DQ_LN_CLKID_SRC_TMR0_TMR1   (0x53)     // TMR1 (pulse)
#define DQ_LN_CLKID_SRC_TMR0_IEXT0  (0x54)     // iso_ext0
#define DQ_LN_CLKID_SRC_TMR0_IEXT1  (0x55)     // iso_ext1
#define DQ_LN_CLKID_SRC_TMR0_IINT0  (0x58)     // "Internal" source 0
#define DQ_LN_CLKID_SRC_TMR0_IINT1  (0x59)     // "Internal" source 1
#define DQ_LN_CLKID_SRC_TMR0_SYNC0  (0x5C)     // External SYNC line 0
#define DQ_LN_CLKID_SRC_TMR0_SYNC1  (0x5D)     // External SYNC line 1
#define DQ_LN_CLKID_SRC_TMR0_SYNC2  (0x5E)     // External SYNC line 2
#define DQ_LN_CLKID_SRC_TMR0_SYNC3  (0x5F)     // External SYNC line 3
#define DQ_LN_CLKID_SRC_TMR1_66M    (0x60)     // 66MHz (default)
#define DQ_LN_CLKID_SRC_TMR1_TMR0   (0x61)     // TMR0 (pulse)
#define DQ_LN_CLKID_SRC_TMR1_IEXT0  (0x64)     // iso_ext0
#define DQ_LN_CLKID_SRC_TMR1_IEXT1  (0x65)     // iso_ext1
#define DQ_LN_CLKID_SRC_TMR1_IINT0  (0x68)     // "Internal" source 0
#define DQ_LN_CLKID_SRC_TMR1_IINT1  (0x69)     // "Internal" source 1
#define DQ_LN_CLKID_SRC_TMR1_SYNC0  (0x6C)     // External SYNC line 0
#define DQ_LN_CLKID_SRC_TMR1_SYNC1  (0x6D)     // External SYNC line 1
#define DQ_LN_CLKID_SRC_TMR1_SYNC2  (0x6E)     // External SYNC line 2
#define DQ_LN_CLKID_SRC_TMR1_SYNC3  (0x6F)     // External SYNC line 3
```

**Note:**

None

### 2.2.31 DqCmdSwTrigger

**Syntax:**

```
int DqCmdSwTrigger(int Iom, uint32 Mask)
```

**Command:**

`DQCMD_START` (0x138), Start/Stop Devices by Mask

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint32 Mask` | layer mask |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

The function issues a software start trigger pulse. If the configuration defines a software trigger, the layer will be programmed, but idle, from the moment of switching into operation state to the moment of receiving a software trigger command (i.e. will not produce any data). The software trigger command causes the logic to issue a start trigger pulse to start data conversion. Every bit in `Mask` represents a layer in the device stack, where bit 0 corresponds with device 0, etc. For example, to issue a software start trigger for layer 1, Mask = (1L << 1);

**Note:**

None

### 2.2.32 DqCmdSetChannelList

**Syntax:**

`int DqCmdSetChannelList(int Iom, pDQSETCL pDQSetCl, uint32 *entries)`

**Command:**

`DQCMD_SETCL` (0x13C)

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQSETCL pDQSetCl` | array of channel list entries |
| `uint32 *entries` | number of entries |

**Output:**

| | |
|---|---|
| `uint32 *entries` | number of entries actually processed |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Timeout duration |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

Sets up the device channel list. This function parses the presented channel list and fills the input and output channel lists.

Channel list entries are stored in the `DQSETCL` structures:

```
typedef struct {
    uint8 dev;     // device number
    uint8 ss;      // subsystem
    uint32 entry;  // channel list entry
} DQSETCL, *pDQSETCL;
```

The channel list usually has following format:

```
    31 - 15      15 - 8         7 - 0
    <flags>    <mode,gain>     <ch number>
```

`If any particular device has a channel list that is different from this format, there will be device specific macros and defines listed in the powerdna.h file.`

Flags are defined as follows:

```
    #define DQ_LNCL_NEXT    (1UL<<31)    // channel list has next entry
    #define DQ_LNCL_INOUT   (1UL<<30)    // (reserved for future use)
    #define DQ_LNCL_SS1     (1UL<<29)    // (reserved for future use)
    #define DQ_LNCL_SS0     (1UL<<28)    // (reserved for future use)
    #define DQ_LNCL_IRQ     (1UL<<27)    // (reserved for future use)
    #define DQ_LNCL_NOWAIT  (1UL<<26)    // (reserved for future use)
    #define DQ_LNCL_SKIP    (1UL<<25)    // (reserved for future use)
    #define DQ_LNCL_CLK     (1UL<<24)    // (reserved for future use)
    #define DQ_LNCL_CTR     (1UL<<23)    // (reserved for future use)
    #define DQ_LNCL_WRITE   (1UL<<22)    // write to the channel but not update
    #define DQ_LNCL_UPDALL  (1UL<<21)    // update all written channels
    #define DQ_LNCL_TSRQ    (1UL<<20)    // (reserved for future use)
```

```
#define DQ_LNCL_SLOW    (1UL<<19)    // slow down operation
#define DQ_LNCL_DIO     (1UL<<18)    // write/read DIO
#define DQ_LNCL_RSVD1   (1UL<<17)    // (reserved for future use)
#define DQ_LNCL_RSVD0   (1UL<<16)    // (reserved for future use)
#define DQ_LNCL_DIFF    (1UL<<15)    // differential mode
```

**Note:**

`DQCMD_SETCL` is an additive function. Each time you write it, it adds a channel to the existing channel list. Please reset the device to clear the channel list.

Only a few flags are supported by current set of layers: AI-201/205 (`DQ_LNCL_SLOW`, `DQ_LNCL_NEXT`, `DQ_LNCL_DIFF`) or AO-302 (`DQ_LNCL_WRITE`, `DQ_LNCL_UPDALL`).

### *2.2.33 DqCmdSetTransferList*

**Syntax:**

`int DqCmdSetTransferList(int Iom, pDQSETTRL pDQSetTrl)`

**Command:**

`DQCMD_SETTRL` (0x140), Set Transfer List

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQSETTRL pDQSetTrl` | transfer list descriptor |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function sets up a transfer list for DMap operations given by `DQSETTRL` structure.

`DqCmdSetTransferList()` has additive behavior. The firmware accumulates transfer lists with the same `dmapid` in the memory. Every time the firmware finds a new `dmapid,` it allocates memory to store a transfer list.

To set up a transfer list for a device, use the `DQSETTRL` structure.

```
/* DQCMD_SETTRL */
typedef struct {
    uint16 dmapid;      // DMAP id
    uint8  dev;         // device
    uint8  ss;          // subsystem
    uint32 ch;          // channel (channel list entry)
    uint32 flags;       // control flags, including channel information
    uint16 samples;     // number of samples from this channel
} DQSETTRL, *pDQSETTRL;
```

**Note:**

This function is called automatically in `DqDmapInitOps().`

### *2.2.34 DqCmdWriteAll*

**Syntax:**

`int DqCmdWriteAll(int Iom, pDQWRRD pDQWr)`

**Command:**

`DQCMD_WRRD` (0x14C), Write All Data and Read Data Back

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQWRRD pDQWr` | data for output DMap |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Timeout duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function initiates exchange of data as described by a transfer list. Caller must fill the `pDqWr` structure with proper DMap Ids and specify size of the output data. Setting an input or output DMap ID to zero disables transferring of data in that direction.

This function doesn’t wait for the IOM to reply with the input data. Call DqCmdReadAll() to retrieve the input data.

This function is useful in a real-time environment where the real-time task can’t afford to wait for the IOM’s response. DqCmdWriteAll() should be called at the end of the real-time cycle and DqCmdReadAll() should be called at the beginning of the next cycle.

The following structure is used to exchange data during DMap operations:

```
/* DQCMD_WRRD */
typedef struct {
    uint32 dmapid;      // DMap ID
    uint32 size;        // size of data
    uint8 data[];       // data
} DQWRRD, *pDQWRRD;
```

**Note:**

None

### *2.2.35 DqCmdReadAll*

**Syntax:**

`int DqCmdReadAll(int Iom, pDQWRRD pDQRd)`

**Command:**

`DQCMD_WRRD` (0x14C), Write All Data and Read Data Back

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQWRRD pDQRd` | pointer to store input DMap data |

**Output:**

| | |
|---|---|
| `pDQWRRD pDQRd` | pointer to received input data |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function reads the response from an IOM to a request to exchange data initiated by a call to DqCmdWriteAll().

This function is useful in a real-time environment where the real-time task can't afford to wait for the IOM's response. DqCmdWriteAll() should be called at the end of the real-time cycle and DqCmdReadAll() should be called at the beginning of the next cycle.

The following structure is used to exchange data during DMap operations:

```
/* DQCMD_WRRD */
typedef struct {
    uint32 dmapid;       // DMap ID
    uint32 size;         // size of data
    uint8 data[];        // data
} DQWRRD, *pDQWRRD;
```

**Note:**

None

## 2.2.36 *DqCmdWriteReadAll*

**Syntax:**

`int DqCmdWriteReadAll(int Iom, pDQWRRD pDQWr, pDQWRRD pDQRd)`

**Command:**

`DQCMD_WRRD` (0x14C), Write All Data and Read Data Back

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQWRRD pDQWr` | data for output DMap |
| `pDQWRRD pDQRd` | pointer to store input DMap data |

**Output:**

| | |
|---|---|
| `pDQWRRD pDQRd` | pointer to received input data |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function initiates exchange of data described by transfer list. Caller must fill `pDqWr` structure with proper DMap Ids and specify size of input and output data. Setting input or output DMap ID to zero disables transferring of data in that direction.

The following structure is used to exchange data during DMap operations:

```
/* DQCMD_WRRD */
typedef struct {
    uint32 dmapid;      // DMap ID
    uint32 size;        // size of data
    uint8 data[];       // data
} DQWRRD, *pDQWRRD;
```

**Note:**

None

## 2.2.37 DqCmdWriteFIFO

**Syntax:**

`int DqCmdWriteFIFO(int Iom, pDQFIFO pDQFifo)`

**Command:**

`DQCMD_WRFIFO` (0x150), Writes Data to Device FIFO

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQFIFO pDQFifo` | output data |

**Output:**

| | |
|---|---|
| `pDQFifo->size` | number of bytes actually transferred |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Timeout duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function writes data to the FIFO on the layer. The subsystem `pDQFifo->ss` field represents a physical subsystem (SS0IN, for example). `DQ_FIFO_SET_DATA` (0x10) defines a channel to write data to the layer.

The following structure is used in the process of exchanging data stream between host and IOM:

```
/* DQCMD_WRFIFO and DQCMD_RDFIFO */
typedef struct {
    uint8 dev;
    uint8 ss;
    uint16 size;
    uint8 data[];
} DQFIFO, *pDQFIFO;
```

**Note:**

Maximum data size is 514 bytes.

ACB operations performed by DQE are based on exchanging `DQCMD_WRFIFO` packets. Please see User Manual for internal working of DQE.

### *2.2.38 DqCmdReadFIFO*

**Syntax:**

`int DqCmdReadFIFO(int Iom, pDQFIFO pDQFifo)`

**Command:**

`DQCMD_RDFIFO` (0x154), Read Device Data from the FIFO

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQFIFO pDQFifo` | pointer to store input data |
| `pDQFifo->size` | number of bytes requested |

**Output:**

| | |
|---|---|
| `pDQFifo->size` | number of bytes actually transferred |
| `pDQFifo->data` | data received |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function requests data from the FIFO on the layer. The subsystem `pDQFifo->ss` field represents a physical subsystem (`DQ_SS0IN`, for example). `DQ_FIFO_GET_DATA` (0x10) defines a channel for reading data from the layer. `DQ_FIFO_GET_CAL` (0x20) defines a channel for retrieving calibration data.

The following structure is used in the process of retrieving a data stream from the IOM:

```
/* DQCMD_WRFIFO and DQCMD_RDFIFO */
typedef struct {
    uint8 dev;
    uint8 ss;
    uint16 size;  // bytes
    uint8 data[];
} DQFIFO, *pDQFIFO;
```

**Note:**

The maximum data size is 1432 bytes. `pDQFifo->data` must be at least `pDQFifo->size` bytes in size.

If the returned data are not entirely composed of 8-bit data then the appropriate network byte order corrections must be performed by the user by applying the ntohl() and ntohs() functions.

ACB operations performed by DQE are based on exchanging `DQCMD_RDFIFO` packets. Please see User Manual for internal workings of DQE.

## *2.2.39 DqCmdReadFIFO32*

**Syntax:**

`int DqCmdReadFIFO32(int Iom, pDQFIFO pDQFifo)`

**Command:**

`DQCMD_RDFIFO` (0x154), Read Device Data from the FIFO

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQFIFO pDQFifo` | struct pointer to store input data |
| `pDQFifo->ss` | Channel number or one of the following defines:<br>`DQ_FIFO_GET_DATA`<br>`DQ_FIFO_GET_CAL`<br>`DQ_FIFO_GET_CUSTOM` |
| `pDQFifo->size` | number of 32-bit words requested |

**Output:**

| | |
|---|---|
| `pDQFifo->size` | number of 32-bit words actually transferred |
| `pDQFifo->data` | data received |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function requests data from the FIFO on the layer. The subsystem `pDQFifo->ss` field represents a physical subsystem (`DQ_SS0IN`, for example). `DQ_FIFO_GET_DATA` (0x10) defines a channel for reading data from the layer. `DQ_FIFO_GET_CAL` (0x20) defines a channel for retrieving calibration data.

The following structure is used in the process of retrieving a data stream from the IOM:

```
/* DQCMD_WRFIFO and DQCMD_RDFIFO */
typedef struct {
    uint8 dev;
    uint8 ss;
    uint16 size;   // number of 32-bit words
    uint32 data[];
} DQFIFO32, *pDQFIFO32;
```

This function is used to access 32-bit data only. The function internally performs correction for network byte order. If the fifo read data for this particular layer is not entirely composed of 32-bit data then the DqCmdReadFIFO() function must be used instead and the appropriate network byte order corrections must be performed by the user by applying the ntohl() and ntohs() functions.

**Note:**

The maximum data size is 358 words. `pDQFifo->data` must be at least `pDQFifo->size` 32-bit words in size.

### 2.2.40 DqCmdWriteReadFIFO

**Syntax:**

`int DqCmdWriteReadFIFO(int Iom, pDQWRRDFIFO pDQWrRdFifo)`

**Command:**

`DQCMD_WRRDFIFO` (0x158), Write Data to the Device FIFO and read it back

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQWRRDFIFO pDQWrRdFifo` | pointer to input/output data |

**Output:**

| | |
|---|---|
| `pDQWrRdFifo->sizew` | number of bytes actually written |
| `pDQWrRdFifo->sizer` | number of bytes actually read |
| `pDQWrRdFifo->data` | data from the device |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function writes data to the FIFO and retrieves data back (from the same device/subsystem). This function combines the functionality of `DqCmdRdFifo()` and `DqCmdWrFifo()` in one call. It produces one request and receives one reply packet.

The following structure is used in the process of exchanging a data stream between host and IOM:

```
/* DQCMD_WRRDFIFO */
typedef struct {
    uint8 dev;        // device
    uint8 ssw;        // subsystem to write
    uint8 ssr;        // subsystem to read
    uint16 sizew;     // amount of data to write
    uint16 sizer;     // amount of data to read
    uint8 data[];
} DQWRRDFIFO, *pDQWRRDFIFO;
```

**Note:**

DQE doesn't use this call. You can use this function to retrieve information from device with an input and output subsystem. If you select `DQ_SS0IN` as a subsystem to write, the function uses `DQ_SS0OUT` as a subsystem to read. In other words, this function ignores the direction bit in subsystem definition.

### 2.2.41 DqCmdWriteToFlashBuffer

**Syntax:**

`int DqCmdWriteToFlashBuffer(int Iom, uint32 Size, uint8 *Data, uint32 *CRC)`

**Command:**

DQCMD_WRFLASH (0x15C), Write Data to the Flash Update Buffer

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint32 Size` | amount of data to write, bytes, `0` to reset |
| `uint8 *Data` | data to write |
| `uint32 *CRC` | pointer to uninitialized CRC |

**Output:**

| | |
|---|---|
| `uint32 *CRC` | Standard IEEE 802.3 CRC-32 value of data from user |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM, or packet got corrupted |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function transfers large amount of data into IOM memory. This data can be stored into one of the IOM flash memory locations upon a `DqCmdUpdateFlashBuffer()` call. Every time a user calls this function, it transfers data and allocates a chunk of memory to store this data. Set `Size` to `0` to reset memory.

**Note:**

Do not call in operating mode.
Use the firmware update utility supplied for safe update.
This function is password-protected with super-user level password

### *2.2.42 DqCmdUpdateFlashBuffer*

**Syntax:**

```
int DqCmdUpdateFlashBuffer(int Iom, uint16 Sector, uint32
Address, uint32 Size, uint32 *CRC)
```

**Command:**

`DQCMD_UPDFLASH` (0x160), Update Flash Data from the Flash Update Buffer

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `u16 sector` | Flash sector to start from |
| `u32 address` | Flash chip address |
| `u32 size` | amount of data to write |
| `uint32 *CRC` | pointer to uninitialized CRC |

**Output:**

| | |
|---|---|
| `uint32 *CRC` | Standard IEEE 802.3 CRC-32 value of data in flash memory |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function initiates a write operation into a firmware sector. The current CM-1 layer has two flash chips that are located at different addresses. The system flash is located at `DQ_FLASH_ADDRESS` and has a size of `DQ_FLASH_SIZE.` The firmware starts at sector `DQ_UPUSER_SEC`. These are the only allowable parameters for writing system flash. Auxiliary flash is located at `DQ_FLASHAUX_ADDRESS` and has a size of `DQ_FLASHAUX_SIZE`. All of auxiliary flash is available to the user.

**Note:**

This is a pending command. Use a large timeout. Do not call in operating mode. This operation takes tens of seconds to complete. At the time of Flash update, the device will be unreachable via DaqBIOS.

Use the firmware update utility supplied for safe update.

This function is password-protected with super-user level password

### *2.2.43 DqCmdSetCommParameters*

Syntax:

```
int DqCmdSetCommParameters(int Iom, pDQSETCOMM pDQSetComm,
uint32 *CRC)
```

**Command:**

`DQCMD_SETCOMM` (0x164), Set Communication Parameters

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQSETCOMM pDQSetComm` | communication parameters |
| `uint32 *CRC` | pointer to uninitialized CRC (unused, pass in `NULL`) |

**Output:**

| | |
|---|---|
| `pDQSETCOMM pDQSetComm` | current parameter values, if value of passed in `pDQSetComm->todo` field was `DQ_READCOMM` |
| `uint32 *CRC` | CRC of data actually written (unused, always returns `0` if not `NULL`) |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function sets device communication parameters: Ethernet, serial, UDP, and startup.

`DQCMD_SETCOMM` uses the following structure to set or retrieve communication parameters of a PowerDNA cube:

```
/* DQCMD_SETCOMM */
typedef struct {
    uint8 todo;            // function to perform
    uint8  MAC[6];         // MAC address
    uint32 netip;          // IP address
    uint32 gateway;         // gateway
    uint32 netmask;         // network mask
    uint32 startup;         // startup state
    uint32 baudrate;        // baud rate for serial interface
    uint16 udpport;         // default UDP port
    uint8   signature[28];  // logger license code, first char NULL if unused
} DQSETCOMM, *pDQSETCOMM;
```

`todo` can be one of the following:

- `DQ_READCOMM` (1): read parameters. This function returns parameters read from the parameter memory
- `DQ_WRITECOMM` (2): write parameters. This function writes specified parameters into parameter memory.
- `DQ_READCOMM2` (3): read ip2 parameters. This function returns ip2 parameters read from the parameter memory
- `DQ_WRITECOMM2` (4): write ip2 parameters. This function writes specified ip2 parameters into parameter memory. The startup, baudrate, udpport and signature members of the struct will be ignored
- `DQ_WRITECOMMSRV` (5): same as `DQ_WRITECOMM` except that the netip address is also written to the srv address
- `DQ_WRITECOMM2SRV` (6): same as `DQ_WRITECOMM2` except that the netip2 address is also written to the srv2 address

`startup` consists of four bytes:

- `uint8 autorun`: `0` – wait for user input from terminal, `1` – load and execute firmware
- `uint8 runtype`: `0` – wait in configuration mode, `1` – switch to operating mode upon initialization
- `uint8 portnum`: unused, set to `0`
- `uint8 protocol`: unused, set to `0`

Byte order is big-endian:

| | | | |
|---|---|---|---|
| autorun | runtype | portnum | protocol |

**Note:**

This is a pending command. Use a large timeout. Do not call in operating mode.

Parameters will not affect execution right away. To make parameters go into effect, the user has to reset the PowerDNA cube – either physically or by issuing a `DQCMD_RST` command.

This function is password-protected with user level password

### 2.2.44 DqCmdSetName

**Syntax:**

`int DqCmdSetName(int Iom, char *Name, int32 *CRC)`

**Command:**

`DQCMD_SETNAME` (0x168), Set Device Name

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `char *Name` | ASCIIZ name of the IOM (`DQ_DEVNAME_SIZE` bytes max) |
| `int32 *CRC` | pointer to store CRC |

**Output:**

| | |
|---|---|
| `uint32 *CRC` | Standard IEEE 802.3 CRC-32 value of the written name |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function stores the IOM name in the IOM RAM. User has to store the PowerDNA cube name in Flash memory for the change to become permanent.

**Note:**

This function is password-protected with user level password.

### *2.2.45 DqCmdGetName*

**Syntax:**

```
int DqCmdGetName(int Iom, uint32 BufLen, char *Buffer)
```

**Command:**

`DQCMD_SETNAME` (0x168), Read Device Name

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint32 BufLen` | size of `Buffer` in bytes |
| `char *Buffer` | buffer to store ASCIIZ name of the IOM (`DQ_DEVNAME_SIZE` bytes max) |

**Output:**

| | |
|---|---|
| `Char *Buffer` | retrieved IOM name ASCIIZ string |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function retrieves IOM name in the IOM RAM.

**Note:**

None

### 2.2.46 DqCmdSetWatchDog

**Syntax:**

```
int DqCmdSetWatchDog(int Iom, uint32 Mask, uint32 Time, uint32*
T2reset, uint32* Status)
```

**Command:**

`DQCMD_WDSET` (0x172), Set/Reset watchdog

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint32 Mask` | Watchdog mask |
| `uint32 Time` | Watchdog expiration time in milliseconds |

**Output:**

| | |
|---|---|
| `uint32* T2reset` | Time in milliseconds left before reset since last call |
| `uint32* Status` | Current mode of operation |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

The function selects watchdog mode and time. It can also return the time left before reset and the IOM operation mode

**Note:**

Mask can take one of the following values on all versions including UEIPAC:

- DQ_WD_CLEAR_DISABLED Disable watchdog
- DQ_WD_CLEAR_RESET Reset the countdown counter
- DQ_WD_CLEAR_STATUS Return the status, do not reset watchdog

Mask can be ORed with any of the following values on non-UEIPAC devices:

- DQ_WD_CLEAR_ON_CONSOLE Clear watchdog when a character is typed at the serial console
- DQ_WD_CLEAR_ON_RECEIVE Clear watchdog when a DAQBIOS request is received
- DQ_WD_CLEAR_ON_TRANSMIT Clear watchdog when a DAQBIOS response is transmitted
- DQ_WD_CLEAR_ON_OSTASK Clear watchdog periodically in the main firmware task

Status contains one of the following values:

- DQ_IOMODE_INIT IO module is being initialized
- DQ_IOMODE_CFG IO module in configuration mode
- DQ_IOMODE_OPS IO module in operation mode
- DQ_IOMODE_SD IO module in shutdown mode

### *2.2.47 DqCmdSetParameters*

**Syntax:**

int DqCmdSetParameters(int Iom, pDQSETPRM pDQSetParm, uint32 *CRC)

**Command:**

`DQCMD_SETPRM` (0x170), Set Power-up Values

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQSETPRM pDQSetParm` | pointer to parameter specification |
| `uint32 *CRC` | pointer to uninitialized CRC (unused, pass in `NULL`) |

**Output:**

| | |
|---|---|
| `uint32 *CRC` | CRC of data actually written (unused, always returns `0` if not `NULL`) |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function sets up named and modal layer parameters. These parameters reside in a device object and can be partially stored in layer EEPROM. One has to know the EEPROM structure of the layer being accessed to set parameters correctly.

`DQCMD_SETPRM` is used to set and retrieve layer parameters:

```
/* DQCMD_SETPRM  */
typedef struct {
    uint8 dev;          // device
    uint8 ss;           // subsystem
    uint8 mode;         // parameter mode
    uint8 value[];     // array of values
} DQSETPRM, *pDQSETPRM;
```

The following values for `mode` are available:

```
                                // Returned structures (xxx –layer model)
#define DQ_IOM_ACCESS_INIT         // INITPRM_xxx_
#define DQ_IOM_ACCESS_CALIBR       // CALSET_xxx_
#define DQ_IOM_ACCESS_OPERS        // OPMODEPRM_xxx_
#define DQ_IOM_ACCESS_SHUTDOWN     // SDOWNPRM_xxx_
#define DQ_IOM_ACCESS_NAMEDPRM     // named parameters in <value> are expected
#define DQ_IOM_ACCESS_NAMES        // CNAMES_xx_
#define DQ_IOM_ACCESS_EECMNDEVS    // EECMNDEVS
```

The first four modes are used to store (copy to memory) `DQOPMODEPRM_`… (`DQ_IOMODE_OPS`) , `DQINITPRM_`… (`DQ_IOMODE_INIT`), `DQSDOWNPRM`…(`DQ_IOMODE_SD`), `DQCALSET_`… (`DQ_IOMODE_CFG`).  If one needs to store or retrieve names of channels, (`DQ_IOMODE_NAMES`) should be used. Appropriate data should be stored in `value` array for set operation. The function performs a direct copy of the contents of the buffer into IOM device object memory. Thus, one should check firmware and library versions prior to calling this function — to avoid incompatibility in structures defined in library and firmware.

A special mode is used to store or retrieve named parameters. Named parameters cannot be stored in EEPROM, but instead can be used to affect operations of the PowerDNA cube. To set up or retrieve named parameters, pass `IOMODE_NAMEDPRM` as `mode` and one of the following named parameters as uint32 field in the `value` buffer:

```
#define DQ_IOPRM_NBUFS     0x100      // number of buffers for streaming
#define DQ_IOPRM_CLPERINT  0x200      // number of channel lists per interrupt
#define DQ_IOPRM_ADDLDELAY 0x400      // additional delay control
#define DQ_IOPRM_RQID      0x1000     // Request id - for streaming RDFIFO
#define DQ_IOPRM_MIDPOS0   0x1100     // watermark control for midposion interrupt
```

**Note:**

Named parameters are layer-dependent. Not all of them are implemented.

### *2.2.48 DqCmdGetParameters*

**Syntax:**

`int DqCmdGetParameters(int Iom, pDQGETPRM pDQGetParm, uint8 *data)`

**Command:**

`DQCMD_GETPRM` (0x171), Get Power-up Values

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQGETPRM pDQGetParm` | specified parameters |
| `uint8 *data` | pointer to uninitialized data |

**Output:**

| | |
|---|---|
| `uint8 *data` | retrieved data |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function requests named and modal layer parameters from IOM. These parameters reside in the device object and can be partially stored in layer EEPROM.

Before calling this function, caller should fill out the `DQGETPRM` structure.

```
/* DQCMD_GETPRM  */
typedef struct {
      uint8 dev;   // device
      uint8 ss;    // subsystem
      uint8 mode;  // parameter mode
   uint8 value[];  // array of values
} DQGETPRM, *pDQGETPRM;
```

Following values for `mode` are available:

```
                                  // Returned structures (xxx –layer model)
#define DQ_IOM_ACCESS_INIT        // INITPRM_xxx_
#define DQ_IOM_ACCESS_CALIBR      // CALSET_xxx_
#define DQ_IOM_ACCESS_OPERS       // OPMODEPRM_xxx_
#define DQ_IOM_ACCESS_SHUTDOWN    // SDOWNPRM_xxx_
#define DQ_IOM_ACCESS_NAMEDPRM    // named parameters in <value> are expected
#define DQ_IOM_ACCESS_NAMES       // CNAMES_xx_
#define DQ_IOM_ACCESS_EECMNDEVS   // EECMNDEVS
```

The first four modes are used to store (copy to memory) `DQOPMODEPRM_`… (`DQ_IOMODE_OPS`) , `DQINITPRM_`… (`DQ_IOMODE_INIT`), `DQSDOWNPRM`…(`DQ_IOMODE_SD`), `DQCALSET_`… (`DQ_IOMODE_CFG`). If one needs to store or retrieve names of channels, (`DQ_IOMODE_NAMES`) should be used. Appropriate data should be stored in the `value` array for set operation. The function performs a direct copy of the contents of the buffer into IOM device object memory. Thus, one should check firmware and library versions prior to calling this function — to avoid incompatibility in structures defined in library and firmware.

A special mode is used to store or retrieve named parameters. Named parameters cannot be stored in EEPROM, but instead can be used to affect operations of the PowerDNA cube. To set up or retrieve named parameters, pass `IOMODE_NAMEDPRM` as `mode` and one of the following named parameters as a uint32 field in the `value` buffer:

```
#define DQ_IOPRM_NBUFS     0x100      // number of buffers for streaming
#define DQ_IOPRM_CLPERINT  0x200      // number of channel lists per interrupt
#define DQ_IOPRM_ADDLDELAY 0x400      // additional delay control
#define DQ_IOPRM_RQID      0x1000     // Request id - for streaming RDFIFO
```

**Note:**

Named parameters are layer-dependent. Not all of them are implemented.

### *2.2.49 DqCmdSaveParameters*

**Syntax:**

`int DqCmdSaveParameters(int Iom, uint32 devn, uint32 *HashCode)`

**Command:**

`DQCMD_SAVEPRM` (0x178), Save Parameters

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint32 devn` | layer ID to store data into EEPROM |
| `uint32 *HashCode` | pointer to receive hash code of saved parameters |

**Output:**

| | |
|---|---|
| `uint32 *HashCode` | hash code of saved parameters |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function writes parameters stored in device object into layer EEPROM or system Flash (parameter area). User has to specify one of the values as `devn`:

| | |
|---|---|
| Actual layer id (0x0 – 0xf) | write layer EEPROM |
| `DQ_LNPRM_DEVIOM` (0xFE) | write parameter sector of the Flash |
| `DQ_LNPRM_DEVALL` (0xFF) | save both |

**Note:**

This command can take several seconds to complete (pending command execution). Please reset the PowerDNA cube for parameters to go into effect.

### *2.2.50 DqCmdSetCalibration*

**Syntax:**

```
int DqCmdSetCalibration(int Iom,  pDQSETCAL pDQSetCal, int
*entries)
```

**Command:**

`DQCMD_SETCAL` (0x174), Set up Calibration Values

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQSETCAL pDQSetCal` | Array of `DQSETCAL` structures containing calibration values |
| `int *entries` | Number of calibration entries in `pDQSetCal` |

**Output:**

| | |
|---|---|
| `int *entries` | Number of calibration entries actually processed |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function sets up values of calibration DACs. The main purpose of it is to calibrate hardware. Not every piece of hardware can be calibrated; some layers (like the AI-225) do not have calibration circuitry or do not require calibration (DIO-403). See the User Manual for layer-specific details.

Every calibration entry is represented by the `DQSETCAL` structure.

```
/* DQCMD_SETCAL */
typedef struct {
    uint8 dev;      // device
    uint8 ss;       // subsystem
    uint8 channel;  // channel
    uint8 dac;      // parameter mode
    uint32 value;   // DAC value to write
} DQSETCAL, *pDQSETCAL;
```

**Note:**

User has to specify `pDqSetCal->dev`, `pDqSetCal->ss`, `pDqSetCal->dac` and `pDqSetCal->value`. `pDqSetCal->channel` field is reserved for future use.

## *2.2.51 DqCmdSetMode*

**Syntax:**

`int DqCmdSetMode(int Iom, uint32 Mode, uint32 Mask)`

**Command:**

`DQCMD_SETMD` (0x17C). Set Mode of Operation

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint32 Mode` | mode to switch IOM layers into |
| `uint32 Mask` | what layers to switch (mask) (bits 0 through11) |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

The function switches IOM layers between operation, configuration, shutdown, and init modes. The valid `Mode` settings are:

```
#define DQ_IOMODE_INIT   1L     // device is being initialized
#define DQ_IOMODE_CFG    2L     // device in configuration mode
#define DQ_IOMODE_OPS    4L     // device in operation mode
#define DQ_IOMODE_SD     8L     // set device in shutdown mode

// Power management modes
#define DQ_IOMODE_SLEEP   0x10    // sleep mode
#define DQ_IOMODE_PWRDN   0x20    // power down device
#define DQ_IOMODE_PWRUP   0x40    // switch device power on

// The following extended modes are intended for multi-master IOMs
#define DQ_IOMODE_GETCTRBUS  0x100   // become a master on the bus
#define DQ_IOMODE_GIVEUPBUS  0x200   // become a slave on the bus
```

Current implementation of IOM does not have power-management modes.

When the PowerDNA cube is powered-up and configuration is set to normal, it loads firmware, and goes into initialization mode. In initialization mode, it sets all output to pre-defined values and initializes hardware and software. The PowerDNA cube then automatically switches all layers into configuration mode. In configuration mode, data acquisition is not running and the user can set up acquisition parameters. Also, the user can use the `DQCMD_IOCTL`-based command to retrieve input voltages or set output voltages on a scan-by-scan basis. Before entering configuration mode, the firmware reads all parameters of the operating mode stored in EEPROM (configuration, channel list, clock and trigger settings, etc.), processes it, and stores it in the operating mode current parameter set. In other words, by switching the layer into operation mode, the PowerDNA cube can start data acquisition right away with previously stored parameters. Switching the layer back into configuration mode stops the data acquisition process.

The user can switch a layer between configuration and operation modes as many times as needed.
If the user switches the layer into shutdown mode, the firmware retrieves the previously stored output levels of layer outputs and sets up output voltages. The unit can enter shutdown mode automatically upon watchdog timeout expiration.

Once a unit is in shutdown mode, there are three ways to return to a normal idle state (configuration mode). 1) Power cycle the unit, 2) Issue a `DQCMD_RST` command or 3) Issue two `DqCmdSetMode` commands, the first with mode = `DQ_IOMODE_INIT` and the second with mode= `DQ_IOMODE_CFG`. Note that both power-up and reset will automatically issue the `DQ_IOMODE_INIT` and `DQ_IOMODE_CFG` mode commands for all layers.

`Mask` is a bitmask representation, in which every bit represents a layer. For example, to switch Layer 2 into initialization mode, the mask should be (1 << 2) = 4.
The `Mask` should only contain a combination of bits 0 through bit 11.

**Note:**

None

### 2.2.52 DqCmdSetReplyMaxSize

**Syntax:**

```
int DqCmdSetReplyMaxSize(int Iom, uint32 EthSize, uint32
EthSizeBuf, uint32 MaxNoPkts, uint32 *accepted)
```

**Command:**

`DQCMD_SETRPLMAX` (0x180), Set Maximum Reply Packet Size

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint32 EthSize` | Maximum size of the packet produced by IOM |
| `uint32 EthSizeBuf` | Maximum Ethernet size for buffered mode |
| `uint32 MaxNoPkts` | Maximum number of packets |

**Output:**

| | |
|---|---|
| `uint32 *accepted` | TRUE if IOM accepted requested sizes |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function sets the size of the packets generated by an IOM. This is typically used for point-by-point or asynchronous communication where a serial message exceeds size of typical packet and needs support of fragmented packets.

For fragmented packets, `EthSize` and `EthSizeBuf` can be set to max allowed size of Ethernet packet, 1518(`DQ_MAX_ETH_SIZE_100`). `MaxNoPkts` should be set to a value between 1 and `DQ_MAX_FRAGMENTS` (inclusive).

**Note:**

This function is not typically needed within an application and used for point-by-point or asynchronous communication where a serial message exceeds size of typical packet (i.e MIL-1553 asynchronous events). For RTVMAP/RTDMAP fragmented packets, refer to `DqRtVmapInitEx()/DqRtDmapInitEx()`.

### *2.2.53 DqCmdSetPassword*

**Syntax:**

```
int DqCmdSetPassword(int Iom, uint32 Mode, char *Password)
```

**Command:**

`DQCMD_SETPASS` (0x184)

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint32 Mode` | function mode (read/write/set) |
| `char *Password` | ASCIIZ password |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

Function controls setting IOM passwords:

```
#define DQ_SETPASS_SUPASS  (1L<<0) // Confirm SU password
#define DQ_SETPASS_USRPASS (1L<<1) // Confirm user password
#define DQ_SETPASS_SETSU   (1L<<2) // Set SU password
#define DQ_SETPASS_SETUSR  (1L<<3) // Set user password
#define DQ_SETPASS_CLEAR   (1L<<4) // clear passwords for current session
#define DQ_SETPASS_EN_DQCMD0 (1L << 5) // Open access to the protected commands on NIC 1
#define DQ_SETPASS_EN_DQCMD1 (1L << 6) // Open access to the protected commands on NIC 2
```

The system requires confirmation of a password before changing parameters. Thus, one should call the `DQCMD_SETPASS` function to confirm the super-user or user password. Once the password is confirmed, the functions that required a password become accessible. To switch them off, call this function again to clear passwords. `DQCMD_SETPASS` is also used to change passwords. First, you have to confirm the related password and only then call the function again to change it. Regardless of whether the password has been changed successfully or not, the flag to execute password-protected functions is cleared. You have to send passwords again to call password-protected functions.

This command can also be used to unlock commands that are locked due to the mode of operation. To do it call this functions with a mode DQ_SETPASS_EN_DQCMD0 for the NIC1 (or NIC if only) port and DQ_SETPASS_EN_DQCMD1 for NIC2 (or diagnostics port). Before you can call it either user or superuser password should be confirmed using <mode==DQ_SETPASS_USRPASS> call.

For DQ_SETPASS_EN_DQCMDx data string should be in form of "1XX Y", where 1XX is a DQCMD command code and Y is one of the follows:

```
#define DQ_EXE_LIMITED  0     // command can be executed at any time, except following:
#define DQ_EXE_OPMD_OK  1     // command can be executed in op mode
#define DQ_EXE_DIAG_OK  2     // execution allowed in diagnostics mode
#define DQ_EXE_OP_DIAG  3     // in both operation and diagnostics
#define DQ_EXE_CONF_OK  4     // good to call in configuration mode
```

DQ_EXE_OP_DIAG is the most relaxed requirement.
DQ_EXE_OPMD_OK is set for the commands that are required in operating mode.
DQ_EXE_OP_DIAG is set for the commands that are allowed to be executed on diagnostics port while the cube is in the operating mode

**Note:**

None

### *2.2.54 DqCmdGetCRC*

**Syntax:**

`int DqCmdGetCRC(int Iom, DQCRCINFO *pDQCRCInfo)`

**Command:**

`DQCMD_GETCRC` (0x188), Returns IEEE 802.3 CRC-32 values for parameter areas

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `DQCRCINFO *pDQCRCInfo` | pointer to uninitialized CRC information |

**Output:**

| | |
|---|---|
| `DQCRCINFO` | CRC information |

`*pDQCRCInfo`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function returns the IEEE 802.3 CRC-32 codes of different configuration areas. The areas `DQCRCINFO` structure has the following fields:

```
/* DQCMD_GETCRC */
typedef struct {
     uint32 fwcrc;     // CRC of firmware code
     uint32 fwver;     // FW version
     uint32 paramcrc;  // CRC of parameter table
     uint32 initcrc;   // Init mode CRC
     uint32 opercrc;   // Operation mode CRC
     uint32 sdcrc;     // Shutdown CRC
     uint32 build;     // Build
} DQCRCINFO, *pDQCRCINFO;
```

where `FWVER` can be converted to string by the following method:

```
sprintf(strver, “%d.%d.%d.%d”, CRCInfo.fwver>> 24, (CRCInfo.fwver>>16)&0xFF,
                               (CRCInfo.fwver>>8)&0xFF, CRCInfo.fwver&0xFF);
```

and the `build` is printed as a decimal value.

**Note:**

None

### 2.2.55 DqCmdGetNVRAM

**Syntax:**

```
int DqCmdGetNVRAM(int Iom, uint32* nvram_data, uint32* size)
```

**Command:**

`DQCMD_GETNVR` (0x133), Returns NVRAM diagnostics content. For 5200 systems only

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint32* nvram_data` | Pointer to status data stored in NVRAM (allocate 0x10 bytes) |
| `uint32* nvram_size` | Pointer to number of words returned |

**Output:**

| | |
|---|---|
| `uint32* nvram_data` | Status data stored in NVRAM |
| `uint32* nvram_size` | Number of words returned |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |

| | |
|---|---|
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function is supported on 5200 systems ONLY (e.g., DNA-PPC5, DNA-PPC8) and returns NVRAM diagnostics content stored in NVRAM of the watchdog IC.

Upon entering Operation (OPS) mode, firmware stores POST and firmware status flags in non-volatile memory. In the event of a powerdown, users can get a snapshot of the state of the system before the powerdown by calling the `DqCmdGetNVRAM()` API before re-entering Operation mode. Operation mode is entered when starting ACB or VMAP mode.

The API returns the following:

- `nvram_data[0] =` current mode (1=INIT; 2=CONFIG; 4=OPS; 8=SHUTDOWN)
- `nvram_data[1] =` POST values for the CPU layer
- `nvram_data[2] =` FW status values (combined)

Behavior descriptions for POST values / FW status values:

- `(B)` - boot up
- `(C)` - continuously monitored
- `(F)` - upon executing functions

POST (`nvram_data[1]`)bit encoding:

| | | |
|---|---|---|
| `STS_POST_MEM_FAIL` | `(1L<<0)` | `// (B) Memory test failed` |
| `STS_POST_EEPROM_FAIL` | `(1L<<1)` | `// (B) E2PROM read failed` |
| `STS_POST_LAYER_FAILED` | `(1L<<2)` | `// (C) Layer failure` |
| `STS_POST_FLASH_FAILED` | `(1L<<3)` | `// (B) Flash checksum error` |
| `STS_POST_SDCARD_FAILED` | `(1L<<4)` | `// (C) SD Card is not present` |
| `STS_POST_DC24` | `(1L<<5)` | `// (C) DC->24 layer failed` |
| `STS_POST_DCCORE` | `(1L<<6)` | `// (C) Core voltages problem` |
| `STS_POST_BUSTEST_FAILED` | `(1L<<7)` | `// (B) bus test failed`<br>`//     (hw_test.c)` |
| `STS_POST_BUSFAIL_DATA` | `(1L<<8)` | `// (B) Bus test failed on data`<br>`//     test` |
| `STS_POST_BUSFAIL_ADDR` | `(1L<<9)` | `// (B) Bus test failed on address`<br>`//     test` |
| `STS_POST_OVERHEAT` | `(1L<<10)` | `// (C) Overheat detected` |
| `STS_POST_OVERCURRENT` | `(1L<<11)` | `// (C) Layer DC/DC reported`<br>`//     overcurrent` |
| `STS_POST_RESET_FAILED` | `(1L<<12)` | `// (B) Unit hasn't been properly`<br>`//     reset` |
| `STS_POST_UNRECOG_LAYER` | `(1L<<13)` | `// (B) Layer or backplane cannot be`<br>`//     recognized by the firmware` |

FW status (`nvram_data[2]`)bit encoding:

| | | |
|---|---|---|
| `STS_FW_CLK_OOR` | `(1UL<<0)` | `// (F) Clock out of range`<br>`//     (IOM also)` |

```
•  STS_FW_SYNC_ERR            (1UL<<1)    // (C) Synchronization interface
                                          //     error (IOM also)
•  STS_FW_CHNL_ERR            (1UL<<2)    // (F) Channel list is incorrect
•  STS_FW_BUF_SCANS_PER_INT   (1UL<<3)    // (F) Too few samples per
                                          //     interrupt - interrupt rate
                                          //     is too high
•  STS_FW_BUF_SAMPS_PER_PKT   (1UL<<4)    // (F) Buf setting error:
                                          //     samples/packet
•  STS_FW_BUF_RING_SZ         (1UL<<5)    // (F) Buf setting error:
                                          //     FW buffer ring size
•  STS_FW_BUF_PREBUF_SZ       (1UL<<6)    // (F) Buf setting error:
                                          //     Pre-buffering size
•  STS_FW_BAD_CONFIG          (1UL<<7)    // (F) Layer cannot operate in
                                          //     current config
•  STS_FW_BUF_OVER            (1UL<<8)    // (C) Firmware buffer overrun
•  STS_FW_BUF_UNDER           (1UL<<9)    // (C) Firmware buffer underrun
•  STS_FW_LYR_FIFO_OVER       (1UL<<10)   // (C) Layer FIFO overrun
•  STS_FW_LYR_FIFO_UNDER      (1UL<<11)   // (C) Layer FIFO underrun warning
•  STS_FW_EEPROM_FAIL         (1UL<<12)   // (BF) Layer EEPROM failed
•  STS_FW_GENERAL_FAIL        (1UL<<13)   // (C) Layer general failure
•  STS_FW_ISO_TIMEOUT         (1UL<<14)   // (C) Isolated part reply timeout
•  STS_FW_FIR_GAIN_ERR        (1UL<<15)   // (F) Sum of FIR coeffs is not
                                          //     correct
•  STS_FW_OUT_FAIL            (1UL<<16)   // (C) Output CB tripped or
                                          //     over-current
•  STS_FW_IO_FAIL             (1UL<<17)   // (C) Messaging I/O failed
                                          //     (5xx layers)
•  STS_FW_NO_MEMORY           (1UL<<18)   // (C) Error with memory
                                          //     allocation
•  STS_FW_BAD_OPER            (1UL<<19)   // (F) Operation wasn't performed
                                          //     properly
•  STS_FW_LAYER_ERR           (1UL<<20)   // (F) Layer returned unexpected
                                          //     value
•  STS_FW_OVERLOAD            (1UL<<21)   // (C) CPU or Ethernet is starved
                                          //     out of time or told to
                                          //     free the wire

•  STS_FW_SD_CARD_BUSY        (1UL<<29)   // (C) SD card subsystem is
                                          //     busy (e.g. format, speed
                                          //     test)
•  STS_FW_CONFIG_DONE         (1UL<<30)   // (F) Configuration is completed
                                          //     (not an error)
•  STS_FW_OPER_MODE           (1UL<<31)   // (F) Layer entered operation
                                          //     mode successfully
```

**Note**:

POST status is expected to display 0x0, indicating no errors.

FW status is expected to display 0xC0000000, which means configuration has successfully completed and devices entered operation mode.

### *2.2.56 DqCmdIoctl*

**Syntax:**

`int DqCmdIoctl(int Iom, pDQIOCTL IoIn, pDQIOCTL IoOut)`

**Command:**

`DQCMD_IOCTL` (0x198), Send I/O Command Directly to Device

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQIOCTL IoIn` | input ioctl data |
| `pDQIOCTL IoOut` | buffer for output data |

**Output:**

| | |
|---|---|
| `pDQIOCTL IoOut` | output data |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

`DQCMD_IOCTL` is a multi-purpose command. The IOM schedules its execution rather than performing it immediately because the code executed by firmware for this command may include hardware delays.

This function issues an `ioctl()` request to the specified device driver on IOM. Both input and output packets have the same `DQIOCTL` structure.

```
/* DQCMD_IOCTL */
typedef struct {
    uint8  dev;     // device number
    uint8  ss;      // subsystem
    uint32 cmd;     // ioctl command
    uint8  arg[];   // arguments (var size)
} DQIOCTL, *pDQIOCTL;
```

Currently, there are few IOCTL commands (`IoIn->cmd`, `IoOut->cmd`) defined.

```
// DQCMD_IOCTL commands
#define DQIOCTL_CVTCHNL     (1)     // convert channel
#define DQIOCTL_SETPARAM    (2)     // set arbitraty parameter
                                    //   defined in arg[]
#define DQIOCTL_GETPARAM    (3)     // get arbitraty parameter
                                    //   defined in arg[]
#define DQIOCTL_SETFILTER   (4)     // force filter to set new values
#define DQIOCTL_SIGROUTING  (5)     // set NIS<->IS and SYNCx routing
```

`DQIOCTL_CVTCHNL` is used throughout all layers. For analog input layers, this command has a channel list (`uint32`) as an argument and returns converted data:

AI-201 data size: `uint16`, raw
AI-225 data size: `uint32`, raw
AI-205 data size: `uint32`, raw
AO-302 data size: `uint16`, raw

An analog output layer (AO-302) accepts uint16 values of all eight channels in an `IoIn->arg`/`IoOut->arg` array.

An AI-205 also executes `DQIOCTL_SETFILTER` command and loads stored values (using `DQCMD_WRFIFO` command) into an FIR[3] filter.

**Note:**

None

### 2.2.57 DqCmdGetCapabilities

**Syntax:**

```
int DqCmdGetCapabilities(int Iom, uint8 Layer, int *moredata,
char *info)
```

**Command:**

`DQCMD_GETCAPS` (0x190), Returns capabilities of the requested layer

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint8 Layer` | layer ID |
| `int *moredata` | pointer for more data flag |
| `char *info` | buffer (no less than `DQ_MAX_PKT_SIZE` bytes) |

**Output:**

| | |
|---|---|
| `int *moredata` | more data flag |
| `char *info` | device capabilities information as text data |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function returns information about a layer's capabilities, which is described with text data. If the capabilities data cannot fit in one packet, the `moredata` flag is set. In this case, the command should be sent repeatedly to get all of the data, until `moredata` returns FALSE. Subsequent packets retrieve additional pieces of information, which can be concatenated to the original string using `strcat`(). A call with different or invalid layer number resets the packet counter and data will be retrieved from the beginning of the text.

**Note:**

None

---

[3] Finite Input Response digital filter.

### 2.2.58 DqCmdInitIOM

**Syntax:**

`int DqCmdInitIOM(int Iom, uint8 Layer, uint32 ParamID, uint32 *Data)`

**Command:**

`DQCMD_INITIOM` (0x194), Sets up initial parameters for IOM and layers

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint8 layer` | layer ID |
| `uint32 ParamID` | Parameter ID |
| `uint32 *Data` | Pointer to parameter data |

**Output:**

| | |
|---|---|
| `uint32 *Data` | Returns previous value for parameter |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the command is processed successfully |
| `DQ_BAD_PARAMETER` | if an input parameter was invalid; or if the IOM was not in `DQ_IOMODE_CFG` mode, see `DqCmdSetMode()` |
| Other negative values | low level IOM error |

**Description:**

This function is used to set up multiple parameters for different parts of the IOM. The parameters can be represented in the format required by parameter type. The following parameters are defined:

| `param_id` | Value | Format | Definition |
|---|---|---|---|
| `DQ_LNPRM_MODID` | `0x101` | `uint16` | IOM model ID (DQPARAM) |
| `DQ_LNPRM_MODOPT` | `0x102` | `uint16` | IOM model option (DQPARAM) |
| `DQ_LNPRM_IOMSN` | `0x103` | `uint32` | IOM serial number (DQPARAM) |
| `DQ_LNPRM_IOMMFG` | `0x104` | `uint32` | 0xddmmyyyy<br>IOM manufacturing date (DQPARAM) |
| `DQ_LNPRM_IOMFRQ` | `0x105` | `uint32` | Base frequency, Hz (DQPARAM) |
| `DQ_LNPRM_TICK` | `0x106` | `uint32` | OS Tick size (may be ignored by OS) |
| `DQ_LNPRM_PERIOD` | `0x107` | `uint32` | Periodic tick size (may be ignored by OS) |
| `DQ_LNPRM_WDDLY` | `0x108` | `uint32` | Watchdog timer reset delay |
| `DQ_LNPRM_TIME` | `0x109` | `uint32` | Current layer time |
| `DQ_LNPRM_LNID` | `0x201` | `uint16` | Layer ID |
| `DQ_LNPRM_LNOPT` | `0x202` | `uint16` | Layer option |

| | | | |
|---|---|---|---|
| `DQ_LNPRM_LNSN` | `0x203` | `uint32` | Layer serial number |
| `DQ_LNPRM_TOTAL` | `0x204` | `uint16` | Total EEPROM size |
| `DQ_LNPRM_LNMFG` | `0x205` | `uint32` | 0xddmmyyyy<br>Layer manufacturing date |
| `DQ_LNPRM_LNCAL` | `0x206` | `uint32` | 0xddmmyyyy<br>Layer calibration date |
| `DQ_LNPRM_LNEXP` | `0x207` | `uint32` | 0xddmmyyyy<br>Layer calibration expiration date |
| | | | |
| `LNPRM_NOCHANGE` | 0xF00 | | Do not apply changes, test only |
| `LMPRM_BYOFFS` | 0xF01 | | Write data by offset to the layer EEPROM.<br>The data should be specified as follows:<br>u16 offs, u16 size, u8[] data |

The function doesn't write anything into EEPROM. Instead, it writes into the common part of EEPROM copied in the RAM at initialization mode. The location and structure of the common part is the same for all layers. You have to call `DQCMD_SAVEPRM` with proper device identification to flash RAM data into the layer EEPROM or IOM flash sector. If you need to change something layer-specific, call this function with `LMPRM_BYOFFS` code and specify the offset of the data to be written.

**Note:**

None

### *2.2.59 DqCmdSetTrigger*

**Syntax:**

`int DqCmdSetTrigger(int Iom, pDQSETTRIG pDQSetTrig, uint32 *entries)`

**Command:**

`DQCMD_SETTRIG` (0x1A0), Set trigger parameters

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `pDQSETTRIG pDQSetTrig` | trigger settings (pointer to an array of structures) |
| `uint32 *entries` | number of entries in `pDQSetTrig` |

**Output:**

| | |
|---|---|
| `uint32 *entries` | actual number of entries sent |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function sets the triggering parameters for one or more devices.

`DQSETTRIG` structure is defined as follows:

```
typedef struct {
    uint8 dev;          // device
    uint8 ss;           // subsystem
    uint8 ch;           // channel
    uint8 mode;         // trigger mode (AND or OR)

    // Start trigger
    uint32 trigtypeS;   // trigger type (enable, edge or mask)
    union {
        float levelS;       // level
        uint32 maskS;
    } uS;

    // stoP trigger
    uint32 trigtypeP;   // trigger type (enable, edge or mask)
    union {
        float levelP;       // level
        uint32 maskP;
    } uP;

    float hyster;      // hysteresis for both triggers
    int prescans;      // pre-trigger scans
    int postscans;     // post-trigger scans, -1 = continue operations
} DQSETTRIG, *pDQSETTRIG;
```

**Note:**

field `mode` in the structure defines the meaning of other fields.

```
// Definitions for triggering and synchronization interface
#define DQ_TRIGGER_SET_OR   (1L<< 5)    // set alternative condition
                                        //    (OR trigger)
#define DQ_TRIGGER_SET      (1L<< 4)    // set trigger
                                        //  (or add additional AND
                                        //   condition)
#define DQ_TRIGGER_ONCE     (1L<< 3)    // trigger only once
#define DQ_TRIGGER_RESET    (1L<< 2)    // reset trigger to default
#define DQ_TRIGGER_STOP     (1L<< 1)    // issue stop trigger
#define DQ_TRIGGER_START    (1L<< 0)    // issue start trigger

// For layers with logic >= 02.11.1A
#define DQ_TRIGGER_TS_CL (0x7) //trigger will sync timestamp
                               //   to first CL clock data
#define DQ_TRIGGER_TS_CV (0x6) //trigger will sync timestamp
                               //   to first CV clock data
#define DQ_TRIGGER_TS_NO (0x5) //normal operation, trigger
                               //   will not sync timestamp
```

### 2.2.60 *DqCmdSetLock*

**Syntax:**

`int DqCmdSetLock(int Iom, uint8 Mode, char *Password, uint32 *IP)`

**Command:**

`DQCMD_SETLOCK`(0x1C0), Set/clear lock, or query lock status

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint8 Mode` | function mode: (lock/unlock/check/checkdiag/diag)<br>(see Note below) |
| `char *Password` | password string: ignored (and can be NULL) if `Mode` is `DQSETLOCK_CHECK,DQSETLOCK_DIAG` or `DQSETLOCK_CHECKDIAG` (see Note below) |
| `uint32 *IP` | pointer to receive the IP address of the locking host if `Mode` is `DQSETLOCK_CHECK` or `DQSETLOCK_DIAG`<br>`or` pointer to receive the true/false state if `Mode` is `DQSETLOCK_CHECKDIAG` (see Note below) |

**Output:**

| | |
|---|---|
| `uint32 *IP` | the IP address of the locking host if `Mode` is `DQSETLOCK_CHECK` or `DQSETLOCK_DIAG`<br>`or` pointer to receive the true/false state if `Mode` is `DQSETLOCK_CHECKDIAG` (see Note below) |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_BAD_PARAMETER_3` | `Mode` is `DQSETLOCK_CHECK` or `DQSETLOCK_CHECKDIAG` and `IP` is NULL |
| `DQ_SUCCESS` | if the command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

Locks the IOM to the current host, unlocks locked IOM, or retrieves IP address of locking host. For locking and unlocking, the `Password` parameter must contain the IOM's user level password.

The following `Mode` constants are defined:

```
#define DQSETLOCK_LOCK       0     // Lock IOM to host
#define DQSETLOCK_UNLOCK     1     // Unlock IOM
#define DQSETLOCK_CHECK      2     // Get locking host IP
#define DQSETLOCK_CHECKDIAG* 3     // Check whether or not chassis
                                   //   is in diagnostics mode
                                   //   (see Note below)
#define DQSETLOCK_DIAG*      4     // Switch into diagnostics mode
                                   //   (see Note below)
```

**Note:**

*Note that diagnostic modes set with `DQSETLOCK_CHECKDIAG` and `DQSETLOCK_DIAG` are only available with RACK and 1G Cube chassis.

### 2.2.61 DqCmdResetTimestamp

**Syntax:**

```
int DqCmdResetTimestamp(int Iom, int dev_mask, int resolution);
```

**Command:**

`DQCMD_RSTTS` (0x111), Set/reset timestamp generators

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `int dev_mask` | Bitmask of the devices in IOM to reset/set timestamp for |
| `int resolution` | Timestamp source or Divider for 66MHz timestamp generator |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_BAD_PARAMETER` | `Mode` is `DQSETLOCK_CHECK` and `IP` is NULL |
| `DQ_SUCCESS` | if the command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

<`dev_mask`> != 0 – resets the timestamp counter on the selected layers to zero and loads the `resolution` value to the timestamp divider. For example for 100us timestamp with 66MHz base clock the value should be (6600-1). Written timestamp resolution is then stored for future use with the selected device.

<`dev_mask`> == 0 – resets the timestamp counter on all layers installed using broadcast write. Keeps resolution unchanged.

**Note:**

User can pass any timestamp resolution value. The timestamp generator is a 32-bit counter running from a 66MHz clock by default; each layer has its own DQ_xxxx_BASE constant, some layers may have different base clocks.

Logic 0x010210D6 and newer let you select the timestamp source in the upper four bits of `resolution`. All current DQ_LN_1xxx_TIMESTAMP constants (found in powerdna.h) use up to 24 bits in 32-bit word and as a result programs default 66MHz base clock as a clock source.

```
    #define DQL_TMRCFG_TSTS3        (1L<<31)
    #define DQL_TMRCFG_TSTS0        (1L<<28)
    #define DQL_TMRCFG_TSTS(N)      ((N)<<28) // where N is one of the
DQL_TMRCFG_TSTS_... constants
```

```
    // SOURCE - timestamp source selector
    DQL_TMRCFG_TSTS_66M       0     // 0    0000 - 66MHz (default)
    DQL_TMRCFG_TSTS_TMR0      1     // 1    0001 - TMR0 (pulse)
    DQL_TMRCFG_TSTS_TMR1      3      // 3    0011 - TMR1 (pulse)
    DQL_TMRCFG_TSTS_IEXT0     4      // 4    0100 - iso_ext0
    DQL_TMRCFG_TSTS_IEXT1     5      // 5    0101 - iso_ext1
    DQL_TMRCFG_TSTS_IINT0     8      // 8    1000 - "Internal" source 0
    DQL_TMRCFG_TSTS_IINT1     9      // 9    1001 - "Internal" source 1
    DQL_TMRCFG_TSTS_SYNC0    12      // 12    1100 - External SYNC line 0
    DQL_TMRCFG_TSTS_SYNC1    13      // 13    1101 - External SYNC line 1
    DQL_TMRCFG_TSTS_SYNC2    14      // 14    1110 - External SYNC line 2
    DQL_TMRCFG_TSTS_SYNC3    15      // 15    1111 - External SYNC line 3
```

Therefore it is possible, for example, to set the IRIG-650 as a high-prescision timestamp generator on SYNC line 0, and then set various layers to use SYNC line 0 for their timestamp source with the call:

`DqCmdResetTimestamp(hd, 1L<<DEVN, DQL_TMRCFG_TSTS(DQL_TMRCFG_TSTS_SYNC0)).`

Timestamp resolution is defined as a divider for 66MHz bus clock as follows:

```
// Timestamps (66MHz divider)
// resolution parameter for DqCmdResetTimestamp() function
#define DQ_LN_01us_TIMESTAMP        (6-1)       // 0.090909us
#define DQ_LN_1us_TIMESTAMP         (66-1)      // 1us
#define DQ_LN_10us_TIMESTAMP        (660-1)     // 10us - default value for most AI layers
#define DQ_LN_100us_TIMESTAMP       (6600-1)    // 100us
#define DQ_LN_1ms_TIMESTAMP         (66000-1)   // 1ms
#define DQ_LN_10ms_TIMESTAMP        (660000-1)  // 10ms
#define DQ_LN_100ms_TIMESTAMP       (6600000-1) // 100ms
#define DQ_LN_1s_TIMESTAMP          (66000000-1)// 1s

#define DQ_LN_MASK_TIMESTAMP        (0x7ffffff) // the timestamp resolution cannot be larger
than 2 seconds
```

## *2.2.62 DqGetLastDqPktTimestamp*

**Syntax:**

```
int DAQLIB DqGetLastDqPktTimestamp(int Iom, uint32 *timestamp, uint32*
counter)
```

**Command:**

This call doesn’t send any command

**Input:**

| | |
|---|---|
| `int Iom` | Handle to the IOM returned by `DqOpenIOM()` |

**Output:**

| | |
|---|---|
| `uint32 *timestamp` | Last received timestamp associated with the handle (uint16* dqTimestamp) |
| `uint32* counter` | Last received counter associated with the handle (uint16 dqCounter) |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_SUCCESS` | if the command is processed successfully |

**Description:**
From 4.11.2 build PowerDNA software received ability to return timestamp and packet counter associated with last executed command. When IOM executes command it stores timestamp from that layer at the time of execution. For example, timestamp is written when analog output is updated.

**Note:**
Use `DqCmdResetTimestamp()`to reset and select resolution for the layer of interest. Please notice since DaqBIOS packet allocates only 16-bit for timestamp and for counter the resolution of the timestamp should be selected accordingly. For example, with 1us resolution timestamp field rolls over in 65ms.

There are two main uses for this function – to keep track when an output event took place relatively to the input timestamp and also to find out execution time by calling function and then immediately recording timestamps.

### *2.2.63 DqCmdReceiveEvent*

**Syntax:**

```
int DqCmdReceiveEvent(int handle, uint32 flags, int timeout_us,
pDQEVENT* event_buf, int* size)
```

**Command:**

This call doesn't send any commands.

**Input:**

| | |
|---|---|
| `int handle` | Async handle to the IOM received from `DqAddIOMPort()` |
| `int flags` | Define behavior of the function |
| `int timeout_us` | Event timeout in micro-seconds |

**Output:**

| | |
|---|---|
| `pDQEvent* event_buf` | Structure describing the received asynchronous event |
| `int* size` | Size of the event structure in bytes |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_NOT_ENOUGH_ROOM` | the size indicated by the size parameter is not big enough to hold the received message |
| `DQ_TIMEOUT_ERROR` | No event was received during time out duration |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_BAD_PARAMETER` | One of the input parameters is invalid |
| `DQ_SUCCESS` | if the command is processed successfully |
| Other negative values | low level IOM error |

**Description:**
`DqCmdReceiveEvent()` waits for asynchronous event packets sent by the IOM to the host PC. It returns `DQ_SUCCESS` if an event is successfully received or `DQ_TIMEOUT_ERROR` if the timeout expires before any event was received. Events must be configured first with one of the `DqAdvxxxConfigEvents()` calls.

`<flags>`

- Reserved for future use in PowerDNA mode
- In UEIPAC mode, set to device number (DEVN) of the IO board/layer where the events will be received from

`<event_buf>`

- Output parameter that points to an internal structure consisting of information about the received event: device that emitted the event, type of the event, size of the data attached to the event (if any). Refer to the `DqAdvxxxConfigEvents()` API you are using for more information about the `pDQEvent` structure for the board for your application.

`<size>`

- Output parameter contains the size of the `event_buf` event structure in bytes. You must copy the event structure to a local buffer before calling `DqCmdReceiveEvent()` again.

**Notes:**
To additionally retrieve a packet ID (`dqCounter`) to verify the sequence of the received event, refer to the `DqCmdReceiveEventPkt()` API.

### *2.2.64 DqCmdReceiveEventPkt*

**Syntax:**

```
int DqCmdReceiveEventPkt(int handle, uint32 flags, int timeout_us,
pDQEVENT* event_buf, int* size, pDQPKT* dqpkt)
```

**Command:**

This call doesn't send any command

**Input:**

| | |
|---|---|
| `int handle` | Async handle to the IOM received from `DqAddIOMPort()` |
| `int flags` | Reserved for future use |
| `int timeout_us` | Event timeout in micro-seconds |

**Output:**

| | |
|---|---|
| `pDQEvent* event_buf` | Structure describing the received asynchronous event |
| `int* size` | Size of the event structure in bytes |
| `pDQPKT* dqpkt` | Pointer to the received packet. See description below for `pDQPKT` packet structure |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_NOT_ENOUGH_ROOM` | the size indicated by the size parameter is not big enough to hold the received message |

| | |
|---|---|
| `DQ_TIMEOUT_ERROR` | No event was received during time out duration |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_BAD_PARAMETER` | One of the input parameters is invalid |
| `DQ_SUCCESS` | if the command is processed successfully |
| Other negative values | low level IOM error |

**Description:**

This function waits for an asynchronous event packet sent by IOM to host and additionally provides packet information for verifying the sequence of the received event. `DqCmdReceiveEventPkt()` is the same function as `DqCmdReceiveEvent()` with the inclusion the `pDQPKT` packet structure.

`DqCmdReceiveEventPkt()` returns `DQ_SUCCESS` if an event is successfully received or `DQ_TIMEOUT_ERROR` if the `timeout_us` expires before any event was received.

Events must be configured first with one of the `DqAdvxxxConfigEvents()` calls.

`DqCmdReceiveEventPkt()` parameters include the following:

- `<event_buf>`: output parameter points to an internal structure consisting of information about the received event: device that emitted the event, type of the event, size of the data attached to the event (if any). Refer to the `DqAdvxxxConfigEvents()` API you are using for more information about the `pDQEvent` structure for the board you are using.
- `<size>`: output parameter contains the size of the `event_buf` event structure in bytes. You must copy the event structure to a local buffer before calling `DqCmdReceiveEvent()` again.
- `<dqpkt>`: packet structure contains the `dqCounter` element, which increments with each packet sent.

  Users can monitor `dqCounter` to confirm the sequence of the received events.

  The `DQPKT` structure is

```
typedef struct DQPKT{
    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[DQ_FLEX_ARRAY];    /* Data - bytes variable size */
} DQPKT, * pDQPKT;
```

  `dqCounter` increments with each packet sent from the I/O board you configured with `DqAdvxxxConfigEvents().` The event packet increments from. 1..65535 and then wraps back to 1.

  You must copy the `DQPKT` structure to a local buffer before calling `DqCmdReceiveEventPkt()` again and converting the `dqCounter` data from big endian (network data representation) to little endian:

```
   localdqCounterBuf = ntohs(dqpkt->dqCounter);
```

**Notes:**

# 3 High-Level API

The high-level API is based on the use of DQEngine – a programming environment that takes care of handling streams of data and that performs error correction. The high-level API provides two mechanisms: Advanced Circular Buffer (ACB) and Direct Data Mapping (DMap).

Not all layers support ACB or DMap functions. Therefore, it is important to call `DqAcbIsSupported()` and `DqDmapIsSupported()` to verify that operations exist for the layer.

This section describes common ACB and DMap functions, dedicated ACB functions, dedicated DMap functions and common ACB and DMap control and event functions.

## 3.1 Common ACB, DMap, and Msg functions

These are functions common to both ACB and DMap mechanisms.

### 3.1.1 DqAcbIsSupported

**Syntax:**

```
int DqAcbIsSupported(int iom, uint32 devn, uint32 ss, int
*supported)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int iom` | handle to IOM received from `DqOpenIOM()` |
| `uint32 devn` | layer inside the IOM |
| `uint32 ss` | subsystem of layer |
| `int *supported` | pointer to Boolean to receive value |

**Output:**

| | |
|---|---|
| `int *supported` | returns `TRUE` if the IOM/device/subsystem supports ACB operation, `FALSE` if not |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_SUCCESS` | successful completion |
| other negative values | other error |

**Description:**

This function verifies that the IOM/device/subsystem supports ACB operation.

**Note:**

None

### *3.1.2 DqDmapIsSupported*

**Syntax:**

```
int DqDmapIsSupported(int iom, uint32 devn, uint32 ss, int
*supported)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int iom` | Handle to IOM received from `DqOpenIOM()` |
| `uint32 devn` | layer inside the IOM |
| `uint32 ss` | subsystem of layer |
| `int *supported` | pointer to receive result |

**Output:**

| | |
|---|---|
| `int *supported` | returns `TRUE` if the IOM/device/subsystem supports DMap operation, `FALSE` if not |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid iom handle |
| `DQ_SUCCESS` | successful completion |
| other negative values | other error |

**Description:**

This function verifies that the IOM/device/subsystem supports DMap operation.

**Note:**

None

### *3.1.3 DqVmapIsSupported*

**Syntax:**

```
int DqVmapIsSupported(int iom, uint32 devn, uint32 ss, int
*supported)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int iom` | Handle to IOM received from `DqOpenIOM()` |
| `uint32 devn` | layer inside the IOM |
| `uint32 ss` | subsystem of layer |
| `int *supported` | pointer to receive result |

**Output:**

| | |
|---|---|
| `int *supported` | returns `TRUE` if the IOM/device/subsystem supports VMap operation, `FALSE` if not |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_SUCCESS` | successful completion |
| other negative values | other error |

**Description:**

This function verifies that the IOM/device/subsystem supports VMap operation.

**Note:**

None

### *3.1.4 DqMsgIsSupported*

**Syntax:**

```
int DqMsgIsSupported(int iom, uint32 devn, uint32 ss, int
*supported)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int iom` | Handle to IOM received from `DqOpenIOM()` |
| `uint32 devn` | layer inside the IOM |
| `uint32 ss` | subsystem of layer |
| `int *supported` | pointer to receive result |

**Output:**

| | |
|---|---|
| `int *supported` | returns `TRUE` if the IOM/device/subsystem supports Msg operation, `FALSE` if not |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid iom handle |
| `DQ_SUCCESS` | successful completion |
| other negative values | other error |

**Description:**

This function verifies that the IOM/device/subsystem supports Msg operation.

**Note:**

None

### *3.1.5 DqSupportedModes*

**Syntax:**

```
int DqSupportedModes(uint16 lymod, int *supported)
```

**Input:**

| | |
|---|---|
| `uint16 lymod` | Layer/Board Model |

**Output:**

| | |
|---|---|
| `int *supported` | Bitmap of acquisition modes supported by the layer |

**Return:**

| | |
|---|---|
| `DQ_SUCCESS` | successful completion |

**Description:**

This function is used to identify which acquisition modes are supported by the specified layer. The supported bitmap can have the following bits set for showing acquisition mode support:

```
#define DQ_ACQ_MODE_RTVMAP       (1L << 0)
#define DQ_ACQ_MODE_RTDMAP       (1L << 1)
#define DQ_ACQ_MODE_ACB          (1L << 2)
```

**Note:**

None

### *3.1.6 DqStartDQEngine*

**Syntax:**

```
int DqStartDQEngine(uint32 period_us, pDQE *pDqe, pDQEPRM
dqeprm)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `uint32 period_us` | main clock period in microseconds |
| `pDQE *pDqe` | pointer to DQE pointer |
| `pDQEPRM dqeprm` | pointer for DQE parameters (or NULL to use defaults) |

**Output:**

| | |
|---|---|
| `pDQE *pDqe` | pointer to the created DQE instance |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | memory allocation error |
| `DQ_TIMEOUT_ERROR` | timer error |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function initializes DQEngine and prepares it for operation. It:

1. Creates events for thread synchronization.
2. Starts sending threads and set their priorities.
3. Verifies and stores startup parameters.
4. Starts a periodic routine that wakes up threads.

The user can change default parameters by filling the DQEPRM structure and supplying a pointer to it. Set fields for parameters you don’t want to change to NULL.

```
// DQE parameters
typedef struct {
    uint32 *timeout;           // reply wait timeout
    uint32 *retries_async;     // number of retries for asynchronous commands before return an error
    uint32 *retries_receive;   // number of retries while receiving stream before placing filler
    uint32 *retries_send;      // number of retries while sending stream upon dumping packet
    uint32 *max_inbound_packet;  // maximum packet size from IOM
    uint32 *max_outbound_packet; // maximum packet size send to IOM
    uint32 *abort_after;       // when streaming abort operation after number of lost packets
    uint32 *use_protocol;      // (RESERVED)
    uint32 *packets_at_once;   // maximum number of packets to one IOM upon one tick
} DQEPRM, *pDQEPRM;
```

**Note:**

We recommend using default DQE parameters unless there is a need to change settings.

### *3.1.7 DqStopDQEngine*

**Syntax:**

`int DqStopDQEngine(pDQE pDqe)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQE pDqe` | pointer to DQE object |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid `pDqe` value |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function performs DQEngine clean-up: stops main timer routine and kills all listener threads associated with different IOMs..

**Note:**

None

### *3.1.8 DqParamDQEngine*

**Syntax:**

`int DqParamDQEngine(pDQE pDqe, int setparam, pDQEPRM pDqePrm)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQE pDqe` | pointer to DQE object |
| `int setparam` | `TRUE` to set parameters, `FALSE` to read parameters |
| `pDQEPRM dqeprm` | pointer for the parameter structure |

**Output:**

| | |
|---|---|
| `pDQEPRM dqeprm` | pointer to the parameter structure read |

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | `pdqeprm` is NULL |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function can retrieve or change DQEngine parameters after engine is started.

Set `setparam` to `TRUE` to set parameters or set it to `FALSE` to retrieve current parameters.

```
// DQE parameters
typedef struct {
    uint32 *timeout;             // reply wait timeout
    uint32 *retries_async;       // number of retries for asynchronous commands before return an error
    uint32 *retries_receive;     // number of retries while receiving stream before placing filler
    uint32 *retries_send;        // number of retries while sending stream upon dumping packet
    uint32 *max_inbound_packet;  // maximum packet size from IOM
    uint32 *max_outbound_packet; // maximum packet size send to IOM
    uint32 *abort_after;         // when streaming abort operation after number of lost packets
    uint32 *use_protocol;        // switch between DQ_TS and DQ_VT (RESERVED)
    uint32 *packets_at_once;     // maximum number of packets to one IOM upon one tick

} DQEPRM, *pDQEPRM;
```

Set a field to `NULL` if you don’t want to change that parameter.

**Note:**

None

### 3.1.9 DqAdvReadCalData

**Syntax:**

```
int DqAdvReadCalData(int hd, int devn, BYTE **CalData, uint32
*CalSize)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `BYTE **CalData` | pointer to store retrieved calibration data; may be NULL if caller doesn't want to receive data |
| `uint32 *CalSize` | pointer to store size of calibration data retrieved; may be NULL if `CalData` is NULL |

**Output:**

| | |
|---|---|
| `BYTE **CalData` | pointer to calibration data retrieved |
| `uint32 *CalSize` | size of calibration data retrieved |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or does not support calibration data |
| `DQ_BAD_PARAMETER` | `CalSize` is NULL but `CalData` is not NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function retrieves calibration data and stores it inside DLL.

**Note:**

Data is stored in the DLL memory and shouldn't be changed.

## 3.2 Advanced Circular Buffer (ACB) Functions

These are functions specific to ACB mechanisms.

### 3.2.1 DqAcbCreate

**Syntax:**

```
int DqAcbCreate(pDQE pDqe, int iom, uint32 devn, uint32 ss,
pDQBCB *pBcb)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQE pDqe` | pointer to the previously created instance of DQE |
| `int iom` | handle to IOM received from `DqOpenIOM()` |
| `uint32 devn` | layer inside the IOM |
| `uint32 ss` | subsystem of layer |
| `pDQBCB *pBcb` | pointer for BCB structure |

**Output:**

| | |
|---|---|
| `pDQBCB *pBcb` | newly allocated BCB structure |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | memory allocation error |
| `DQ_BAD_PARAMETER` | `NULL` or `0` as a parameter |
| `DQ_BAD_DEVN` | no device at given index, or device does not support ACB mode |
| `DQ_ILLEGAL_HANDLE` | invalid non-`NULL` value for `iom` |
| `DQ_DEVICE_BUSY` | layer already taken |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function allocates a new ACB-type BCB structure, and links it with DQEngine and the IOM specified.

**Note:**

None

### 3.2.2 DqAcbDestroy

**Syntax:**

```
int DqAcbDestroy(pBCB pBcb)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to a previously allocated ACB |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | `pBcb` is NULL |
| `DQ_BAD_PARAMETER` | memory deallocation error |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function destroys a previously allocated ACB.

**Note:**

None

### 3.2.3 DqAcbInitOps

**Syntax:**

```
int DqAcbInitOps(
                    pDQBCB pBcb,
                    uint32 *Config,
                    uint32 *TrigSize,
                    pDQSETTRIG TrigMode,
                    float * fCLClk,
                    float * fCVClk,
                    uint32 *CLSize,
                    uint32 *CL,
                    uint32 *ScanBlock,
                    pDQACBCFG pAcbCfg)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to BCB (points to ACB) |
| `uint32 *Config` | requested configuration (layer specific) |
| `uint32 *TrigSize` | requested number of triggering conditions; can be `NULL` |
| `pDQSETTRIG TrigMode` | requested triggering mode (layer specific); can be `NULL` if `TrigSize` is `NULL` or `0` |
| `float *fCLClk` | CL clock requested; can be `NULL` if `Config` doesn't include `LN_CLCKSRC0` |
| `float *fCVClk` | CV clock requested; can be `NULL` if `Config` doesn't include `LN_CVCKSRC0` |
| `uint32 *CLSize` | requested channel list size |
| `uint32 *CL` | requested channel list |
| `uint32 *ScanBlock` | requested number of scans to handle together (`0` = default) |
| `pDQACBCFG pAcbCfg` | requested buffer parameters |

**Output:**

| | |
|---|---|
| `uint32 *Config` | actual configuration (layer specific) |
| `uint32 *TrigSize` | actual number of triggering conditions; can be `NULL` |
| `pDQSETTRIG TrigMode` | actual triggering mode (layer specific); can be `NULL` if `TrigSize` is `NULL` or `0` |
| `float *fCLClk` | CL clock - actual; can be null if `Config` doesn't include `LN_CLCKSRC0` |
| `float *fCVClk` | CV clock - actual; can be null if `Config` doesn't include `LN_CVCKSRC0` |

| | |
|---|---|
| `uint32 *CLSize` | actual channel list size |
| `uint32 *CL` | actual channel list |
| `uint32 *ScanBlock` | actual number of scans to handle together (`0` = default) |
| `pDQACBCFG pAcbCfg` | actual buffer parameters |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_PARAMETER` | pBcb is `NULL` or is not an ACB, or `CLSize` is too big |
| `DQ_IOM_ERROR` | IOM reports command execution error |
| `DQ_SEND_ERROR` | cannot send packet |
| `DQ_TIMEOUT_ERROR` | IOM reply wasn't received within timeout period |
| `DQ_DEVICE_BUSY` | device is in operating mode |
| `DQ_NO_MEMORY` | memory allocation error |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function sets up layer configuration for ACB operation.
At the time of calling, BCB should have been allocated using `DqAcbCreate().`

**`Config`:**

Following configuration flags can be used with ACB operations:

```
#define DQ_LN_TMREN      (1L<<11)  // enable layer periodic timer
#define DQ_LN_IRQEN      (1L<<10)  // enable layer irqs
#define DQ_LN_PTRIGEDGE1 (1L<<9)   // stop trigger edge MSB
#define DQ_LN_PTRIGEDGE0 (1L<<8)   // stop trigger edge: 00 - software, 01 - rising,
                                   // 02 - falling
#define DQ_LN_STRIGEDGE1 (1L<<7)   // start trigger edge MSB
#define DQ_LN_STRIGEDGE0 (1L<<6)   // start trigger edge: 00 - software, 01 - rising,
                                   // 02 - falling
#define DQ_LN_CVCKSRC1   (1L<<5)   // CV clock source MSB
#define DQ_LN_CVCKSRC0   (1L<<4)   // CV clock source 01 - SW, 10 - HW, 11 -EXT
#define DQ_LN_CLCKSRC1   (1L<<3)   // CL clock source MSB
#define DQ_LN_CLCKSRC0   (1L<<2)   // CL clock source 01 - SW, 10 - HW, 11 -EXT
#define DQ_LN_ACTIVE     (1L<<1)   // "ACT" LED status
#define DQ_LN_ENABLED    (1L<<0)   // enable operations
```

These flags are hardware-oriented. The user must set up either `DQ_LN_TMREN` or `DQ_LN_IRQEN` flags to configure a mechanism to transfer data from the layer hardware buffer into the output queue. See layer documentation for these settings.

Every layer can have additional configuration flags, starting from bit 16. For example, an AO-301 layer has the following flags:

```
// Upper part of the configuration word - AO-301 specific
#define DQ_AO301_POS10     (1L << 19)  // 0..10V
#define DQ_AO301_NEG10     (2L << 19)  // -10..0 V
#define DQ_AO301_BI10      (3L << 19)  // +/-10V
#define DQ_AO301_OFF       (0L << 19)  // DACs off
#define DQ_AO301_ENCOUT    (1L << 18)  // enable output strobe

#define DQ_AO301_MODESCAN  (0L << 16)  // single scan update mode
#define DQ_AO301_MODEFIFO  (1L << 16)  // continuous output with FIFO
#define DQ_AO301_MODECONT  (2L << 16)  // waveform mode - continuous
#define DQ_AO301_MODEWFGEN (3L << 16)  // waveform mode – hardware
```

Most layers have two flags (where `*` is layer model):

```
#define *_MODESCAN  (0L << 16)  // single scan update mode
#define *_MODEFIFO  (1L << 16)  // continuous output with FIFO
```

There are two definitions common to all models:

```
#define DQ_FIFO_MODESCAN  (0L << 16)  // single scan update mode
#define DQ_FIFO_MODEFIFO  (2L << 16)  // continuous acquisition with FIFO
```

The first mode is selected for DMap operation and maps one or more scans into the physical memory of the device. The second mode is the streaming mode. This mode is selected when the layer streams to the ACB.

The user can select start and stop triggers. These settings are applied to external trigger lines connected to a CM-1 layer.

The user may also select the `DQ_LN_ACTIVE` bit to light-up “STS” – the status LED on the layer.

Flag `DQ_LN_ENABLED` – should be selected (library sets it anyway).

**`TrigMode:`**

Allocate and pass a pointer to the following structure to set up hardware triggering parameters:

```
typedef struct {
    uint8 dev;          // device
    uint8 ss;           // subsystem
    uint8 ch;           // channel
    uint8 mode;         // trigger mode (AND or OR)

    // Start trigger
    uint32 trigtypeS;   // trigger type (enable, edge or mask)
    union {
        float levelS;       // level
        uint32 maskS;
    } uS;

    // stoP trigger
    uint32 trigtypeP;   // trigger type (enable, edge or mask)
    union {
        float levelP;       // level
        uint32 maskP;
    } uP;

    float hyster;      // hysteresis for both triggers
    int prescans;      // pre-trigger scans
    int postscans;     // post-trigger scans, -1 = continue operations
} DQSETTRIG, *pDQSETTRIG;
```

Passing NULL switches the device into software-triggering mode. One can select one or more entries in a triggering table. `mode` specifies whether or not to treat this trigger entry. `mode` applies to the following entry. Because the AND operation has higher priority than the OR operation, a logic expression is calculated accordingly. For example, you can specify a trigger expression such as: T0 AND T1 AND T2 OR T1 AND T2 AND T4, where T0…T4 are trigger conditions. This expression is equal to (T0^ T1^ T2) v (T1 ^ T2 ^ T4).

**`CL`:**

Channel list entries are stored in the following format:

```
typedef struct {
    uint8 dev;     // device number
    uint8 ss;      // subsystem
    uint32 entry;  // channel list entry
} DQSETCL, *pDQSETCL;
```

The whole channel list for a device must fit in one packet. Thus, the maximum number of channel list entries is 85 entries (maximum physical channel list on layer is 64 entries).

Output data: entries - The number of entries actually sent.

The channel list (entry) has following format:

| Bits | [31..15] | [14..12] | [11..8] | [7..0] |
|---|---|---|---|---|
| Description | Flags | Reserved | Gain | Channel number |

The following are values for the Flags bits:

```
// Channel list entries definition - lower 16 bits are reserved for channel number
// gain and special, module-specific settings
#define DQ_LNCL_NEXT    (1UL<<31)    // channel list has next entry
#define DQ_LNCL_INOUT   (1UL<<30)    // (reserved for future use)
#define DQ_LNCL_SS1     (1UL<<29)    // (reserved for future use)
#define DQ_LNCL_SS0     (1UL<<28)    // (reserved for future use)
#define DQ_LNCL_IRQ     (1UL<<27)    // (reserved for future use)
#define DQ_LNCL_NOWAIT  (1UL<<26)    // (reserved for future use)
#define DQ_LNCL_SKIP    (1UL<<25)    // (reserved for future use)
#define DQ_LNCL_CLK     (1UL<<24)    // (reserved for future use)
#define DQ_LNCL_CTR     (1UL<<23)    // (reserved for future use)
#define DQ_LNCL_WRITE   (1UL<<22)    // write to the channel but not update
#define DQ_LNCL_UPDALL  (1UL<<21)    // update all written channels
#define DQ_LNCL_TSRQ    (1UL<<20)    // (reserved for future use)
#define DQ_LNCL_SLOW    (1UL<<19)    // slow down operation
#define DQ_LNCL_DIO     (1UL<<18)    // write/read DIO
#define DQ_LNCL_RSVD1   (1UL<<17)    // (reserved for future use)
#define DQ_LNCL_RSVD0   (1UL<<16)    // (reserved for future use)

#define DQ_LNCL_DIFF    (1UL<<15)    // differential mode
#define DQ_LNCL_GAIN(G) ((G & 0xf)<<8)   // set gain
#define DQ_LNCL_TIMESTAMP (0xff)         // timestamp entry
                                         //   (when used as a channel #)
```

**`pAcbCfg`:**

Allocate and pass pointer to ACB structure. Fields `samplesz`, `scansz`, `framesize`, `frames`, `dirflags`, `mode` and `dirflags` must be specified.

```
// ACB description
typedef struct {
    uint32 samplesz;   // raw sample size, bytes
    uint32 valuesz;    // converted value size, bytes
    uint32 scansz;     // scan size, samples/values
    uint32 framesize;  // number of scans in the frame, max
    uint32 frames;     // frames in the buffer
    uint32 ppevent;    // packets per DQ_ePacketDone event
    uint32 mode;       // mode of operations: Single, Cycle, Recycled, error handling
    uint32 dirflags;   // transfer direction and additional flags
    uint32 maxpktsize; // how much data to accumulate in the packet before
                       //    sending (0=default)
    uint32 hwbufsize;  // how much data to keep on the cube (0 = default)
    uint32 hostringsz; // number of packets in the host ring buffer (0 = default)
    uint32 wtrmark;    // percent of the ring buffer queue packets kept in case
                       //     IOM reports an error
    double eucoeff;    // engineering unit coefficient to multiply voltage data by
    double euoffset;   // engineering unit offset
    double (*euconvert)(uint32 chan, double value); // callback to convert from V to
EU
} DQACBCFG, *pDQACBCFG;
```

Set up `framesize` in number of scans per frame.
`dirflags` is a combination of a direction constant with a data representation constant. Possible options are:

```
// Defines for data conversion function (dirflags)
// direction
#define DQ_ACB_DIRECTION_INPUT     0x0     // dir-in
#define DQ_ACB_DIRECTION_OUTPUT    0x1     // dir-out

// data conversion type
#define DQ_ACB_DATA_SINGLE         0x100   // data in the ACB is <float>
#define DQ_ACB_DATA_DOUBLE         0x200   // data in the ACB is <double>
#define DQ_ACB_DATA_RAW            0x300   // data in the ACB is Raw
#define DQ_ACB_DATA_EUNITS         0x400   // double in engineering units
#define DQ_ACB_DATA_ENHANCED       0x800   // enhanced resolution (18 bit in 32-bit format
                                           // if converter has 18-bit native resolution)
// data processing type
#define DQ_ACB_DATA_TSCOPY         0x1000  // copy timestamp into ACB is available
#define DQ_ACB_NOCALIBRATION       0x2000  // do not perform software calibration
// additional dirflags flag in versions 4.10.0.28+ of DAQLIB+firmware.
// enable or disable the outputting of eeprom init values at start of acb.
//   Default is enabled
#define DQ_ACB_EE_INIT_ON           (0x10000)  // enable
#define DQ_ACB_EE_INIT_OFF          (0x20000)  // disable output of eeprom init values
                                               //at start of acb
```

The user can mix direction flags with data conversion flags and data processing flags. Note for analog outputs, by default, when the ACB is enabled with `DqeEnable()`, the firmware first reads initialization values from EEPROM and outputs these before ACB data. Upon disabling the ACB, outputs will return to the initialization value. To disable this functionality OR in the `DQ_ACB_EE_INIT_OFF` to `dirflags.`

`mode`: user can select one of three available operating modes:

```
// ACB supports following modes:
#define DQ_ACBMODE_SINGLE     1  // stop after buffer is full/empty
#define DQ_ACBMODE_CYCLE      2  // wrap buffer around
#define DQ_ACBMODE_RECYCLED   4  // clear/use unprocessed frames
```

Single Mode treats an ACB buffer as linear. Acquisition (or output) stops when all data is transferred from/to the buffer.

Cycle Mode expresses a buffer as a ring. Acquisition (or output) continues indefinitely. However, if the tail of the buffer reaches its head, operation stops and a `DQ_eBufferError` flag is returned. This flag means that the buffer became full (on input) or empty (on output). Please note that the buffer system employs a ring buffer as well an ACB. In case of an input stream, the DQE will stop acquisition and set an error flag only after both (ACB and ring) buffers are completely full. In case of output stream, the DQE will stop acquisition after both buffers are empty.

Recycle Mode works the same way as Cycle Mode. In case of a full ACB buffer, however, DQE will continuously move the head and tail of the buffer together. In other words, DQE will destroy old measurements in the buffer and replace them with new measurements. This mode is useful when the user needs to see pre-triggering data.

**Note:**

All parameters supplied to the function via pointers are designed to return actual accepted values. For example, if a device cannot support a certain frequency, it will return a frequency it can support. Thus, it is a good idea to check configuration values upon a return from this function.

### *3.2.4 DqAcbGetScansCopy*

**Syntax:**

```
int DqAcbGetScansCopy(pDQBCB pBcb, char *data, uint32 size, uint32
rqsizemin, uint32 *returned, uint32 *avail)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | BCB containing the desired ACB |
| `char *data` | pointer to the data buffer (allocated by calling program) |
| `uint32 size` | size of the data buffer accessible by the pointer, in scans. |
| `uint32 rqsizemin` | minimum size of the data to copy to the user buffer, in scans. If the DQACB buffer doesn't have minimal amount of data to copy, the function doesn't copy any data |
| `uint32 *returned` | buffer for the number of scans actually copied |
| `uint32 *avail` | buffer for the number of scans of data remaining in the buffer after data was removed |

**Output:**

| | |
|---|---|
| `uint32 *returned` | the number of scans actually copied |
| `uint32 *avail` | the number of scans of data remaining in the buffer after data was removed |

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | data is `NULL`, or pBcb is `NULL` or is not an ACB |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function copies a number of scans from the ACB buffer and stores the data in the user supplied data buffer, removing the data from the ACB buffer. If the ACB buffer doesn't have `rqsizemin` scans of data to copy, the function doesn't copy any data. The function returns the number of scans copied in `returned`. `avail` returns the total number of scans worth of data remaining in the buffer. This function is intended to be called every time upon receiving a `DQ_eFrameDone` event. Alternatively, the user can call this function periodically to retrieve scans.

**Note:**

If the ACB buffer doesn't have the minimal amount of data to copy, the function doesn't copy any data.

### 3.2.5 DqAcbGetScans

**Syntax:**

```
int DqAcbGetScans(pDQBCB pBcb, char **data, uint32 size, uint32
rqsizemin, uint32 *returned, uint32 *avail)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | BCB containing the desired ACB |
| `char **data` | pointer for the data buffer |
| `uint32 size` | desired data buffer size, in scans |
| `uint32 rqsizemin` | minimum size of the data to return to the user buffer, in scans. If the ACB can't return a buffer of minimal size, the function returns 0 in `returned` and `NULL` in `data` |
| `uint32 *returned` | buffer for the number of scans actually available |
| `uint32 *avail` | buffer for the number of scans worth of data remaining in the buffer |

**Output:**

| | |
|---|---|
| `uint32 *returned` | the actual size of the buffer accessible from `data`, in scans |
| `uint32 *avail` | the number of scans worth of data remaining in the buffer (not including the returned data buffer) |

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | data is `NULL` |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function was created to avoid copying data from the ACB buffer to the user buffer. It returns a pointer to the ACB buffer where requested number of scans is available.

This function returns in `data` a pointer to a data buffer, internal to the ACB, from which the user can take acquired data. The `returned` parameter indicates how much data is in the returned buffer, in scans. `avail` indicates the amount of remaining data, not including the amount in the returned buffer.

If a data buffer large enough to hold at least `rqsizemin` scans of data cannot be returned, the function returns NULL in the `data` parameter and `0` in the `returned` parameter. In this case, it does not necessarily mean that there isn't enough data in the ACB. It could mean that the data wraps around the end of the ACB, and the amount of data before the wrap is less than `rqsizemin`. To detect this situation, check to see if the value returned in `avail` is greater than or equal to `rqsizemin`.

**Note:**

None

### 3.2.6 DqAcbPutScansCopy

**Syntax:**

```
int DqAcbPutScansCopy(pDQBCB pBcb, char *data, uint32 size, uint32
rqsizemin, uint32 *retrieved, uint32 *avail)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | BCB containing the desired ACB |
| `char *data` | pointer to the data buffer (allocated by calling program) |
| `uint32 size` | size of the data accessible by the pointer, in scans. |
| `uint32 rqsizemin` | minimum size of the data to copy from the user buffer, in scans. If the buffer doesn't have space to put the minimum amount of data, the function doesn't put any data |
| `uint32 *retrieved` | buffer for the number of scans actually copied |
| `uint32 *avail` | buffer for the number of scans can fit in the buffer after data is stored |

**Output:**

| | |
|---|---|
| `uint32 *retrieved` | the number of scans actually copied |
| `uint32 *avail` | the number of scans that can fit in the buffer after data was stored |

**Return:**

| | |
|---|---|
| DQ_BAD_PARAMETER | `data` is NULL, or `pBcb` is NULL or is not an ACB |
| DQ_SUCCESS | successful completion |

**Description:**

This function copies a number of scans from the data buffer and stores the data in the ACB. If the buffer doesn't have enough space in which to put `rqsizemin` scans of data, the function doesn't put any data. The function returns the number of scans copied in `retrieved`. `avail` returns the total number of scans available in the buffer (total free minus `retrieved`).

**Note:**

None

### 3.2.7 DqAcbPutScans

**Syntax:**

```
int DqAcbPutScans(pDQBCB pBcb, char **data, uint32 size, uint32
rqsizemin, uint32 *retrieved, uint32 *avail)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | BCB containing the desired ACB |
| `char **data` | returns a pointer to a buffer into which data can be placed |
| `uint32 size` | desired data buffer size, in scans |
| `uint32 rqsizemin` | minimum size of the data buffer to return, in scans. If the ACB can't return a buffer of minimal size, the function returns `0` in `retrieved` and NULL in `data` |

| | |
|---|---|
| `uint32 *retrieved` | buffer for the actual size of the buffer accessible from `data`, in scans |
| `uint32 *avail` | buffer for the number of scans worth of space in the ACB, not including the returned data buffer |

**Output:**

| | |
|---|---|
| `uint32 *retrieved` | the actual size of the buffer accessible from `data`, in scans |
| `uint32 *avail` | the number of scans worth of space in the ACB, not including the returned data buffer |

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | `data` is NULL, or `pBcb` is NULL or is not an ACB |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function returns in `data` a pointer to a data buffer, internal to the ACB, into which the user can put output data. The `retrieved` parameter indicates how much data the returned buffer can accept, in scans. `avail` indicates the amount of remaining space in the ACB, not including the returned buffer.

If a data buffer large enough to hold at least `rqsizemin` scans of data cannot be returned, the function returns NULL in the `data` parameter and `0` in the `retrieved` parameter. In this case, it does not necessarily mean that there isn't enough space in the ACB. It could mean that the space wraps around the end of the ACB, and the amount of space before the wrap is less than `rqsizemin`. To detect this situation, check to see if the value returned in `avail` is greater than or equal to `rqsizemin`.

**Note:**

None

### *3.2.8 DqAcbSetBurstMode*

**Syntax:**

```
int DqAcbSetBurstMode(int Iom, uint32 trigger, pDQBURST pDqBurst,
uint32 *entries)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int iom` | IOM Descriptor |
| `uint32 trigger` | TRUE to use sync line as trigger |
| `pDQBURST pDqBurst` | desired data buffer size, in scans |
| `uint32 *entries` | Number of pDQBURST entries |

**Output:**

| | |
|---|---|
| `uint32 *entries` | Number of pDQBurst entries processed |

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | data is `NULL` |
| `DQ_SUCCESS` | successful completion |

**Description:**

ACB burst mode is a special mode in which data acquisition into the IOM's internal memory continues until a predetermined number of samples is acquired. Each device can specify a pre and post trigger number to acquire. Devices must be configured with the `DQ_LN_BURST` flag in their config word before calling this function. `pDQBURST` has the following structure:

```
typedef struct {
    uint8 dev;      // device
    uint8 ss;       // sub system
    uint32 pre;     // time
    uint32 post;    // pre time
    uint32 trigger; // trigger flags
} DQBURST, *pDQBURST;
```

Upon completion of the configured acquisition interval(s), normal ACB operation mechanisms are used to transfer the data from the IOM to the host application. It is necessary for the host application to wait for both the `DQ_eBufferDone` and `DQ_eFrameDone` flags. The `DQ_eBufferDone` flag will be raised when the layer has completed transfer of the last buffer from the ACB buffer.

**Note:**

None

## 3.3 Direct Data Mapping (DMap) Functions

These are functions specific to DMap mechanisms.

### 3.3.1 DqDmapCreate

**Syntax:**

```
int DqDmapCreate(pDQE pDqe, int iom, pDQBCB *pBcb, int period,
char  **dmapin, char **dmapout)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQE pDqe` | pointer to the previously created instance of DQE |
| `int iom` | IOM Descriptor |
| `pDQBCB *pBcb` | pointer to receive allocated DMap structure |
| `int period` | base DQE periodic clock divider (defines update rate as DQEngine rate divided by <period>) |
| `char **dmapin` | buffer for memory location of allocated (plain) input part of DMap (allocated by the function) |
| `char **dmapout` | buffer for memory location of allocated (plain) output part of DMap (allocated by the function) |

**Output:**

| | |
|---|---|
| `pDQBCB *pBcb` | return pointer to allocated DMap structure |
| `char **dmapin` | memory location of allocated (plain) input part of DMap |
| `char **dmapout` | memory location of allocated (plain) output part of DMap |

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | `NULL` or `0` as a parameter |
| `DQ_DEVICE_BUSY` | DQE busy |
| `DQ_ILLEGAL_HANDLE` | invalid non-NULL value for `iom` |
| `DQ_NO_MEMORY` | memory allocation error |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function associates DQE with an IOM and creates internal structures required to handle DMap operations. The function returns a pointer to BCB to use for all other calls to this DMap and a pointer to memory allocated for input and output DMap buffers.

**Note:**

One application can have multiple DMaps created and operated at different update rates. For example, one DMap with control data can be updated every 10ms while diagnostics data can be updated every ten seconds. Use <period> parameter to control the relative update rate.

### 3.3.2 DqDmapInitOps

**Syntax:**

```
int DqDmapInitOps(pBCB pBcb)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to a previously allocated DMap structure |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal `pBcb` |
| `DQ_IOM_ERROR` | IOM reports command execution error |
| `DQ_SEND_ERROR` | cannot send packet |
| `DQ_TIMEOUT_ERROR` | IOM reply wasn't received within timeout period |
| `DQ_NO_MEMORY` | memory allocation error |
| `DQ_INIT_ERROR` | error processing PowerDNA cube parameters |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function must be called when setting of DMap entries is complete to finalize it and configure the layer involved. `DqDmapInitOps()` parses the transfer list, calculates parameters for: configuration, channel list, trigger mode and clocks. Then it sequentially calls: `DQCMD_SETCFG`, `DQCMD_SETCL`, `DQCMD_SETCLK` and finally `DQCMD_SETTRL`.

**Note:**

None

### *3.3.3 DqDmapDestroy*

**Syntax:**

`int DqDmapDestroy(pBCB pBcb)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to a previously allocated DMap structure |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid `pBcb` |
| `DQ_BAD_PARAMETER` | nothing to destroy, or `pBcb` not a DMAP |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function destroys all memory structures allocated with `DqDmapCreate()` and stops any ongoing DMap operations associated with this `pBcb.`

**Note:**

It is safe to call this function while DMap operation is running (say, in exception handler.)

### *3.3.4 DqDmapAddEntry*

**Syntax:**

```
int DqDmapAddEntry(pDQBCB pBcb, int dev, int ss, unt32 ch,
uint32 flags, int samples, char **offset)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to DMap table |
| `int dev` | layer ID |
| `int ss` | subsystem |
| `uint32 flags` | configuration flags |
| `uint32 ch` | channel (use the same flags as with channel list) |
| `int samples` | number of samples from this device/subsystem/channel |
| `char  **offset` | buffer for address of this entry in the device map |

**Output:**

| | |
|---|---|
| `char  **offset` | address of this entry in the device map |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid `pBcb` or `pBcb` is not a DMAP |
| `DQ_BAD_DEVN` | no device at given index, or device does not support DMap mode |
| `DQ_BAD_PARAMETER` | bad value for other parameter |
| `DQ_NOT_ENOUGH_ROOM` | translation list size is too big |
| `DQ_DEVICE_BUSY` | `pBcb` is busy |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

The function adds an entry transfer list entry into the transfer list.

**Note:**

1. Using this function, one can request multiple consecutive values from the same channel. While this option cannot guarantee continuity of data, it can be helpful if the user is interested in the average value of the channel.
2. This function is formerly known as DqDmapSetEntry().
3. This function works with AO layers only if the flag DQ_ACB_DATA_RAW is set. It works with AI layers with either DQ_ACB_DATA_RAW, DQ_ACB_DATA_DOUBLE or DQ_ACB_DATA_SINGLE flag set.

### *3.3.5 DqDmapAddMultipeEntries*

**Syntax:**

```
int DqDmapAddMultipeEntries(pDQBCB pBcb, pDQSETTRL pTRL, int
*entries, char **offset)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBcb pBcb` | pointer to DMap table |
| `pDQSETTRL pTRL` | array of transfer list entries |
| `int *entries` | number of entries in `pTRL` array |
| `char **offset` | array of pointers to store addresses of transfer list elements (can be NULL if not desired) |

**Output:**

| | |
|---|---|
| `int *entries` | number of entries actually processed |
| `char **offset` | array of pointer to transfer list elements |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid Descriptor |
| `DQ_SEND_ERROR` | if the unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | if nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | if error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | if the Command is processed successfully |
| Other values | low level IOM status |

**Description:**

User should fill at least one entry in the array of `DQSETTRL` type.

```
/* DQCMD_SETTRL */
typedef struct {
    uint16 dmapid;      // DMAP id (do not fill)
    uint8  dev;         // device
    uint8  ss;          // subsystem
    uint32 ch;          // channel (channel list entry)
    uint32 flags;       // control flags, including channel information
    uint16 samples;     // number of samples from this channel
} DQSETTRL, *pDQSETTRL;
```

Pass the size of array of `DQSETTRL` structures in `entries`. Also, if `offset` is not NULL, function stores addresses of points of data in the host memory.

**Note:**

1. Maximum size of the device map is limited to 512 bytes for 576 bytes packets and 1456 bytes for 1518 bytes Ethernet packets, thus `offset` couldn't be bigger then this number. Multiple packets are not supported
2. This function is formerly known as DqDmapMultipleEntries()

## 3.4 Real-time Data Mapping (Dmap) Functions

DMAP is one of the operation modes of PowerDNA. It continuously refreshes a set of channels that can span multiple layers at a specified rate paced by the cube's hardware clock.
At each clock tick, the cube's 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 that it synchronizes periodically with the PowerDNA cube's version of the DMAP.

This mode is very useful when the host computer runs a real-time operating system to ensure that the host refreshes its DMAP at deterministic intervals. It optimizes network transfer by packing all channels from multiple layers in a single UDP packet, reducing the network overhead.

The current low-level PowerDNA API (DqDmap*** functions) uses the DqEngine to refresh the DMAP at a given rate and to retry DMAP refresh request if for some reason a packet is lost.

The DqEngine is necessary on desktop oriented operating system to ensure that the DMap is refreshed periodically, but is overkill on real-time operating systems. The DqEngine needs its own thread and the user must synchronize it with its own processing loop.

The RtDmap API gives easy access to the DMAP operating mode without needing the DqEngine.

The following is a quick tutorial on using the RTDMAP API (handling of error codes is omitted):

Initialize the DMAP to refresh at 1000 Hz:

```
DqRtDmapInit(handle, &dmapid, 1000.0);
```

Add channel 0 from the first input subsystem of device 1:

```
chentry = 0;
DqRtDmapAddChannel(handle, dmapid, 1, DQ_SS0IN, &chentry, 1);
```

Add channel 1 from the first output subsystem of device 3:

```
chentry = 1;
DqRtDmapAddChannel(handle, dmapid, 3, DQ_SS0OUT, &chentry, 1);
```

Start all devices that have channels configured in the DMAP:

```
DqRtDmapStart(handle);
```

Update the value(s) to output to device 3:

```
outdata[0] = 5.0;
DqRtDmapWriteScaledData(handle, dmapid, 3, outdata, 1);
```

Synchronize the DMAP with all devices:

```
DqRtDmapRefresh(handle, dmapid);
```

Retrieve the data acquired by device 1:

```
DqRtDmapReadScaledData(handle, dmapid, 1, indata, 1);
```

Stop the devices and free all resources:

```
DqRtDmapStop(handle, dmapid);
DqRtDmapClose(handle, dmapid);
```

### *3.4.1 DqRtDmapInit*

**Syntax:**

```
int DqRtDmapInit(int handle, int* dmapid, double refreshRate);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `double refreshRate` | Rate at which the IOM will refresh its version of the DMAP |

**Output:**

| | |
|---|---|
| `int* dmapid` | Identifier of the newly created DMAP |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_NO_MEMORY` | memory allocation error or exceeded maximum table size |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

Initialize the specified IOM to operate in DMAP mode at the specified refresh rate.

**Note:**

- An IOM can have multiple DMAPs (`dmapids`) created with `DqRtDmapInit()`.
- No more than 1 VMAP or 1 DMAP can access an I/O board; however, 1 DMAP or VMAP can access multiple boards if desired.
- The maximum number of channels in 1 DMAP is 2048.
- The maximum number of VMAPs and DMAPs configured in one IOM is 256.

If your application requires a packet to transfer greater than 1472 bytes, use `DqRtDmapInitEx()` to allocate fragmented packets instead of using `DqRtDmapInit()`. See Section 3.8 below for more information.

`DqRtDmapInit` is also used to initialize an aDMap (where packet transfers from the cube/RACK to the host application are timed and initiated on the cube/RACK side). If the aDMap's conversions are synchronized to an external clock on the SYNC bus, you must call `DqRtDmapSetMode()`with the mode set to `DQ_DMAP_SYNC_DMAP_CONV` immediately following the `DqRtDmapInit()` API call. `DqRtDmapSetMode()` is described in a subsection below, and aDMap is described in Section 3.13.

### 3.4.2 DqRtDmapAddChannel

**Syntax:**

```
     int DqRtDmapAddChannel(int handle, int dmapid, int dev, int
subsystem, uint32* cl, int clSize);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP to configure |
| `int dev` | ID of the device where the channels are located |
| `int subsystem` | The subsystem to use on the device (ex: DQ_SS0IN) |
| `uint32* cl` | Array containing the channels to add to the DMAP |
| `int clSize` | Size of the channel array |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_BAD_PARAMETER` | the subsystem is invalid for this device |
| `DQ_SUCCESS` | command processed successfully |
| `Positive value` | |

**Description:**

Add one or more channels to the DMAP.

**Note:**

The maximum number of channels in 1 DMAP is 2048.

If your application requires a packet to transfer greater than 1472 bytes, use `DqRtDmapInitEx()` to allocate fragmented packets instead of using `DqRtDmapInit()`. See Section 3.8 below for more information.

### 3.4.3 DqRtDmapCheckSpace

**Syntax:**

```
int DqRtDmapCheckSpace(int handle, int dmapid, int dev, int subsystem, int*
cl, int clSize, int* in_sz_left, int* out_sz_left);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP to configure. Each `dev` must not be used in more than one MAP at a time. |
| `int dev` | ID of the device where the channels are located |
| `int subsystem` | The subsystem to use on the device (ex: DQ_SS0IN) |
| `int* cl` | Array containing the channels to add to the DMAP |
| `int clSize` | Size of the `cl` array. |

**Output:**

| | |
|---|---|
| `int* in_sz_left` | Returns the number of bytes left in the input DMAP |
| `int* out_sz_left` | Returns the number of bytes left in the output DMAP |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_BAD_PARAMETER` | the subsystem is invalid for this device |
| `DQ_PKT_TOOLONG` | too much data to fit a packet |
| `DQ_SUCCESS` | command processed successfully |
| `Zero or positive value` | Number of bytes left in the buffer |

**Description:**

Checks whether there is enough space in the DMAP to add the specified channel list. Returns a `DQ_PKT_TOOLONG` flag if the packet is too long.

**Notes**:

`in_sz_left` and `out_sz_left` provide the number of bytes that will be left for inputs and outputs after cl is added to the DMAP.

## *3.4.4 DqRtDmapGetInputMap*

**Syntax:**

```
     int DqRtDmapGetInputMap(int handle, int dmapid, int dev,
unsigned char** mappedData);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP |
| `int dev` | ID of the device where the channels are located |

**Output:**

| | |
|---|---|
| `unsigned char** mappedData` | pointer to the beginning of the device's input DMAP buffer |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

Get pointer to the beginning of the input data map allocated for the specified device.

### *3.4.5 DqRtDmapGetInputMapSize*

**Syntax:**

```
int DqRtDmapGetInputMapSize(int handle, int dmapid, int dev, int* mapSize);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP |
| `int dev` | ID of the device where the channels are located |

**Output:**

| | |
|---|---|
| `int* mappedSize` | size in bytes of the device's input data map. |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

Get the size in bytes of the input map allocated for the specified device.

### *3.4.6 DqRtDmapGetOutputMap*

**Syntax:**

```
int DqRtDmapGetOutputMap(int handle, int dmapid, int dev, unsigned char** mappedData);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP |
| `int dev` | ID of the device where the channels are located |

**Output:**

| | |
|---|---|
| `unsigned char** mappedData` | pointer to the beginning of the device's output DMAP buffer |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

Get pointer to the beginning of the output data map allocated for the specified device.

### *3.4.7 DqRtDmapGetOutputMapSize*

**Syntax:**

```
     int DqRtDmapGetOutputMapSize(int handle, int dmapid, int dev,
int* mapSize);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP |
| `int dev` | ID of the device where the channels are located |

**Output:**

| | |
|---|---|
| `int* mappedSize` | size in bytes of the device's output data map. |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

Get the size in bytes of the output map allocated for the specified device.

### *3.4.8 DqRtDmapReadScaledData*

**Syntax:**

```
     int DqRtDmapReadScaledData(int handle, int dmapid, int dev,
double* scaledBuffer, int bufferSize);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP to read from |
| `int dev` | ID of the device where the channels are located |
| `int bufferSize` | Number of elements in scaledBuffer |

**Output:**

| | |
|---|---|
| `double*scaledBuffer` | The buffer containing the scaled data. |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

Read and scale 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 and Guardian boards that readback diagnostic values.

### 3.4.9 DqRtDmapReadRawData16

**Syntax:**

```
     int DqRtDmapReadRawData16(int handle, int dmapid, int dev,
unsigned short* rawBuffer, int bufferSize);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP to read from |
| `int dev` | ID of the device where the channels are located |
| `int bufferSize` | Number of elements in rawBuffer |

**Output:**

| | |
|---|---|
| unsigned short*rawBuffer | The buffer containing the raw data. |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

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 DIO-4xx series layers.

### *3.4.10 DqRtDmapReadRawData32*

**Syntax:**

```
     int DqRtDmapReadRawData32(int handle, int dmapid, int dev,
unsigned int* rawBuffer, int bufferSize);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP to read from |
| `int dev` | ID of the device where the channels are located |
| `int bufferSize` | Number of elements in rawBuffer |

**Output:**

| | |
|---|---|
| unsigned int*rawBuffer | The buffer containing the raw data. |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

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.

### 3.4.11 DqRtDmapWriteScaledData

**Syntax:**

```
     int DqRtDmapWriteScaledData(int handle, int dmapid, int dev,
double* scaledBuffer, int bufferSize);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP to write to |
| `int dev` | ID of the device where the channels are located |
| `int bufferSize` | Number of elements in scaledBuffer |
| `double*scaledBuffer` | The buffer containing the scaled data to send to the device. |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

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 DqRtDmapRfresh().

This function should only be used with devices that generate analog data such as the AO-3xx series layers.

### 3.4.12 DqRtDmapWriteRawData16

**Syntax:**

```
     int DqRtDmapWriteRawData16(int handle, int dmapid, int dev,
unsigned short* rawBuffer, int bufferSize);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP to write to |
| `int dev` | ID of the device where the channels are located |
| `int bufferSize` | Number of elements in rawBuffer |
| `unsigned short*rawBuffer` | The buffer containing the scaled data to send to the device. |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

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 DqRtDmapRfresh().

This function should only be used with devices that generate 16-bit wide digital data such as the DIO-4xx series layers.

### *3.4.13 DqRtDmapWriteRawData32*

**Syntax:**

```
     int DqRtDmapWriteRawData32(int handle, int dmapid, int dev,
unsigned int* rawBuffer, int bufferSize);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP to write to |
| `int dev` | ID of the device where the channels are located |
| `int bufferSize` | Number of elements in rawBuffer |
| `unsigned short*rawBuffer` | The buffer containing the scaled data to send to the device. |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function writes raw data to 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 send 32-bit wide digital data such as the DIO-4xx series layers.

### *3.4.14 DqRtDmapProgram*

**Syntax:**

```
     int DqRtDmapProgram(int handle, int dmapid);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP to program |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function pre-programs the DMAP in the IOM without starting it.

When running multiple DMAPs or VMAPs in the same IOM, you must program them before starting any of them.

### *3.4.15 DqRtDmapStart*

**Syntax:**

```
int DqRtDmapStart(int handle, int dmapid);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP to start |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function starts operation, the IOM updates its internal representation of the map at the rate specified in DqRtDmapInit.

DqRtDmapStart() implicitly calls DqRtDmapProgram() if it hasn’t been called already. When running multiple DMAPs or VMAPs on the same IOM, you must explicitly call DqRtDmapProgram() on all VMAPs and DMAPs before starting any of them.

### *3.4.16 DqRtDmapStop*

**Syntax:**

```
int DqRtDmapStop(int handle, int dmapid);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP to stop |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function stops operation and the cube stops updating its internal representation of the data map.

### *3.4.17 DqRtDmapRefresh*

**Syntax:**

```
int DqRtDmapRefresh(int handle, int dmapid);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP to refresh |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

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 DqRtDMapCreate(). This function needs to be called periodically (a real-time OS is necessary) to synchronize the host and IOM data maps.

### 3.4.18 DqRtDmapRefreshOutputs, DqRtDmapRefreshOutputsExt

**Syntax:**

```
int DqRtDmapRefreshOutputs(int handle, int dmapid);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP to refresh |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function initiates a DMAP refresh by sending the output values to the IOM and returns immediately.

You need to call DqRtDmapRefreshInputs() to complete the refresh and retrieve the input values from the IOM.

**Note:**

The pair DqRtDmapRefreshOutputs()/DqRtDmapRefreshInputs() can be used instead of DqRtDmapRefresh(). This frees the task to do something useful while the DMAP refresh packets are transferred in the background.

DqRtDmapRefreshOutputsExt() is an extended version of the DqRtDmapRefreshOutputs() which additionally returns a packet counter and accepts control flags.

### 3.4.19 DqRtDmapRefreshInputs, DqRtDmapRefreshInputsExt

**Syntax:**

```
int DqRtDmapRefreshInputs(int handle, int dmapid);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP to refresh |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function completes a DMAP refresh initiated with DqRtDmapRefreshOutputs(), retrieving the input values returned by the IOM.

You need to call DqRtDmapRefreshOutputs() first to initiate the refresh and send the output values to the IOM.

**Note:**

The pair DqRtDmapRefreshOutputs()/DqRtDmapRefreshInputs() can be used instead of DqRtDmapRefresh(). This frees the task to do something useful while the DMAP refresh packets are transferred in the background.

DqRtDmapRefreshInputsExt() is an extended version of DqRtDmapRefreshInputs() which additionally returns a packet counter <`counter`> and packet timestamp <`tstamp`>.

### *3.4.20 DqRtDmapClose*

**Syntax:**

```
int DqRtDmapClose(int handle, int dmapid);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP to close |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function frees all resources allocated by the DMAP operation on the specified IOM.

### *3.4.21 DqRtDmapSetMode*

**Syntax:**

```
int DqRtDmapSetMode(int handle, int* dmapid, int mode);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the DMAP to configure |
| `int mode` | Mode flag:<br>`DQ_DMAP_SYNC_DMAP_CONV:` configures aDMap to handle conversions paced by an external clock on the sync bus<br>Other modes: <reserved> |

**Output:**

| | |
|---|---|
| `none` | |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_NO_MEMORY` | memory allocation error or exceeded maximum table size |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function changes the characteristics of the rtDMap or aDMap specified in the `dmapid,` based on the value of `mode.`

Supported `mode` flags:

- `DQ_DMAP_SYNC_DMAP_CONV` mode configures an aDMap to handle conversions paced by an external clock on the sync bus. (aDMap is a data transfer protocol where packet transfers are timed and initiated from the cube/RACK and delivered autonomously to the host application).

**Note:**

When running an aDMap and the layer's conversion is programmed to be an external clock on the sync bus, `DqRtDmapSetMode()` must be run directly after the DMap is created (after `DqRtDmapInit*()`). The `DQ_DMAP_SYNC_DMAP_CONV` configures the aDMap to handle conversions paced by an external clock on the sync bus.

For more information about aDMap, refer to Section 3.13. This API is not used in standard rtDMap.

## *3.5 Real time Variable-size Data Mapping (VMap) Functions*

VMap is a protocol developed for control applications in which the ability to get immediate realtime data is equally important as receiving a continuous flow of the data.

With messaging devices (serial, CAN, ARINC-429 interfaces), the data is a stream of bytes logically divided into frames, messages, strings, etc. There are two distinct features of messaging devices that make use of the DMap protocol inefficient, if not impossible: 1. The data is a stream and 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 and one cannot make a prediction of when and how much data will become available, and also whether or not there are some receiving/transmitting errors on the bus.

Messaging layers are supported by the Msg protocol, which shares the same buffering mechanism as the ACB protocol. Inherently, the Msg protocol buffers received 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.

VMap provides a realtime vehicle for messaging devices at the expense of restricting the ability to recover lost packets and shifting the decision whether or not to recover the lost packet to the user application. At the 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. VMap packets have additional fields. First of all, there is a flag field required to guarantee continuity of the 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. 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, 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, then set *size to send* and *size to receive* to zero. The header has a fixed width set up before starting VMap operation and the user cannot change the header size on the fly even if the channel is no longer in use.

The following is a quick tutorial on using the RTVMAP API (handling of error codes is omitted):

Initialize the VMAP to refresh at 1000 Hz:

```
   DqRtVmapInit(handle, &vmapid, 1000.0);
```

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);
```

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);
```

Add ouput port 0 to VMAP, set flag to retrieve the status of the output FIFO after each transfer:

```
chentry = 0;
flag = DQ_VMAP_FIFO_STATUS;
DqRtVmapAddChannel(handle, vmapid, DEVN, DQ_SS0OUT, &chentry, &flag, 1);
```

Enable ARINC-429 ports

```
DqAdv566Enable(handle, DEVN, TRUE);
```

Start all devices that have channels configured in the VMAP:

```
DqRtVmapStart(handle, vmapid);
```

Prepare ARINC word to send through port 0 and update VMAP:

```
uint32 arincWord = DqAdv566BuildPacket(data, label, sdi, ssm, parity);
DqRtVmapAddOutputData(handle, vmapid, 0, sizeof(uint32), &accepted,
(uint8*)&arincWord);
```

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);
```

Synchronize the VMAP with all devices:

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

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);
```

We can also check how much data was actually transmitted dutring the last refresh:

```
DqRtVmapGetOutputDataSz(handle, vmapid, 0, &tx_data_size, &tx_avl_size);
```

Stop the devices and free all resources:

```
DqRtVmapStop(handle, vmapid);
DqRtVmapClose(handle, vmapid);
```

### *3.5.1 DqRtVmapInit*

**Syntax:**

```
int DqRtVmapInit(int handle, int* vmapid, double refreshRate);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `double refreshRate` | Rate at which the IOM will refresh its version of the VMAP. |

**Output:**

| | |
|---|---|
| `int* vmapid` | Identifier of the newly created VMAP |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_NO_MEMORY` | memory allocation error or exceeded maximum table size |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function initializes the specified IOM to operate in VMAP mode.

If your application requires a packet to transfer greater than 1472 bytes, use `DqRtVmapInitEx()` to allocate fragmented packets instead of using `DqRtVmapInit()`.

**Notes**:

- An IOM can have multiple VMAPs (`vmapids`) created with `DqRtVmapInit()`.
- No more than 1 VMAP or 1 DMAP can access an I/O board; however, 1 DMAP or VMAP can access multiple boards if desired.
- The maximum number of channels in 1 VMAP is 256.
- The maximum number of VMAPs and DMAPs configured in one IOM is 256.
- The refresh rate passed to `DqRtVmapInit()` has no effect on serial boards.

### *3.5.2 DqRtVmapAddChannel*

**Syntax:**

```
int DqRtVmapAddChannel(int handle, int vmapid, int dev,
int subsystem, int* cl, int* flags, int clSize);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to configure. Each `dev` must not be used in more than one VMAP at a time. |
| `int dev` | ID of the device where the channels are located |
| `int subsystem` | The subsystem to use on the device (ex: `DQ_SS0IN`) |
| `int* cl` | Array containing the channels to add to the VMAP |
| `int* flags` | Array of flags (one per channel). Available flags are:<br>• `DQ_VMAP_FIFO_STATUS`: retrieve the state of the input/output FIFO at each refresh of the VMAP |
| `int clSize` | Size of the `cl` and `flags` arrays. See Note. |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_BAD_PARAMETER` | the subsystem is invalid for this device |
| `DQ_SUCCESS` | command processed successfully |
| `Zero or positive value` | Number of bytes left in the buffer |

**Description:**

Add one or more channel(s) to the VMAP.

**Notes**:

- The maximum number of channels in 1 VMAP is 256.
- For AI, AO, and DIO devices, the `clSize` parameter for this function should be set to “1” because these boards use a single FIFO to transfer data. The number of physical channels is set in a separate function (`DqRtVmapSetChannelList`). The VMAP channel for these boards will contain values that were read from (AI) or will be written to (AO) the physical channels, with physical channel data interleaved in the single FIFO. I/O boards that do not use a single FIFO for transfers (e.g., CAN boards, ARINC boards, etc.) will set `clSize` according to however many channels are configured on the device, and each channel counted in the `clSize` will map to a seperate VMAP channel.
- For AO, AI or DIO devices, after calling the `DqRtVmapAddChannel()` function but before `DqRtVmapStart()`, users must call `DqRtVmapSetChannelList()` to set the physical channel list by entering the number of `channels` as the `clSize` parameter in `DqRtVmapSetChannelList()`.
- No more than 1 VMAP or 1 DMAP can access an I/O board; however, 1 DMAP or VMAP can access multiple boards if desired.

If your application requires a packet to transfer greater than 1472 bytes, use `DqRtVmapInitEx()` to allocate fragmented packets instead of using `DqRtVmapInit()`. See Section 3.8 below for more information.

### 3.5.3 DqRtVmapSetChannelList

**Syntax:**

```
int DqRtVmapSetChannelList(int handle, int vmapid, int dev,
int subsystem, int* cl, int clSize);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to configure. Each `dev` must not be used in more than one VMAP at a time. |
| `int dev` | ID of the device where the channels are located |
| `int subsystem` | The subsystem to use on the device (ex: `DQ_SS0IN`) |
| `int* cl` | Array containing the channels to add to the VMAP |
| `int clSize` | Size of the `cl` array |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_BAD_PARAMETER` | the subsystem is invalid for this device |
| `DQ_SUCCESS` | command processed successfully |
| `Zero or positive value` | Number of bytes left in the buffer |

**Description:**

Set the channel list for analog input (AI), analog output (AO) or digital I/O (DIO) devices to be included into the VMAP.

**Note**:

This function is only used for VMAPs for AI, AO and DIO devices (These boards consume/produce data at the same rate for each of the board’s configured channels and use a single FIFO for transfer data). `DqRtVmapSetChannelList()` is called after the `DqRtVmapAddChannel()` function but before `DqRtVmapStart()` to identify the number of physical channels for the device that will be used in the VMAP.

## *3.5.4 DqRtVmapAddChannelMaxSize*

**Syntax:**

```
int DqRtVmapAddChannelMaxSize(int handle, int vmapid, int dev, int
subsystem, int cl, int flags, int max_size);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to configure. Each `dev` must not be used in more than one VMAP at a time. |
| `int dev` | ID of the device where the channels are located |
| `int subsystem` | The subsystem to use on the device (ex: `DQ_SS0IN`) |
| `int cl` | Channel to add to the VMAP |
| `int flags` | Flag for channel. Available flags are:<br>• `DQ_VMAP_FIFO_STATUS`: retrieve the state of the input/output FIFO at each refresh of the VMAP |
| `int max_size` | Maximum number of bytes to receive (for inputs) or write (for outputs) for this channel with each refresh. |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_BAD_PARAMETER` | the subsystem is invalid for this device |
| `DQ_PKT_TOOLONG` | too much data to fit a packet |
| `DQ_SUCCESS` | command processed successfully |
| `Zero or positive value` | Number of bytes left in the buffer |

**Description:**

Add a channel to the VMAP and specify the maximum number of bytes to receive or send per packet transfer for that channel. Returns `DQ_PKT_TOOLONG` if `max_size` will result in a payload too big for the specified VMAP and packet size defined for the system.

This API allows users to calculate the remaining size of the VMAP buffer at the time of adding channels (not in runtime).

**Notes**:

- The maximum number of channels in 1 VMAP is 256.
- No more than 1 VMAP or 1 DMAP can access an I/O board; however, 1 DMAP or VMAP can access multiple boards if desired.

If your application requires a packet size for transfers that is greater than 1472 bytes, you can use `DqRtVmapInitEx()` to allocate fragmented packets instead of using `DqRtVmapInit()`.

## 3.5.5 DqRtVmapSetScanRate

**Syntax:**

```
int DqRtVmapSetScanRate(int handle, int vmapid, int dev,
int subsystem, float scan_rate);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to configure. Each `dev` must not be used in more than one VMAP at a time. |
| `int dev` | ID of the device where the channels are located |
| `int subsystem` | The subsystem to use on the device (ex: `DQ_SS0IN`) |
| `float scan_rate` | Update rate for the specified device (`dev`) |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_BAD_PARAMETER` | the subsystem is invalid for this device |
| `DQ_SUCCESS` | command processed successfully |
| `Zero or positive value` | Number of bytes left in the buffer |

**Description:**

Sets the scan rate for the specified device. `DqRtVmapSetScanRate`() only applies to AI, AO, and DIO devices.

**Note**:

This function is called before `DqRtVmapStart()`.

## 3.5.6 DqRtVmapSetConfig

**Syntax:**

```
int DqRtVmapSetConfig(int handle, int vmapid, int dev, int subsystem,
uint32 config);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to configure. Each `dev` must not be used in more than one VMAP at a time. |
| `int dev` | ID of the device where the channels are located |
| `int subsystem` | The subsystem to use on the device (ex: `DQ_SS0IN`) |
| `uint32 config` | Configuration settings for specified device (`dev`) |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |

| | |
|---|---|
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_BAD_PARAMETER` | the subsystem is invalid for this device |
| `DQ_SUCCESS` | command processed successfully |
| `Zero or positive value` | Number of bytes left in the buffer |

**Description:**

Sets configuration bits for a specified device. This function is used to configure the clock source and trigger source for a device and is only needed if you want to use an external clock or trigger. The `config` parameter is defined by logically ORing #defined configuration parameters. Refer to `DqCmdSetCfg()` for a list of `config` clock/trigger parameters.

**Note**:

This function is called before `DqRtVmapStart()`.

### *3.5.7 DqRtVmapGetInputMap*

**Syntax:**

```
int DqRtVmapGetInputMap(int handle, int vmapid, int trl_list,
unsigned char** mappedData);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the VMAP |
| `int trl_list` | Index of the VMAP entry |

**Output:**

| | |
|---|---|
| `unsigned char** mappedData` | pointer to the beginning of the specified entry input VMAP buffer |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

Get pointer to the beginning of the input data map allocated for the specified device.

**Note:**

The `trl_index` parameter is the data index for a given VMAP channel as set up in the VMAP data buffer that is transferred during the refresh call. When setting up the VMAP, the first channel added with `DqRtVmapAddChannel()` is TRL0, the next one is TRL1, and so on. Note that all boards use the same TRL list. If you add 3 analog input boards to the VMAP and then add 3 analog output boards, the first three AI boards map to TRL0 thru TRL2 and the three AO boards will be TRL3 thru TRL5.

To use TRL, users must keep track of the order in which channels are added, which may be difficult to use; alternatively, APIs that use the channel number instead of TRL are described in the *Simplified VMap and VMap+ Functions* section of this manual.

### *3.5.8 DqRtVmapGetOutputMap*

**Syntax:**

```
int DqRtVmapGetOutputMap(int handle, int dmapid, int trl_list,
unsigned char** mappedData);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int dmapid` | Identifier of the VMAP |
| `int trl_list` | Index of the VMAP entry |

**Output:**

| | |
|---|---|
| `unsigned char** mappedData` | pointer to the beginning of the specified entry output VMAP buffer |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

Get pointer to the beginning of the output data map allocated for the specified device.

**Note:**

The `trl_index` parameter is the data index for a given VMAP channel as set up in the VMAP data buffer that is transferred during the refresh call. This parameter is described in detail in `DqRtVmapGetInputMap()`. Note that functions found in the *Simplified VMap and VMap+ Functions* section of this manual perform the VMAP functions based on a channel list instead of a TRL and are easier to use.

### *3.5.9 DqRtVmapAddOutputData*

**Syntax:**

```
int DqRtVmapAddOutputData(int handle, int vmapid, int trl_index,
uint32 data_size, int* act_size, uint8* data);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to write to |
| `int trl_index` | Index of the VMAP entry to update |
| `int data_Size` | Size of the data buffer in bytes |
| `uint8* data` | Data to send to the VMAP |

**Output:**

| | |
|---|---|
| `int* act_size` | Number of bytes actually sent to the VMAP |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

Copies data into the output packet and returns number of bytes left in the packet.

**Note:**

The `trl_index` parameter is the data index for a given VMAP channel as set up in the VMAP data buffer that is transferred during the refresh call. This parameter is described in detail in `DqRtVmapGetInputMap()`. Note that functions found in the *Simplified VMap and VMap+ Functions* section of this manual perform the VMAP functions based on a channel list instead of a TRL and are easier to use.

### *3.5.10 DqRtVmapGetOutputDataSz*

**Syntax:**

```
int DqRtVmapGetOutputDataSz(int handle, int vmapid, int trl_index,
int* data_size, int* avl_size);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to write to |
| `int trl_index` | Index of the VMAP entry |

**Output:**

| | |
|---|---|
| `int* data_size` | Number of bytes sent to the device |
| `Int* avl_size` | Number of bytes that the output FIFO can accept. |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

Returns how many bytes were copied from the output packet and how much more room is available in the output FIFO.

**Note:**

The `trl_index` parameter is the data index for a given VMAP channel as set up in the VMAP data buffer that is transferred during the refresh call. This parameter is described in detail in `DqRtVmapGetInputMap()`. Note that functions found in the *Simplified VMap and VMap+ Functions* section of this manual perform the VMAP functions based on a channel list instead of a TRL and are easier to use.

### 3.5.11 DqRtVmapRqInputDataSz

**Syntax:**

```
int DqRtVmapRqInputDataSz(int handle, int vmapid, int trl_index,
uint32 rq_size, int* act_size, uint8 **indataptr);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to read from |
| `int trl_index` | Index of the VMAP entry |
| `int rq_size` | Number of bytes to read from the device |

**Output:**

| | |
|---|---|
| `int* act_size` | Maximum number of bytes that can possibly be transferred (might be smaller than the requested number) |
| `uint8** indataptr` | Pointer to the buffer where the received data will be accessible once the refresh is complete. |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

Requests the maximum number of bytes to read from the device.

**Note:**

The `trl_index` parameter is the data index for a given VMAP channel as set up in the VMAP data buffer that is transferred during the refresh call. This parameter is described in detail in `DqRtVmapGetInputMap()`. Note that functions found in the *Simplified VMap and VMap+ Functions* section of this manual perform the VMAP functions based on a channel list instead of a TRL and are easier to use.

### 3.5.12 DqRtVmapGetInputData

**Syntax:**

```
int DqRtVmapGetInputData(int handle, int vmapid, int trl_index,
uint32 max_size, int* data_size, int* avl_size, uint8* data);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to read from |
| `int trl_index` | Index of the VMAP entry |
| `uint32 max_size` | Size of the “data” buffer |

**Output:**

| | |
|---|---|
| `int* data_size` | Number of bytes actually read. |
| `Int* avl_size` | Number of un-read bytes still sitting in the device’s input FIFO |
| `uint8* data` | Buffer containing the received bytes |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

Copies data from the input packet and returns number of bytes copied and available size in the input FIFO.

**Note:**

The `trl_index` parameter is the data index for a given VMAP channel as set up in the VMAP data buffer that is transferred during the refresh call. This parameter is described in detail in `DqRtVmapGetInputMap()`.

The `data_size` parameter returns the number of bytes returned in the `data` buffer. Verify `data_size` is greater than zero before using `data`.

Note that functions found in the *Simplified VMap and VMap+ Functions* section of this manual perform the VMAP functions based on a channel list instead of a TRL and are easier to use.

### *3.5.13 DqRtVmapProgram*

**Syntax:**

```
int DqRtVmapProgram(int handle, int vmapid);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to program |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function pre-programs the VMAP in the IOM without starting it.

When running multiple DMAPs or VMAPs in the same IOM, you must program them before starting any of them.

### 3.5.14 DqRtVmapStart

**Syntax:**

`int DqRtVmapStart(int handle, int vmapid);`

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to start |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function starts operation.

`DqRtVmapStart()` implicitly calls `DqRtVmapProgram()` if it hasn’t been called already. When running multiple DMAPs or VMAPs on the same IOM, you must explicitly call `DqRtVmapProgram()` on all VMAPs and DMAPs before starting any of them.

### 3.5.15 DqRtVmapStartTr

**Syntax:**

`int DqRtVmapStartTr(int handle, int vmapid, int trig);`

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to start |
| `int trig` | Disable for triggering layers in software<br>0 = no software trigger, >0 = software trigger |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function starts operation. When `(trig > 0),` this function operates exactly the same as `DqRtVmapStart()`. When `(trig == 0),` this function forces devices to wait for a hardware or broadcast trigger.

`DqRtVmapStart*()` functions implicitly call `DqRtVmapProgram()` if it hasn’t been called already. When running multiple DMAPs or VMAPs on the same IOM, you must explicitly call `DqRtVmapProgram()` on all VMAPs and DMAPs before starting any of them.

### *3.5.16 DqRtVmapStop*

**Syntax:**

```
int DqRtVmapStop(int handle, int vmapid);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to stop |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function stops operation.

### *3.5.17 DqRtVmapRefresh, DqRtVmapRefreshExt*

**Syntax:**

```
int DqRtVmapRefresh(int handle, int vmapid , int flags);

int DqRtVmapRefreshExt(int handle, int vmapid, int flags,
uint16* counter, uint16* tstamp)
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to refresh |
| `int flags` | `DQ_VMAP_REREQUEST` to request last VMap if the previous call to this function has timed out |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_SUCCESS` | command processed successfully |
| `DQ_FIFO_OVERFLOW` | FIFO on device overflowed |

**Description:**

This function refreshes the host's version of the map by downloading the IOM's map.

`DqRtVmapRefreshExt()` is an extended version of `DqRtVmapRefresh()` which additionally returns packet counter in <`counter`> and timestamp in <`timestamp`>.

It can be also used to re-request unreceived VMap packet by setting `DQ_VMAP_REREQUEST` flag and either passing zero as a <`counter`> or an actual packet counter of timed-out request. Firmware stores last two transmitted VMap packets by default.

The error `DQ_FIFO_OVERFLOW` occurs when data is refreshed too slow and FIFO on device has overflow. The acquisition loop needs to run fast enough to empty the FIFO before it overflows. Once the error occurs you must stop the RTVMAP and start it again.

### 3.5.18 DqRtVmapRefreshOutputs, DqRtVmapRefreshOutputsExt

**Syntax:**

```
int DqRtVmapRefreshOutputs(int handle, int vmapid, int flags);

int DqRtVmapRefreshOutputsExt(int handle, int vmapid, int flags,
uint16* counter)
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to refresh |
| `int flags` | For future extension of VMAP mode |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function initiates a VMAP refresh by sending the output values to the IOM and returns immediately.

You need to call `DqRtVmapRefreshInputs()` to complete the refresh and retrieve the input values from the IOM.

`DqRtVmapRefreshOutputsExt()` is an extended version of `DqRtVmapRefreshOutputs().` Extended function accepts control flags and packet counter. One of the important flags is `DQ_VMAP_REREQUEST` which allows to re-request missing VMap reply packet without the risk of output the same message twice. The packet number should be specified in <`counter`> parameter. If zero is passed function re-request the previous packet (packet counter is maintained in the VMap control structure inside the library).

**Note:**

The pair `DqRtVmapRefreshOutputs()/DqRtVmapRefreshInputs()` can be used instead of `DqRtVmapRefresh().` This frees the task to do something useful while the VMAP refresh packets are transferred in the background.

Extended version of this function can be used to re-request missed packet. If `DqRtVmapRefreshInputsExt()` is used the missed counter can be calculated from the last successfully received packet for the VMap as

```
if (dqCounter < 0xffff) dqCounter++; else dqCounter = 1;
```

### *3.5.19 DqRtVmapRefreshInputs, DqRtVmapRefreshInputsExt*

**Syntax:**

```
int DqRtVmapRefreshInputs(int handle, int vmapid)

int DqRtVmapRefreshInputsExt(int handle, int vmapid, uint16* counter,
uint16* tstamp)
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to refresh |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function completes a VMAP refresh initiated with `DqRtVmapRefreshOutputs()`, retrieving the input values returned by the IOM.

You need to call `DqRtVmapRefreshOutputs()` first to initiate the refresh and send the output values to the IOM.

`DqRtVmapRefreshInputsExt()` is an extended version of the `DqRtVmapRefreshInputs()`which additionally returns packet number in <`counter`> and packet timestamp in <`tstamp`>

**Note:**

The pair `DqRtVmapRefreshOutputs()/DqRtVmapRefreshInputs()` can be used instead of `DqRtVmapRefresh()`. This frees the task to do something useful while the VMAP refresh packets are transferred in the background.

### *3.5.20 DqRtVmapClose*

**Syntax:**

```
int DqRtVmapClose(int handle, int vmapid);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to close |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function frees all resources allocated by the VMAP operation on the specified IOM.

## 3.6 Real time Variable-size Data Variable-channels Mapping (VMap+) Functions

Variable data/variable channel mode of mapping is required for layers with large number of possible addresses. For example, DNx-1553-553 layer implementing 1553B military avionics protocol is capable of supporting two independent buses with 32 remote terminals each containing 32 subadresses and multiple status words.

VMap+ operations require adding channels with `DQ_VMAP_SPEC_CHANNEL` flag. When the flag is specified the channel specified in the DqRtVmapAddChannel() is only used to determine data size; the actual channel to perform the operation with is specified in the runtime using DqRtVmapAddOutputChannelData() and DqRtVmapRqInputChannelData().

VMap+ packet contains information for both amount of data and channel to retrieve. Request packet always contains two 16-bit entries per each input or output channel. Reply packet can contain either one or two 16-bit entries per channel (the same way as with VMap). One of the entries contains the number of actually read or written data points and the second one is optional information about the size of data (or empty space for write) remains in the FIFO.

| Host to Cube (request) | | Cube to Host (reply) |
|---|---|---|
| Flags | | Flags |
| Size Write 0 | | Size Written 0 |
| Channel Write 0 | → to IOM | [Remains 0] |
| … | | … |
| Size Write N | | Size Written N |
| Channel Write N | | [Remains N] |
| Size Request 0 | | Size Read 0 |
| Channel Request 0 | | [Remains 0] |
| … | | … |
| Size Request N | | Size Read N |
| Channel Request N | ← from IOM | [Remains N] |
| Data Write 0 | | Data Read 0 |
| … | | … |
| | | |

### 3.6.1 DqRtVmapAddOutputChannelData

**Syntax:**

```
     int DqRtVmapAddOutputChannelData(int handle, int vmapid, int
trl_index, uint32 channel, uint32 data_size, int* act_size, uint8*
data);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to write to |
| `int trl_index` | Index of the VMAP entry to update |
| `uint32 channel` | Channel to output data to |
| `int data_Size` | Size of the data buffer in bytes |
| `uint8* data` | Data to send to the VMAP |

**Output:**

| | |
|---|---|
| `int* act_size` | Number of bytes actually sent to the VMAP |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

Copies channel information and data into the output packet and returns number of bytes left in the packet.

## *3.6.2 DqRtVmapRqInputChannelDataSz*

**Syntax:**

```
     int DqRtVmapRqInputChannelDataSz(int handle, int vmapid, int
trl_index, uint32 channel, uint32 rq_size, int* act_size, uint8
**indataptr);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to read from |
| `int trl_index` | Index of the VMAP entry |
| `uint32 channel` | Channel to request data from |
| `int rq_size` | Number of bytes to read from the device |

**Output:**

| | |
|---|---|
| `int* act_size` | Maximum number of bytes that can possibly be transferred (might be smaller than the requested number) |
| `uint8** indataptr` | Pointer to the buffer where the received data will be accessible once the refresh is complete. |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_BAD_DEVN` | there is no device with the specified number |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

Function requests the maximum number of bytes to read from the specified channel of the device.

## 3.7 Simplified VMap and VMap+ Functions

A set of simplified functions was developed to make writing of VMap-based real-time application simpler. After creating VMap or VMap+ the following functions may be used instead of `DqRtVmapAddOutputChannelData(), DqRtVmapAddOutputData(), DqRtVmapRqInputChannelDataSz(), DqRtVmapRqInputDataSz().` The main idea of the following API is to avoid using index in the transfer list and require user to update all entries in the outgoing VMap packet. Instead, following functions are recommended:

```
DqRtVmapInitOutputPacket()
DqRtVmapWriteOutput()
DqRtVmapPlusWriteOutput()
DqRtVmapRequestInput()
DqRtVmapPlusRequestInput()
DqRtVmapReadInput()
DqRtVmapInputFifoAvailable()
DqRtVmapOutputFifoAvailable()
```

These functions are intended to be called in the main cycle of the application:

1. Allocate a fixed or a circular buffer for each channel of each device. We suggest to use a circular queue as a data structure to store device data
2. Initialize VMap, Add channels to VMap/VMap+ and start VMap
3. Enable operations, then

```
while (!stop) {

DqRtVmapInitOutputPacket()  // clear outgoing packet
```

4. Store output data only for channels you would like to output data
   `DqRtVmapWriteOutput(, *data)`
   or
   `DqRtVmapPlusWriteOutput(, channel,…, *data)`
   Last parameter is a pointer to the user buffer where data should be copied from.
   For VMap/VMap+ <channel> is the same channel which was added using `DqRtVmapAddChannel().` For VMap+ <rq_channel> is the channel user actually requests data from.

5. Store requests for input data. Input data is what is going to be returned from the layer.

   ```
   DqRtVmapRequestInput()
   DqRtVmapPlusRequestInput()
   ```
   The same difference between <channel> and <rq_channel> applies when user requests input

6. Exchange data with the cube. There are two ways to refresh data

   a. using `DqRtVmapRefresh()` to send packet with the output data and input data requests, wait for and the receive a packet with the requested input data

   b. Call `DqRtVmapRefreshOutputs()` to send output packet at the end of the cycle and then at the beginning of the cycle call `DqRtVmapRefreshInputs()` to receive recently received data.
      As an option `DqRtXmapRefreshInputs()` can be called. This function receives VMap/VMap+ packet regardless of <vmap_id> of the packet and returns <vmap_id> which can be used to select appropriate processing for the data. An additional IOM handle (see `DqRtAddIomPort()`) is recommended for this way of refreshing data.
7. Call `DqRtVmapReadInput()` to copy received data from the VMap into user buffer
8. Optionally, if the flag `DQ_VMAP_FIFO_STATUS` is set user can use `DqRtVmapInputFifoAvailable()` and `DqRtVmapOutputFifoAvailable()` functions to retrieve the amount of data either available in the input FIFO or room available in the output FIFO.
9. End while loop (step 3).

```
}
```

## *3.7.1 DqRtVmapInitOutputPacket*

**Syntax:**

```
int DAQLIB DqRtVmapInitOutputPacket(int handle, int vmapid)
```

**Command:**

RT

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM returned by `DqOpenIOM()` |
| `int vmapid` | VMap handle |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_ENTRY` | vmapid is illegal |
| `DQ_ILLEGAL_HANDLE` | invalid non-NULL value for `iom` |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function clears all structures of the outgoing packet associated with <vmapid>

**Note:**

None

## *3.7.2 DqRtVmapWriteOutput*

**Syntax:**

```
int DAQLIB DqRtVmapWriteOutput(int handle, int vmapid, int device,
uint32 channel, int rq_size, uint8* data)
```

**Command:**

RT

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM returned by `DqOpenIOM()` |

| | |
|---|---|
| `int vmapid` | VMap handle |
| `int device` | Device number |
| `uint32 channel` | Channel number |
| `int rq_size` | Requested size |
| `uint8* data` | Pointer the data to copy to the output packet |

**Output:**

| | |
|---|---|
| Returns positive value | Number of bytes actually written to the output buffer |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_ENTRY` | vmapid is illegal |
| `DQ_ILLEGAL_HANDLE` | invalid non-NULL value for `iom` |
| `DQ_ILLEGAL_INDEX` | Couldn't find index in the transfer table |
| `Zero or positive` | successful completion |

**Description:**

This function copies data from the user buffer to the output VMap packet

**Note:**

As explained in the previous two sections, the VMap packet does not contain information on channels included into channel list created with VmapAddChannel and therefore must be filled in the order of the transfer list. Thus, first entry corresponds to the 1st channel, second to 2nd... DMap can be filled out of order since each channel takes a fixed number of bytes in the packet

### *3.7.3 DqRtVmapPlusWriteOutput*

**Syntax:**

```
int DAQLIB DqRtVmapPlusWriteOutput(int handle, int vmapid, int
device, uint32 channel, uint32 rq_channel, int rq_size, uint8* data)
```

**Command:**

RT

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM returned by `DqOpenIOM()` |
| `int vmapid` | VMap handle |
| `int device` | Device number |
| `uint32 channel` | Channel number as specified in DqRtVmapAddChannel() |
| `uint32 rq_channel` | Requested channel |
| `int rq_size` | Requested size |
| `uint8* data` | Pointer the data to copy to the output packet |

**Output:**

| | |
|---|---|
| Returns positive value | Number of bytes actually written to the output buffer |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_ENTRY` | vmapid is illegal |
| `DQ_ILLEGAL_HANDLE` | invalid non-NULL value for `iom` |
| `DQ_ILLEGAL_INDEX` | Couldn't find index in the transfer table |
| `Zero or positive` | successful completion |

**Description:**

This function copies data for the specified <rq_channel> from the user buffer to the output VMap+ packet

**Note:**

`<channel>` channel number as specified in `DqRtVmapAddChannel()` when this entry into the VMap is created. VMap+ allows you to specify *actual* channel (to substitute the channel number assigned when the channel was added to VMap+). This actual channel has to be passed as `<rq_channel>`

### 3.7.4 DqRtVmapRequestInput

**Syntax:**

```
int DAQLIB DqRtVmapRequestInput(int handle, int vmapid, int device,
uint32 channel, int rq_size)
```

**Command:**

RT

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM returned by `DqOpenIOM()` |
| `int vmapid` | VMap handle |
| `int device` | Device number |
| `uint32 channel` | Channel number |
| `int rq_size` | Requested size |

**Output:**

| | |
|---|---|
| Returns positive value | Number of bytes actually requested |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_ENTRY` | vmapid is illegal |
| `DQ_ILLEGAL_HANDLE` | invalid non-NULL value for `iom` |
| `DQ_ILLEGAL_INDEX` | Couldn't find index in the transfer table |
| `Zero or positive` | successful completion |

**Description:**

This function requests input data and store request in the output VMap packet

**Note:**

None

### 3.7.5 DqRtVmapPlusRequestInput

**Syntax:**

```
int DAQLIB DqRtVmapPlusRequestInput(int handle, int vmapid, int
device, uint32 channel, uint32 rq_channel, int rq_size)
```

**Command:**

RT

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM returned by `DqOpenIOM()` |
| `int vmapid` | VMap handle |
| `int device` | Device number |

| | |
|---|---|
| `uint32 channel` | Channel number as specified in DqRtVmapAddChannel() |
| `uint32 rq_channel` | Requested channel |
| `int rq_size` | Requested size |

**Output:**

| | |
|---|---|
| Returns positive value | Number of bytes actually written to the output buffer |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_ENTRY` | vmapid is illegal |
| `DQ_ILLEGAL_HANDLE` | invalid non-NULL value for `iom` |
| `DQ_ILLEGAL_INDEX` | Couldn’t find index in the transfer table |
| `Zero or positive` | successful completion |

**Description:**

This function requests input data and store request in the output VMap packet

**Note:**

`<channel>` channel number as specified in `DqRtVmapAddChannel()` when this entry into the VMap is created. VMap+ allows you to specify *actual* channel (to substitute the channel number assigned when the channel was added to VMap+). This actual channel has to be passed as `<rq_channel>`

### *3.7.6 DqRtVmapReadInput*

**Syntax:**

```
int DAQLIB DqRtVmapReadInput(int handle, int vmapid, int device,
uint32 channel, int rq_size, int* ret_size, uint8* data)
```

**Command:**

RT

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM returned by `DqOpenIOM()` |
| `int vmapid` | VMap handle |
| `int device` | Device number |
| `uint32 channel` | Channel number |
| `int rq_size` | Requested size |

**Output:**

| | |
|---|---|
| `int* ret_size` | Actually received |
| `uint8* data` | Pointer to the user-allocated to copy data |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_ENTRY` | vmapid is illegal |
| `DQ_ILLEGAL_HANDLE` | invalid non-NULL value for `iom` |
| `DQ_ILLEGAL_INDEX` | Couldn’t find index in the transfer table |
| `Zero or positive` | successful completion |

**Description:**

This function copies received data into the user-allocated buffer

**Note:**

The allocated buffer should not be less than `<rq_size>` bytes

### *3.7.7 DqRtVmapInputFifoAvailable*

**Syntax:**

```
int DAQLIB DqRtVmapInputFifoAvailable(int handle, int vmapid, int
device, uint32 channel, int* available)
```

**Command:**

RT

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM returned by `DqOpenIOM()` |
| `int vmapid` | VMap handle |
| `int device` | Device number |
| `uint32 channel` | Channel number |

**Output:**

| | |
|---|---|
| `int* available` | Amount of data available in the FIFO after VMap operation |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_ENTRY` | vmapid is illegal |
| `DQ_ILLEGAL_HANDLE` | invalid non-NULL value for `iom` |
| `DQ_ILLEGAL_INDEX` | Couldn't find index in the transfer table |
| `Zero or positive` | successful completion |

**Description:**

This function returns amount of data additionally available in the FIFO after VMap operation is completed (in bytes).

**Note:**

If the request data size is zero but user still wants to see the amount of data it must use DQ_VMAP_FIFO_RQSIZE flag when channel is added. Otherwise, if the requested size is zero then the request to calculate amount of data available in the FIFO is ignored by the firmware.

### *3.7.8 DqRtVmapOutputFifoAvailable*

**Syntax:**

```
int DAQLIB DqRtVmapOutputFifoAvailable(int handle, int vmapid, int
device, uint32 channel, int* available)
```

**Command:**

RT

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM returned by `DqOpenIOM()` |
| `int vmapid` | VMap handle |
| `int device` | Device number |
| `uint32 channel` | Channel number |

**Output:**

| | |
|---|---|
| `int* available` | Amount of data available in the FIFO after VMap operation |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_ENTRY` | vmapid is illegal |
| `DQ_ILLEGAL_HANDLE` | invalid non-NULL value for `iom` |
| `DQ_ILLEGAL_INDEX` | Couldn't find index in the transfer table |
| `Zero or positive` | successful completion |

**Description:**

This function returns amount of room additionally available in the output FIFO after VMap operation is completed (in bytes).

**Note:**

If the request data size is zero but user still wants to see the amount of data it must use DQ_VMAP_FIFO_RQSIZE flag when channel is added. Otherwise, if the requested size is zero then the request to calculate amount of data available in the FIFO is ignored by the firmware.

## *3.7.9 DqAdvRawToScaleValue*

**Syntax:**

```
int DAQLIB DqAdvRawToScaleValue(int handle, int device, uint32
channel, uint32 rawVal, double* pScaledVal)
```

**Command:**

RT

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM returned by `DqOpenIOM()` |
| `int device` | Device number |
| `uint32 channel` | Channel number |
| `uint32 rawVal` | Raw ADC value |

**Output:**

| | |
|---|---|
| `double* pScaledVal` | Returned scaled ADC value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Invalid non-NULL value for `iom` |
| `DQ_BAD_DEVN` | Incorrect device number |
| `DQ_BAD_PARAMETER_2` | Unsupported gain value detected as channel parameter |
| `Zero or positive` | Successful completion |

**Description:**

This function converts raw ADC values and returns reading as scaled voltage or current.

**Note:**

### 3.7.10 *DqAdvScaleToRawValue*

**Syntax:**

```
int DAQLIB DqAdvScaleToRawValue(int handle, int device,
uint32 channel, double scaledVal, uint32* pRawVal)
```

**Command:**

RT

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM returned by `DqOpenIOM()` |
| `int device` | Device number |
| `uint32 channel` | Channel number |
| `double scaledVal` | Scaled DAC value |

**Output:**

| | |
|---|---|
| `uint32* pRawVal` | Returned raw DAC value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Invalid non-NULL value for `iom` |
| `DQ_BAD_DEVN` | Incorrect device number |
| `DQ_BAD_PARAMETER_2` | Incorrect channel parameter |
| `DQ_NOT_IMPLEMENTED` | Parameter not implemented |
| `Zero or positive` | Successful completion |

**Description:**

This function converts a scaled DAC voltage or current reading and returns a raw DAC value.

**Note:**

## 3.8 DMap / VMap Packet Fragmentation Functions

For applications that require a single, larger packet, packet fragmentation can be enabled with the `DqRtVmapInitEx()` / `DqRtDmapInitEx()` API described below. The API allows an increase in the MAP buffer beyond one packet size.

An advantage of using fragmented packets is that data can be sent in conveniently sized transmissions. Larger transmissions of data can simplify managing the data (all data is transferred in one packet). Generally, one big fragmented packet is better performance-wise than smaller individual packets sent per device. Transmissions arrive to the IOM as multiple buffers and are serviced by one interrupt and context switch. Servicing a VMAP/DMAP takes approximately 40 us, so processing all devices together instead of individually could be more time efficient.

Disadvantages of using fragmentation is that it introduces additional delays upon loss of a packet: loss of a single fragment requires retransmission of all fragments; reassembly of IP packets takes time and delays data to the app until all fragments are successfully received.

**Calculating packet size:**
Both API accept a `DQ_RTMAP_PARAM` structure that includes elements for setting the maximum payload size and the maximum transfer unit (`mtu`).

Packets will be fragmented if the number of bytes to be transmitted is greater than the programmed mtu value, which represents the greatest data size that can be transmitted in a single network transfer.

For the Ethernet v2 protocol, the mtu is 1518 bytes (this is slightly reduced by header requirements):
1518 (mtu) - 18 (Ethernet header) - 20 (IP header) - 8 (UDP header)
= a maximum of 1472 bytes for UDP packet size (1st fragment)
(and subsequent fragments would be 1480 because the UDP header doesn’t retransmit)

**Maximum packet size:**
The RFC limits the size of a fragmented packet to 64k (65535 bytes). However, UEI IOMs' maximum packet size is limited to 12000 bytes, a UEI real-time network stack limitation. This yields a maximum of 8 fragments:

- The DQ_RTMAP_PARAM element, `max_payload_sz`, represents the max payload size, and cannot exceed 12000.
- Packets will be fragmented if `DqRtVmapInitEx()` or `DqRtDmapInitEx()` API is used, and if the packet to transmit is greater than the `mtu` size and less than 12000.
- This results in an 8 fragment maximum.

In VMAP mode, the size of the outgoing packet can be checked by using the `DqRtVmapRqInputDataSz()` function, which requests space for the data and returns the amount to of data still available in the packet.

**Note:** The ***InitEx API only use as many packets as needed to fit data. However, these functions do use additional memory to allocate required buffer structures than the `***Init()` API use.

### 3.8.1 DqRtDmapInitEx

**Syntax:**

```
int DqRtDmapInitEx(int handle, int* dmapid, DQ_RTMAP_PARAM* vmapparam);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `DQ_RTMAP_PARAM* dmapparam` | Structure identifying maximum size of payload, maximum transfer unit, and desired refresh rate. See Note below for structure information and example |

**Output:**

| | |
|---|---|
| `int* dmapid` | Identifier of the newly created DMAP |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_NO_MEMORY` | memory allocation error or exceeded maximum table size |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

Extended version of `DqRtDmapInit(). DqRtDmapInitEx`() initializes the specified IOM to operate in DMAP mode and allows an increase in the DqRtDmap buffer beyond one packet size. This allows the use of fragmented packets when the DMAP size is ≤ 11.5 kilobytes.

**Note**:

- The `DQ_RTMAP_PARAM dmapparm` structure must be declared and initialized before calling this function. The structure format is as follows:

```
typedef struct {
         int max_payload_sz; // maximum size of payload, packet
                             //   will be fragmented
                             //   if > GetIOM(handle)->MaxDqPayload
         int mtu;            // maximum transfer unit, packets
                             //   will be fragmented if exceeded
         double refreshRate; // desired refresh rate,
                             //   used as a guideline to program
                             //   layer parameters (not the actual
                             //    A/D rate)
} DQ_RTMAP_PARAM, *pDQ_RTMAP_PARAM;
```

For example:

```
DQ_RTMAP_PARAM dmapparam = {12000, 1518, 0.1};
```

- An IOM can have multiple DMAPs (`dmapids`) created with `DqRtDmapInitEx()`.
- No more than 1 VMAP or 1 DMAP can access an I/O board; however, 1 DMAP or VMAP can access multiple boards if desired.
- The maximum number of channels in 1 DMAP is 2048.
- `DqRtDmapInitEx` can be used to initialize an aDMap (where packet transfers from the cube/RACK to the host application are timed and initiated on the cube/RACK side). If the aDMap's conversions are synchronized to an external clock on the SYNC bus, you must call `DqRtDmapSetMode()`with the mode set to `DQ_DMAP_SYNC_DMAP_CONV` immediately following

the `DqRtDmapInitEx()` API call. `DqRtDmapSetMode()` is described in a Section 3.4.21, and aDMap is described in Section 3.13.

- The maximum number of VMAPs and DMAPs configured in one IOM is 256.

### *3.8.2 DqRtVmapInitEx*

**Syntax:**

```
     int DqRtVmapInitEx(int handle, int* vmapid, DQ_RTMAP_PARAM*
vmapparam);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `DQ_RTMAP_PARAM* vmapparam` | Structure identifying maximum size of payload, maximum transfer unit, and desired refresh rate. See Note below for structure information and example. |

**Output:**

| | |
|---|---|
| `int* vmapid` | Identifier of the newly created VMAP |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_NO_MEMORY` | memory allocation error or exceeded maximum table size |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

Extended version of `DqRtVmapInit(). DqRtVmapInitEx`() initializes the specified IOM to operate in VMAP mode and allows an increase in the DqRtVmap buffer beyond one packet size. This allows the use of fragmented packets when the VMAP size is ≤ 11.5 kilobytes.

**Note**:

The `DQ_RTMAP_PARAM vmapparm` structure must be declared and initialized before calling this function. The structure format is as follows:

```
     typedef struct {
              int max_payload_sz; // maximum size of payload, packet
                                  //   will be fragmented
                                  //   if > GetIOM(handle)->MaxDqPayload
              int mtu;            // maximum transfer unit, packets
                                  //   will be fragmented if exceeded
              double refreshRate; // desired refresh rate,
                                  //   used as a guideline to program
                                  //   layer parameters (not the actual
                                  //    A/D rate)
     } DQ_RTMAP_PARAM, *pDQ_RTMAP_PARAM;
```

For example:

```
     DQ_RTMAP_PARAM vmapparam = {12000, 1518, 0.1};
```

- An IOM can have multiple VMAPs (`vmapids`) created with `DqRtVmapInitEx()`.

- No more than 1 VMAP or 1 DMAP can access an I/O board; however, 1 DMAP or VMAP can access multiple boards if desired.
- The maximum number of channels in 1 VMAP is 256.
- The maximum number of VMAPs and DMAPs configured in one IOM is 256.

## 3.9 Messaging (Msg) Functions

These are functions specific to Msg mechanisms.

### 3.9.1 DqMsgCreate

**Syntax:**

```
int DqMsgCreate(pDQE pDqe, int iom, uint32 devn, uint32 ss,
uint32 dir, uint32 config, uint32 queueSize, pDQBCB *pBcb)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQE pDqe` | pointer to the previously created instance of DQE |
| `int iom` | Handle to the IOM returned by `DqOpenIOM()` |
| `uint32 devn` | layer number inside the IOM |
| `uint32 ss` | subsystem of layer |
| `uint32 dir` | direction for message queue; either `DQ_MSG_DIRECTION_RECEIVING` or `DQ_MSG_DIRECTION_SENDING` |
| `uint32 config` | configuration (layer specific) |
| `uint32 queueSize` | maximum number of entries in message queue |
| `pDQBCB *pBcb` | pointer to receive a newly allocated BCB structure |

**Output:**

| | |
|---|---|
| `pDQBCB *pBcb` | receives a newly allocated BCB structure |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | memory allocation error |
| `DQ_BAD_PARAMETER` | NULL or 0 as a parameter, `queueSize` is less than `DQ_MSG_MIN_QUEUE_SIZE`, or `dir` is neither `DQ_MSG_DIRECTION_SENDING` nor `DQ_MSG_DIRECTION_RECEIVING` |
| `DQ_ILLEGAL_HANDLE` | invalid non-NULL value for `iom` |
| `DQ_DEVICE_BUSY` | layer already taken |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function allocates a new Msg queue-type BCB.

**Note:**

None.

### 3.9.2 DqMsgInitOps

**Syntax:**

```
int DqMsgInitOps(pDQBCB pBcb)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to Msg queue |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | `pBcb` is NULL |
| `DQ_ILLEGAL_HANDLE` | `pBcb` is not a Msg queue |
| `DQ_NO_MEMORY` | unable to allocate necessary structure |
| `DQ_SUCCESS` | successful completion |

**Description:**

The function sets up configuration for one device in the IOM.

**Note:**

None.

### *3.9.3 DqMsgDestroy*

**Syntax:**

```
int DqMsgDestroy(pDQBCB pBcb)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to Msg queue |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | `pBcb` is NULL |
| `DQ_ILLEGAL_HANDLE` | `pBcb` is already deallocated or is not a Msg queue |
| `DQ_SUCCESS` | successful completion |

**Description:**

The function destroys a Msg queue created by `DqMsgCreate()`.

**Note:**

None.

### *3.9.4 DqMsgRecvMessage*

**Syntax:**

```
int DqMsgRecvMessage(pDQBCB pBcb, pDqMessage message, int
*gotMsg, uint32 *avail)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to Msg queue |
| `pDqMessage message` | `DqMessage` structure in which to store the received message. The `dataSize` field should indicate the amount of allocated space in the `data` field. |
| `int *gotMsg` | pointer to receive value indicating whether a message was received from the queue |
| `uint32 *avail` | pointer to receive the number of messages still in the receiving message queue |

**Output:**

| | |
|---|---|
| `pDqMessage message` | The `data` field is filled with the received message. The `dataSize` field's value is set to the data's actual size. |
| `int *gotMsg` | returns `TRUE` if a message was retrieved and stored in `message` parameter, `FALSE` if there was no message in the queue |
| `uint32 *avail` | returns the number of messages still in the receiving message queue |

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | if any parameter is NULL |
| `DQ_ILLEGAL_HANDLE` | if `pBcb` does not reference a receiving Msg queue |
| `DQ_NOT_ENOUGH_ROOM` | if the data field of `message`, indicated by the `dataSize` field, is not large enough to store the message |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function gets a message received by one device in the IOM.

**Note:**

Check the `gotMsg` parameter after calling to see if a message was actually retrieved.

### *3.9.5 DqMsgSendMessage*

**Syntax:**

```
int DqMsgSendMessage(pDQBCB pBcb, pDqMessage message, uint32
*avail)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to Msg queue |
| `pDqMessage message` | message to send |
| `uint32 *avail` | pointer to receive the number of messages that there is still room for in the sending message queue |

**Output:**

| | |
|---|---|
| `uint32 *avail` | returns the number of messages that there is still room for in the sending message queue |

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | if any parameter is NULL |
| `DQ_ILLEGAL_HANDLE` | if `pBcb` does not reference a sending Msg queue |
| `DQ_NOT_ENOUGH_ROOM` | the message queue is full |
| `DQ_SUCCESS` | successful completion |

**Description:**

The function sends a message from one device in the IOM.

**Note:**

This function stores a copy of the message in the message queue; thus, the caller is responsible for freeing the memory of the passed in message.

### *3.9.6 DqMsgCount*

**Syntax:**

```
int DqMsgCount(pDQBCB pBcb, uint32 *msg_avail, uint32
*space_avail)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to Msg queue |
| `uint32 *msg_avail` | pointer to receive the number of messages in the message queue |
| `uint32 *space_avail` | pointer to receive the number of messages that there is available space for in the message queue |

**Output:**

| | |
|---|---|
| `uint32 *msg_avail` | returns the number of messages in the message queue |
| `uint32 *space_avail` | returns the number of messages that there is available space for in the message queue |

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | if any parameter is NULL |
| `DQ_ILLEGAL_HANDLE` | if `pBcb` does not reference a Msg queue |
| `DQ_SUCCESS` | successful completion |

**Description:**

The function tells how many messages are in a queue, and how many messages can be added to the queue before it is full.

**Note:**

None.

## *3.10 Mapped Messaging Mode (M3) Functions (No longer supported in 3.8.0+ releases)*

These are functions specific to mapped messaging mode mechanisms.

### *3.10.1 DqMmCreate*

**Syntax:**

```
int DqMmCreate(pDQE pDqe, int iom, uint32 dir, uint32 config,
uint32 queueSize, pDQBCB *pBcb)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQE pDqe` | pointer to the previously created instance of DQE |
| `int iom` | IOM Descriptor |
| `uint32 dir` | Direction for message queue; either `DQ_MSG_DIRECTION_RECEIVING` or `DQ_MSG_DIRECTION_SENDING` |
| `uint32 config` | configuration (layer specific) |
| `uint32 queueSize` | maximum number of entries in message queue |
| `pDQBCB *pBcb` | receives a newly allocated bcb structure |

**Output:**

| | |
|---|---|
| `pDQBCB *pBcb` | return pointer to allocated bcb structure |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | memory allocation error |
| `DQ_BAD_PARAMETER` | NULL or 0 as a parameter, `queueSize` is less than `DQ_MSG_MIN_QUEUE_SIZE`, or `dir` is neither `DQ_MSG_DIRECTION_SENDING` nor `DQ_MSG_DIRECTION_RECEIVING` |
| `DQ_ILLEGAL_HANDLE` | invalid non-NULL value for `iom` |
| `DQ_DEVICE_BUSY` | layer already taken |
| `DQ_SUCCESS` | successful completion |

**Description:**

Allocates a new M3 queue-type BCB.

**Note:**

None.

### 3.10.2 DqMmInitOps

**Syntax:**

```
int  DqMmInitOps(pDQBCB pBcb, int scan_rate_hz, int
scans_per_packet)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to a previously allocated M3 structure |
| `int scan_rate_hz` | rate at which to obtain scans |
| `int scans_per_packet` | number of scans to collect in one packet before sending |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal `pBcb` |
| `DQ_IOM_ERROR` | IOM reports command execution error |
| `DQ_SEND_ERROR` | cannot send packet |
| `DQ_TIMEOUT_ERROR` | IOM reply wasn't received within timeout period |
| `DQ_NO_MEMORY` | memory allocation error |
| `DQ_INIT_ERROR` | error processing PowerDNA cube parameters |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function must be called when setting of M3 entries is complete to finalize it and configure the layer involved. `DqMmInitOps()` parses a transfer list, calculates parameters for: configuration, channel list, trigger mode, and clocks.

**Note:**

None

### 3.10.3 DqMmDestroy

**Syntax:**

`int DqMmDestroy(pBCB pBcb)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to a previously allocated M3 structure |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid `pBcb` |
| `DQ_BAD_PARAMETER` | nothing to destroy, or `pBcb` not a M3 |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function destroys all memory structures allocated with `DqMmCreate()` and stops any ongoing M3 operations associated with this `pBcb.`

**Note:**

It is safe to call this function while M3 operation is running (say, in exception handler.)

### *3.10.4 DqMmSetEntry*

**Syntax:**

```
int DqMmSetEntry(pDQBCB pBcb, int dev, int ss, uint32 ch, uint32
flags, int samples, int *offset)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to M3 structure |
| `int dev` | layer ID |
| `int ss` | subsystem |
| `uint32 flags` | configuration flags |
| `uint32 ch` | channel (use the same flags as with channel list) |
| `int samples` | number of samples from this device/subsystem/channel |
| `char  **offset` | buffer for address of this entry in the device map |

**Output:**

| | |
|---|---|
| `char  **offset` | address of this entry in the device map |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid `pBcb` or `pBcb` is not a M3 |
| `DQ_BAD_DEVN` | no device at given index, or device does not support M3 mode |
| `DQ_BAD_PARAMETER` | bad value for other parameter |
| `DQ_NOT_ENOUGH_ROOM` | translation list size is too big |
| `DQ_DEVICE_BUSY` | `pBcb` is busy |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

The function adds an entry transfer list entry to a transfer list.

**Note:**

Using this function, one can request multiple consecutive values from the same channel. While this option cannot guarantee continuity of data, it can be helpful if the user is interested in the average value of the channel.

### 3.10.5 DqMmSetLayerConfig

**Syntax:**

```
int DqMmSetLayerConfig(pDQBCB pBcb, int dev, int ss, uint32
flags, float rate, float *actual)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to M3 structure |
| `int devn` | layer ID |
| `int ss` | subsystem |
| `uint32 flags` | configuration flags |
| `float rate` | requested clock rate for the device |
| `float *actual` | returns the actual clock rate for the device |

**Output:**

| | |
|---|---|
| `float *actual` | address of this entry in the device map |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid `pBcb` or `pBcb` is not a M3 |
| `DQ_BAD_DEVN` | no device at given index, or device does not support M3 mode |
| `DQ_BAD_PARAMETER` | bad value for other parameter |
| `DQ_NOT_ENOUGH_ROOM` | translation list size is too big |
| `DQ_DEVICE_BUSY` | `pBcb` is busy |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets the configuration flags and clock rate for a device in mapped messaging mode

**Note:**

None

### 3.10.6 DqMmRecvMessage

**Syntax:**

```
int DqMmRecvMessage(pDQBCB pBcb, pDqMessage message, int
*gotMsg, uint32 *avail)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to Msg queue |
| `pDqMessage message` | `DqMessage` structure in which to store the received message. The `dataSize` field should indicate the amount of allocated space in the `data` field. |

| | |
|---|---|
| `int *gotMsg` | pointer to receive value indicating whether a message was received from the queue |
| `uint32 *avail` | pointer to receive the number of messages still in the receiving message queue |

**Output:**

| | |
|---|---|
| `pDqMessage message` | The `data` field is filled with the received message. The `dataSize` field's value is set to the data's actual size. |
| `int *gotMsg` | returns `TRUE` if a message was retrieved and stored in `message` parameter, `FALSE` if there was no message in the queue |
| `uint32 *avail` | returns the number of messages still in the receiving message queue |

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | if any parameter is NULL |
| `DQ_ILLEGAL_HANDLE` | if `pBcb` does not reference a receiving M3 queue |
| `DQ_NOT_ENOUGH_ROOM` | if the data field of `message`, indicated by the `dataSize` field, is not large enough to store the message |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function gets a message received by one device in the IOM.

**Note:**

Check the `gotMsg` parameter after calling to see if a message was actually retrieved.

### *3.10.7 DqMmSendMessage*

**Syntax:**

```
int DqMmSendMessage(pDQBCB pBcb, pDqMessage message, uint32
*avail)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to Msg queue |
| `pDqMessage message` | message to send |
| `uint32 *avail` | pointer to receive the number of messages that there is still room for in the sending message queue |

**Output:**

| | |
|---|---|
| `uint32 *avail` | returns the number of messages that there is still room for in the sending message queue |

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | if any parameter is NULL |
| `DQ_ILLEGAL_HANDLE` | if `pBcb` does not reference a sending M3 queue |
| `DQ_NOT_ENOUGH_ROOM` | the message queue is full |
| `DQ_SUCCESS` | successful completion |

**Description:**

The function sends a message from one device in the IOM.

**Note:**

This function stores a copy of the message in the message queue; thus, the caller is responsible for freeing the memory of the passed in message.

### *3.10.8 DqMmCount*

**Syntax:**

`int DqMmCount(pDQBCB pBcb, uint32 *msg_avail, uint32 *space_avail)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to M3 queue |
| `uint32 *msg_avail` | pointer to receive the number of messages in the message queue |
| `uint32 *space_avail` | pointer to receive the number of messages that there is available space for in the message queue |

**Output:**

| | |
|---|---|
| `uint32 *msg_avail` | returns the number of messages in the message queue |
| `uint32 *space_avail` | returns the number of messages that there is available space for in the message queue |

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | if any parameter is NULL |
| `DQ_ILLEGAL_HANDLE` | if `pBcb` does not reference a Msg queue |
| `DQ_SUCCESS` | successful completion |

**Description:**

The function tells how many messages are in a queue, and how many messages can be added to the queue before it is full.

**Note:**

None.

## *3.11 ACB and DMap Control and Event Functions*

These are control and event functions common to both ACB and DMap mechanisms.

### *3.11.1 DqeEnable*

**Syntax:**

`int DqeEnable(uint32 enable, pDQBCB *ppBcb, int BcbNum, int broadcast)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `uint32 enable` | `TRUE` to enable/`FALSE` to disable |
| `pDQBCB *ppBcb` | pointer to [an array of] BCB (points to ACBE or DMap) |
| `int BcbNum` | array size |
| `int broadcast` | if `TRUE` start simultaneously using software trigger |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_IOM_ERROR` | IOM reports command execution error |
| `DQ_SEND_ERROR` | cannot send packet |
| `DQ_TIMEOUT_ERROR` | IOM reply wasn't received within timeout period |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function enables and disables acquisition/output. Set `enable` TRUE to start operation or FALSE to stop it.

If `DqeEnable()` starts operation, it issues a `DQCMD_SETMD` command to switch listed devices into operating mode. If `DqeEnable()` is about to stop operation, it switches devices into configuration mode.

A device can be switched into operation mode and back to configuration mode independently. Switching one device to a different mode doesn't affect other devices in the same IOM. Thus, one device can perform ACB streaming operation while another can be configured in DMap mode.

After a device is switched into operation mode, it waits for any trigger to start operation. If the configuration calls for a hardware trigger, the layer waits for an edge on the trigger line. If a software trigger is selected, this trigger pulse can be issued by using the `DQCMD_START` command. Upon this command, the firmware clock triggers the line itself.

If device is configured to use software start and/or stop triggers, `DqeEnable()`also sends the `DQCMD_START` command.

`ppBcb` is an array of `BcbNum` size that contains a list of all operations you would like to start or stop.
If `broadcast` parameter is set to TRUE, `DqeEnable()` uses a broadcast type of `DQCMD_START` to start all devices nearly simultaneously. Otherwise, it sends `DQCMD_START` commands according to the device order in the list supplied.

**Note:**

None

### *3.11.2 DqeSetEvent*

**Syntax:**

```
int DqeSetEvent(pDQBCB pBcb, uint32 evtFlags)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to a previously allocated BCB |
| `uint32 evtFlags` | event flags |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_INIT_ERROR` | failed to create an event |

| | |
|---|---|
| `DQ_SUCCESS` | successful completion |

**Description:**

This function sets up event flags about which the user wants to be informed. Normal execution flow assumes that user application waits for events by calling `DqWaitForEvent()` and relinquishes control to the OS. The PowerDNA header file defines the following event flags:

```
// BCB events
#define DQ_eDataAvailable    (1L<<0)  // Data is available, DMap completed data exchange
#define DQ_eFrameDone        (1L<<1)  // One or more frames are filled with data
#define DQ_eBufferDone       (1L<<2)  // Buffer is completed (straight buffer only)
#define DQ_ePacketDone       (1L<<3)  // One or more packets with data has arrived
#define DQ_eTransmitError    (1L<<4)  // Error encountered during transmission
#define DQ_eReceiveError     (1L<<5)  // Error encountered during receiving
#define DQ_eStarted          (1L<<6)  // Acquisition/output started
#define DQ_eStopped          (1L<<7)  // Acquisition/output stopped
#define DQ_eStartTrig        (1L<<8)  // Start trigger received
#define DQ_eStopTrig         (1L<<9)  // Stop trigger received
#define DQ_eBufferError      (1L<<10) // Overrun/under-run event (DQACB buffer only)
#define DQ_ePacketLost       (1L<<11) // Error is unrecoverable, one or more packets are lost
#define DQ_eTimeOut          (1L<<12) // Timeout (no events received)
#define DQ_ePacketOOB        (1L<<13) // Packet is out of bounds
#define DQ_eStatusError      (1L<<14) // Error detected by DQCMD_RDSTS sent by heartbeat
#define DQ_eNoHeartReply     (1L<<15) // IOM did not respond to heartbeat

#define DQ_eAllEvents        (0x3FFF) // all events mask
```

- `DQ_eDataAvailable` is generated when the writer thread transfers any data from the ring buffer to the ACB. This event can also be set when DMap operation completes a data exchange between host and IOM, or when a Msg queue has received a message.
- `DQ_eFrameDone` is set when incoming data crosses a frame boundary. In reality, it is the time when the writer thread has contiguous data in the ring buffer and transfers it into ACB. In case of an output operation, a reader thread takes data from ACB, converts it and writes it into the output ring buffer. Thus, at the beginning of output operation, a `DQ_eFrameDone` event is set quite often when data is transferred from an ACB to the empty ring buffer.
- `DQ_eBufferDone` is set in a Single-mode ACB when the buffer becomes full on input (or empty on output). Normally `DQ_eBufferDone` is accomplished with the `DQ_eStopped` flag
- `DQ_ePacketDone` – (not implemented)
- `DQ_eTransmitError` is a NIC or TCP/IP transmit error
- `DQ_eReceiveError` is a NIC or TCP/IP receive error
- `DQ_eStarted` is set when IOM starts operation (not implemented in revision 2)
- `DQ_eStopped` is set when IOM stops operation (not implemented in revision 2)
- `DQ_eStartTrig` is set when start trigger event occurred (either in hardware or in software – reaching triggering conditions) (not implemented in revision 2)
- `DQ_eStopTrig` is set when start trigger event occurred (either in hardware or in software – reaching triggering conditions) (not implemented in revision 2)
- `DQ_eBufferError` is set upon buffer overrun (input, Cycle mode) or buffer underrun (output, Cycle mode) condition
- `DQ_ePacketLost` is set when one or more packets are unrecoverably lost

- `DQ_ePacketOOB` is set when packet received by host is so far off that host cannot insert this packet into the ring buffer
- `DQ_eStatusError` is set when the Heartbeat mechanism sends a `DQCMD_RDSTS` command to the IOM, and the IOM responds with error flags set. You can then call `DqGetLastStatus()` to get the received status flags.
- `DQ_eNoHeartReply` is set when the Heartbeat mechanism sends a `DQCMD_RDSTS` command to the IOM, and the IOM fails to respond within the timeout period

**Note:**

None

### *3.11.3 DqeGetEvent*

**Syntax:**

```
int DqeGetEvent(pDQBCB pBcb, uint32 *evtFlags)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB pBcb` | pointer to a previously allocated BCB |
| `uint32 *evtFlags` | buffer for event flags associated with `pBcb` |

**Output:**

| | |
|---|---|
| `uint32 *evtFlags` | event flags associated with `pBcb` |

**Return:**

| | |
|---|---|
| `DQ_SUCCESS` | successful completion |

**Description:**

This function gets event notification flags indicating events that have occurred associated with BCB.

**Note:**

None

### *3.11.4 DqeWaitForEvent*

**Syntax:**

```
int DqeWaitForEvent(pDQBCB *ppBcb, int num, int WaitAll, int
timeout, uint32 *pEvents)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `pDQBCB *ppBcb` | pointer to an array of BCB |
| `int num` | size of `ppBcb` array |
| `int WaitAll` | wait mode |
| `int timeout` | wait timeout |
| `uint32 *pEvents` | buffer for event flags associated with `ppBcb` |

**Output:**

| | |
|---|---|
| `uint32 *pEvents` | event flags associated with `ppBcb` |

**Return:**

| | |
|---|---|
| `DQ_TIMEOUT_ERROR` | timeout passed before the desired event(s) triggered |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function waits for either an event to happen or for a timeout to occur.

Set `WaitAll` to TRUE to wait for all events included in `ppBcb[]` (AND-mode) or set it to FALSE to return if any of events included in `ppBcb[]` occur (OR-mode).

Set `timeout` to time in ms or to `DQ_WAIT_INDEFINITELY` to never timeout.

`pEvents` is a pointer to a bit field in which to store event flags that have been triggered for the first BCB only. If `NULL`, event flags are not returned, and `DqeGetEvent()` must be called to retrieve the event flags. In either case, `DqeGetEvent()` must be called to retrieve the flags for BCBs other than the first one.

**Note:**

This function encapsulates an OS-specific function to wait for an event to make the user application portable.

In Windows, this function calls `WaitForMultipleEvents().`

In Linux, this function calls `pthread_cond_timewait()` or `pthread_cond_wait()` which restricts us to waiting for one event.

DMap produces only two types of events: `DQ_eDataAvailable` (when transfer is completed) and `DQ_ePacketLost` (when transfer has failed.)

## *3.12 Asynchronous Event Functions*

For realtime operations with multiple cubes, it is critical that cubes send packets to the host only upon host request. This requirement comes from the need to avoid network collisions in case two or more cubes try to send packets at the same time.

In most cases, the host request requirement is not a major limitation. However, it might present a problem in situations when there is no reason to query the cube frequently because of infrequent changes in the data, yet the host needs to react quickly if some special event – for example an error – has happened on the cube. In this case, the host would not know about the error until after a request for status information is sent. Asynchronous mode was developed to resolve this issue.

Asynchronous mode uses a separate communication channel to receive packets originated upon certain events on the cube. The function set for these operations starts with the DqRtAsync () prefix. These functions use the same IP address as all other functions to communicate with the cube but a different UDP port. The default UDP port for these operations is 6344 and can be changed using cube configuration commands.

There are two major applications for asynchronous or event-driven operations:

1. Event notification (receiving a certain message type, for example)
2. aDMap/aVMap application when data packets are originated from the cube side upon the timer or a trigger event

To use asynchronous mode, create a new handle to the IOM using `DqAddIOMPort()`. This function adds a secondary communication socket to the main one. Next, enable layer-specific events, for example using `DqAdv553ConfigEvents()` for the DNx-1553-553 layer.
To process events, create a thread which will wait on the `DqRtAsyncReceive()` function call. Packets will be processed as they arrive, one by one, unless timeout occurs.

```
while(!stop) {
      // wait for the packet to arrive
      DqRtAsyncReceive()
      if (ret == DQ_TIMEOUT_ERROR) continue;


      // process packet
      …
}
```

**Note:** If the call flag `DQ_RTRE_WAIT_PACKET` is not set, `DqRtAsyncReceive()` will not wait for an event packet to arrive. Instead, the function quits if there are no new event packets in the TCP/IP stack queue. Without this flag, the timeout is set to 1 microsecond.

### *3.12.1 DqRtAsyncOpenIOM*

**Syntax:**

```
int DAQLIB DqRtAsyncOpenIOM(int handle, int* new_handle, uint16
UDP_Port, uint32 mTimeOut, int depth, pDQRDCFG *pDqCfg)
```

**Command:**

Open an asynchronous communications channel

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `uint16 UDP_Port` | UDP port to use – default is `DQ_UDP_DAQ_PORT_ASYNC` (6344) |
| `uint32 mTimeOut` | Timeout associated with the new handle, ms |
| `int depth` | reserved |
| `pDQRDCFG *pDqCfg` | Pointer to the cube configuration structure |

**Output:**

| | |
|---|---|
| `int* new_handle` | Created channel handle to use with DqRtAsync…() functions |
| `pDQRDCFG *pDqCfg` | reserved, pass NULL |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SOCK_LIB_ERROR` | Socket library error |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |

| | |
|---|---|
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**
This function is now obsolete. Please use `DqAddIOMPort()`to obtain a handle for asynchronous events.

### 3.12.2 DqRtAsyncEnableEvents

**Syntax:**

```
int DAQLIB DqRtAsyncEnableEvents(int handle, uint32 flags, uint32
enable_mask)
```

**Command:**

Enable or disable asynchronous events

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `uint32 flags` | Flags define function behavior |
| `uint32 enable_mask` | Which layers are enabled (1 in that bit positions) and which are disabled (0) |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**
This function enables or disables handling of events on per-layer basis. It must be called before receiving any events.

`<flags>`

- Reserved for future use in PowerDNA mode
- In UEIPAC mode, set to device number (DEVN) of the IO board/layer where the events are being enabled.

`<enable_mask>`

- In PowerDNA mode, this is a bit mask of the IO boards/layers where the events are being enabled.
- In UEIPAC mode, set to 1 to enable events.

### 3.12.3 DqRtAsyncReceive

**Syntax:**

```
int DAQLIB DqRtAsyncReceive(int handle, uint32 flags, pDQPKT* buffer,
int* size)
```

**Command:**

Receive event packet

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `int flags` | Define behavior of the function |
| `pDQPKT* buffer` | Pointer to the received packet |
| `int* size` | Size of the received packet |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function receives a packet from the UDP port queue associated with the handle, marks this entry as processed, then discards the packet.

`<flags>`

`DQ_RTRE_WAIT_PACKET` – wait for incoming packet. If this flag is not set function waits only 1us (i.e. returns a packet if the packet is already in the queue).

**Notes:**

`DqRtAsyncReceive` does not work on the **UEIPAC**. Instead, please use `DqCmdReceiveEvent` on **UEIPAC** systems.

### 3.12.4 DqRtAsyncSend

**Syntax:**

```
int DAQLIB DqRtAsyncSend(int handle, uint32 flags, int pkt_size,
pDQPKT buffer)
```

**Command:**

Send event packet

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |

| | |
|---|---|
| `int flags` | Define behavior of the function |
| `int pkt_size` | Packet size |
| `pDQPKT buffer` | Pointer to the packet |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sends a packet as a reply to the remote events

`<flags>` - reserved

**Notes:**

The function sends a standard DQPKT packet using an asynchronous socket. It is your responsibility to fill the packet with proper data including DQ command because this function fills only two fields of the packet:

```
    if (0 == buffer->dqCommand) = htonl(DQCMD_EVENT);
    buffer->dqProlog = htonl(DQ_PROLOG);
```

### 3.12.5 DqRtAsyncSendResponse

**Syntax:**

```
int DAQLIB DqRtAsyncSendResponse(int hd, uint32 flags, int pkt_size,
pDQEVENT pEvent)
```

**Command:**

Send event packet as a response to the event

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `int flags` | Define behavior of the function |
| `int pkt_size` | Packet size |
| `pDQEVENT pEvent` | Event packet |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |

| | |
|---|---|
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**
This function sends a packet as a reply to the received event.

`<flags>`
`DQ_RTRE_REPLY_REQD` - tells the cube to send a confirmation packet back

## *3.13 aDMap and aVMap Timer/Clocked/Watermark Event Functions*

In aDMap or aVMap mode, packets originate on the cube/RACK side, triggered by a clock on the sync bus, a timer, or (in the case of aVMap mode) a FIFO watermark. An aDMap or aVMap is initialized similar to a standard rtDMap or rtVMap and requires calls to many of the functions described in the sections above.

In addition the aD/VMap must be initialized with `DqRtAXMapStart`. The user then must refresh the aD/VMap normally once before calling `DqRtAXMapEnable`.

At that point the aD/Vmap will be running autonomously where data transfers from the cube/RACK to the host application occur as programmed.

**Note:** When using aDMap **and** the layer's conversion is programmed to be an external clock on the sync bus, the following aXMap initialization sequence must be run:

1. `DqRtDmapInit()` must be called to initialize the IOM to run in DMAP mode
2. `DqRtDmapSetMode()` must be called directly after `DqRtDmapInit()`, and the `DQ_DMAP_SYNC_DMAP_CONV` flag must be passed to `DqRtDmapSetMode()` to configure aDMap to correctly handle externally clocked data
3. `DqRtAXMapStart()` is called after any other channel, etc initialization is complete
4. The user must then refresh the aDMap normally once before calling `DqRtAXMapEnable()`

### *3.13.1 DqRtAXMapStart*

**Syntax:**

```
int DqRtAXMapStart(int hd, int xmapid, uint32 amaptype, double rate,
uint32 emitCfg, uint32 flags);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int xmapid` | Identifier of the aVMAP or aDMAP to configure |
| `uint32 amaptype` | Set aVMAP/aDMAP to run from clock on sync bus, timer or watermark mode |
| `double rate` | In timer mode, the rate at which a packet will be emitted (Hz) |
| `uint32 emitCfg` | The sync line to specify for clock (in clock on sync bus mode) or the FIFO watermark in samples (in aVMap |

| | |
|---|---|
| | watermark mode) at which a packet is emitted |
| `uint32 flags` | <Reserved> |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Handle is illegal |
| `DQ_ILLEGAL_ENTRY` | xmapid is illegal |
| `DQ_BAD_PARAMETER` | `amaptype, emitCfg, or flags` value is invalid |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function sets up an already created VMap or DMap for aVMap/aDMap operation. (The VMap or DMap should already be created with the DqRtVmapInit* or DqRtDmapInit* functions). The VMap or DMap can be configured to emit packets at the clock rate of a clock on the sync bus (using external clock mode), a given frequency (using timer mode) or a VMap can emit packets on a FIFO watermark.

`<amaptype>`:

`amaptype` sets the aXMap mode:

```
#define DQ_AVMAP_TYP_WMRK (0)       // send data samples on watermark
#define DQ_AVMAP_TYP_TIME (1)       // send data samples on timing
#define DQ_AVMAP_TYP_CLCK (2)       // send data samples on external clock
```

Note: `DQ_AVMAP_TYP_WMRK` is only valid for aVMAP mode

`<rate>`:

`rate` sets the rate that the IOM will send packets when in timed mode (`DQ_AVMAP_TYP_TIME`). If the `rate` set via this API is too high, the function call to `DqRtAXMapEnable` will return with a bad parameter. This parameter is only used if `amaptype` is set to timed mode; otherwise it is ignored.

`<emitCfg>:`

`emitCfg` configures when packets are emitted in external clock and watermark modes. In watermark mode this value represents the number of samples in a layer's FIFO to be filled before a packet is emitted. If it is larger than the FIFO on one of the layers, the function will return an error. In external clock mode, this value will contain the sync line to take the clocks from (sync line 0,1,2 or 3 mapped on bits 0 through 1) and the divider for that on bits 2-31. This parameter is ignored in timed mode.

Note: When programming `emitCfg` with the sync lines in external clock mode, the `DQ_AVMAP_CLK_CFG` macro can be used for aDMaps or AVMaps. For example:

```
DQ_AVMAP_CLK_CFG(2, 0); // aXMap uses external clock on sync line 2 with no divider
DQ_AVMAP_CLK_CFG(3, 1); // aXMap uses external clock on sync line 3 with divider=2
                        //  (half rate of clock on sync line 3)
```

**Note:**

The three modes where the IOM emits packets are

- **Timed mode**: Sets the rate at which you want to receive packets in Hz. The rate is set with the `rate` parameter. Different aD/VMaps can be set to different rates but will be rounded off into multiples of the fastest rate. Care must be taken to choose the rate to be slow enough for the host to handle but fast enough to prevent FIFO overflows on the layers.
- **Watermark mode**: Sets the number of samples you want in the FIFO before the aVMap packet is sent. The number of samples is configured with the `emitCfg` parameter. The watermark will be the same for all layers on the aVMap and must be smaller than the FIFO size of the layer with the smallest FIFO. The first layer to reach this watermark will trigger an interrupt, and the IOM will send as much data as possible from all the layers on the aVMap. You may not get the same amount of data on all layers if they are not configured for the same rate or if the packet is processed and one of the layers hasn't completed the current conversion.
- **External clock mode**: the aD/Vmap is refreshed by the pulse programmed on one of the 4 sync bus lines. The clock rate on the sync line can be divided down so the packet will be emitted once every n clock pulses. All affected aV/DMaps must be configured with the same sync line and divider due to hardware limitations.
  In this mode, the `emitCfg` parameter contains both the sync line and divider. The `line` (sync line) is selected in bits 0 through 1. The `div` (divider-1) is selected on bits 2-31. For example, a value of 0x01 will send a packet on every pulse on sync line 1, while a value of 0x05 will send a packet on every other pulse of sync line 1.
  A macro `DQ_AVMAP_CLK_CFG(line, div)` is provided to simplify this configuration. Using the macro would make the above examples `DQ_AVMAP_CLK_CFG(1, 0)` and `DQ_AVMAP_CLK_CFG(1, 1)` respectively.

### 3.13.2 DqRtAXMapEnable

**Syntax:**

```
int DqRtAXMapEnable(int hd, int enable);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int enable` | TRUE to enable FALSE to disable |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| Other negative values | low level IOM error |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

Enables or disables the aDMap/aVMaps on the IOM given by handle. Enable should be set to TRUE to enable an aDMAP/aVMAP and FALSE to disable it.

### 3.13.3 DqRtAXMapSlotAllocate

**Syntax:**

```
int DqRtAXMapSlotAllocate(int hd, uint32 flags, int dmapID,
int slot_uS);
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM |
| `uint32 flags` | Either TRUE (1) to enable or FALSE (0) to clear |
| `int dmapID` | XMap ID to allocate slot for |
| `int slot_uS` | Slot delay from the beginning of the time period, or zero to disable |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| Other negative values | low level IOM error |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function sets the time slot allocated for the particular aDMAP or aVMAP. It also sets the enable flags for a particular aVMAP or aDMAP; therefore, it will need to be called for both timed, clocked and watermark aDMAP or aVMAPs.

### *3.13.4 DqRtAVmapRefreshInputsExt*

**Syntax:**

```
int DqRtAVmapRefreshInputsExt(int handle, int vmapid,
uint32* packet_type, uint16* counter, uint16* tstamp, int* mapid);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int vmapid` | Identifier of the VMAP to refresh |

**Output:**

| | |
|---|---|
| `uint32* packet_type` | Flags for the various packet conditions received |
| `uint16* counter` | Packet number |
| `uint16* tstamp` | Packet timestamp |
| `int* mapid` | The id of the VMAP returned |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_SUCCESS` | command processed successfully |
| `DQ_WAIT_ENDED` | Function call exceeded timeout |

**Description:**

This function refreshes the host's aVMAP and returns with the parameters of the packet. These parameters include the packet number, type of packet, the timestamp of when the packet was sent, and the actual aVMAP id.

`<packet_type>:`

```
#define DQ_AVMAP_INPKT_NEXT     (1)  // input packet is received
                                     //     with a proper counter
#define DQ_AVMAP_INPKT_MISSED   (2)  // input packet is received
                                     //     with a missed counter
#define DQ_AVMAP_INPKT_RESENT   (3)  // missed input packet is received
#define DQ_AVMAP_OUTPKT_ACK     (4)  // output packet acknowledge
#define DQ_AVMAP_OUTPKT_MISSED  (5)  // output packet is missed
                                     //     with the specified counter
#define DQ_AVMAP_TIMEOUT        (6)  // timeout is encountered
                                     //     (also returning DQ_TIMEOUT_ERROR)
#define DQ_AVMAP_IN_ERROR       (7)  // receive function returned an error other than
                                     //     timeout
```

NOTE: Unlike most other functions, `DqRtAVmapRefreshInputsExt` will not return a timeout error (DQ_TIMEOUT_ERROR) upon timeout. Instead the function returns DQ_WAIT_ENDED as it's not necessarily an error upon asynchronous VMAP timeout.

### *3.13.5 DqRtXmapRefreshInputsEx*

**Syntax:**

```
int DqRtXmapRefreshInputsExt(int handle, int flags, int* mapid,
uint16* dqCounter, uint16* dqTStamp);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int flags` | Reserved for future use set to zero |

**Output:**

| | |
|---|---|
| `int* mapid` | Identifier of the DMAP or VMAP received |
| `uint16* dqCounter` | The packet number |
| `uint16* dqTStamp` | The time stamp of the packet |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | invalid IOM handle |
| `DQ_WRONG_DMAP` | Dmap id of received packet does not exist |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This function refreshes a host's aVMAP or aDMAP by receiving any available aDMAP/aVMAP packet. The function fills the appropriate host copy and then returns the VMap or DMap id of the packet received in the `mapid` parameter. The `mapid` parameter is required and cannot be NULL. The parameters `dqCounter` and `dqTStamp` return the packet counter and time stamp respectively. They can be set to NULL if not needed.

NOTE: It is recommended for use with 2 or more aDMAP/aVMAPs.

### *3.13.6 DqRtAsyncXMapInit/DqRtAXMAPInit (deprecated)*

**Syntax:**

```
int DqRtAsyncXMapInit(int hd, int xmapid, uint32 *Config,
uint32 trigType, uint32 amaptype, double rate, uint32 wmrk,
uint32 TrigRoute)
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM |
| `int xmapid` | Identifier of the asynchronous VMAP or DMAP to configure |
| `uint32 *Config` | Requested configuration (layer specific) |
| `uint32 trigType` | Software or rising/falling edge trigger |
| `uint32 amaptype` | Async V/DMAP will run in timed or watermark mode |
| `double rate` | The rate at which a packet will be emitted in Hz |
| `uint32 wmrk` | The FIFO watermark at which a packet is emitted in samples |
| `uint32 TrigRoute` | The source and route of the trigger. |

**Output:**

| | |
|---|---|
| `uint32 *Config` | Actual configuration adjusted based on trigger (layer specific) |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | handle is illegal |
| `DQ_ILLEGAL_ENTRY` | xmapid is illegal |
| `DQ_BAD_PARAMETER` | NULL or 0 in Config,<br>`trigType, amaptype, wmrk, or TrigRoute` value is invalid |
| `DQ_SUCCESS` | command processed successfully |

**Description:**

This is a deprecated function and is replaced by `DqRtAXMapStart`. (It was used to set up an already created VMap or DMap up for Asynchronous/autonomous VMap or DMap operation. The

VMap or DMap can be configured to emit packets at a given frequency or a VMap can emit packets on a FIFO watermark. The function also configures the layer and sets the trigger options.)

`<Config>`:

```
#define DQ_LN_TMREN      (1L<<11)  // enable layer periodic timer
#define DQ_LN_IRQEN      (1L<<10)  // enable layer irqs
#define DQ_LN_PTRIGEDGE1 (1L<<9)   // stop trigger edge MSB
#define DQ_LN_PTRIGEDGE0 (1L<<8)   // stop trigger edge: 00 - software, 01 - rising,
                                   // 02 - falling
#define DQ_LN_STRIGEDGE1 (1L<<7)   // start trigger edge MSB
#define DQ_LN_STRIGEDGE0 (1L<<6)   // start trigger edge: 00 - software, 01 - rising,
                                   // 02 - falling
#define DQ_LN_CVCKSRC1   (1L<<5)   // CV clock source MSB
#define DQ_LN_CVCKSRC0   (1L<<4)   // CV clock source 01 - SW, 10 - HW, 11 -EXT
#define DQ_LN_CLCKSRC1   (1L<<3)   // CL clock source MSB
#define DQ_LN_CLCKSRC0   (1L<<2)   // CL clock source 01 - SW, 10 - HW, 11 -EXT
#define DQ_LN_ACTIVE     (1L<<1)   // "ACT" LED status
#define DQ_LN_ENABLED    (1L<<0)   // enable operations
```

This parameter contains the main configuration flags of the IOM. The parameter will be adjusted based on the values given in `trigType` and `amaptype`. `DqRtVmapSetConfig()` will then be called from this function to initialize the IOM.

`<trigType>`:

```
#define DQ_LN_PTRIGEDGE1 (1L<<9)   // stop trigger edge MSB
#define DQ_LN_PTRIGEDGE0 (1L<<8)   // stop trigger edge: 00 - software, 01 - rising,
                                   // 02 - falling
#define DQ_LN_STRIGEDGE1 (1L<<7)   // start trigger edge MSB
#define DQ_LN_STRIGEDGE0 (1L<<6)   // start trigger edge: 00 - software, 01 – rising
                                   // 02 – falling
```

`<amaptype>`:

```
#define DQ_AVMAP_TYP_WMRK (0)       // send asynchronous on watermark
#define DQ_AVMAP_TYP_TIME (1)       // send asynchronous on timing
```

Note: `DQ_AVMAP_TYP_WMRK` is only valid for VMAP mode

`<rate>`:

This is the rate at which the IOM will send packets when in timed mode. If the rate set here is too high the function call to `DqRtAsyncXMapEnable()` will return with a bad parameter. This parameter is only used if amaptype is set to timed mode otherwise it is ignored.

`<wmrk>`:

This is the number of samples a layer’s FIFO is filled to before a packet is emitted. If it is larger than the FIFO on one of the layers the function will return an error. This parameter is only used if amaptype is set to watermark mode otherwise it is ignored.

`<TrigRoute>`:

```
#define DQ_EXT_EXT0                (0x34)      // ISO_EXT0 line
#define DQ_EXT_EXT1                (0x35)      // ISO_EXT1 line
// SYNCx interface lines
#define DQ_EXT_SYNC0               (0x10)      // SYNC0 line
#define DQ_EXT_SYNC1               (0x11)      // SYNC1 line
#define DQ_EXT_SYNC2               (0x12)      // SYNC2 line
#define DQ_EXT_SYNC3               (0x13)      // SYNC3 line
```

Note: 0 can be passed if using a software trigger.

## 3.14 Offline/Fast Data Conversion Functions

The data conversion API provide the option of postprocessing raw data offline, outside of the real-time acquisition portion of an application. Alternatively, data conversion API can be used inside your application to improve data conversion time by 3x or 4x on UEIPAC deployments. The API accesses device-specific channel setup and calibration data structures allowing a fast conversion of the raw data to scaled data.

### 3.14.1 DqConvCopyDataGetMemsize

**Syntax:**

```
int DAQLIB DqConvCopyDataGetMemsize(int hd, int devn, int* memsize)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int* memsize` | Pointer to size of the memory to allocate for `DATACONV` structure |

**Output:**

| | |
|---|---|
| `int* memsize` | size of the memory to allocate for `DATACONV` structure |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_NOT_IMPLEMENTED` | error in data structure |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Returns amount of memory to be allocated in user space in bytes for the `DATACONV` structure for an I/O board (`devn`).

**Note:**

Use this function to obtain the size of memory buffer needed to store calibration / channel configuration data for a device.

The `DqConvCopyDataGetMemsize` API is used with `DqConvCopyDataConv:` `DqConvCopyDataGetMemsize` returns the required size of a memory buffer, and `DqConvCopyDataConv` copies the calibration/setup structure to memory of that size to be used for converting raw data offline.

### *3.14.2 DqConvCopyDataConv*

**Syntax:**

```
int DAQLIB DqConvCopyDataConv(int hd, int devn, char* mem, int memsize)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `char* mem` | pointer to the memory to copy `DATACONV` and associated structures |
| `int memsize` | size of the memory allocated for `DATACONV` structure |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | insufficient memory to perform command |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_NOT_IMPLEMENTED` | error in data structure |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Copies the `DATACONV` structure (calibration / configuration data structure) to user application memory (`mem`). The API also adjusts pointers to point into the copied calibration data structures instead of pointing into DLL-allocated memory.

The `DATACONV` structure can be copied for off-line data conversion in the future.

**Note:**

If the data is not used in the same program that it was called from (i.e. the structure was saved into a file and later restored), `DqConvFormatPdcFromBinary()` needs to be called to assign correct pointers to the conversion routines.

Note that `DqConvCopyDataConv` must be called after the board has been completely configured including channel list setup and any specific clocking configuration (i.e. 1PPS or PTP synchronization). This is to ensure the correct calibration structure is used when converting data and timestamps.

### 3.14.3 DqConvGetDataConv

**Syntax:**

```
int DAQLIB DqConvGetDataConv(int hd, int devn, pDATACONV* pdc)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `pDATACONV* pdc` | pointer to the conversion structure in DLL memory |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_NOT_IMPLEMENTED` | error in data structure |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Returns a pointer to `DATACONV` structure for the particular IOM and device from DLL-allocated memory. This is required if the conversion functions will later be called in the same program.

**Note:**

This function is used for applications that will acquire data and convert it to scaled values in the same program (not offline). `DqConvGetDataConv` is used with `DqConvRaw2ScalePdc` or `DqConvScale2RawPdc. (DqConvGetDataConv` points to the appropriate calibration / configuration structure, and `DqConvRaw2ScalePdc` does the actual conversion.)

Note that `DqConvGetDataConv` must be called after the board has been completely configured including channel list setup and any specific clocking configuration (i.e. 1PPS or PTP synchronization). This is to ensure the correct calibration structure is used when converting data and timestamps.

### *3.14.4 DqConvRaw2ScalePdc*

**Syntax:**

```
int DAQLIB DqConvRaw2ScalePdc(pDATACONV pdc, uint32* chlst, uint32
chlstsz, uint32 bufsz, char* raw, double* scaled)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `pDATACONV* pdc` | pointer to the conversion structure |
| `uint32* chlst` | input channel list for the device |
| `uint32 chlstsz` | channel list size |
| `uint32 bufsz` | buffer size in samples |
| `char* raw` | pointer to the raw buffer (where the binary data stored) |
| `double* scaled` | pointer to the double buffer where to store converted data |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | error in `pdc` |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Converts a buffer of raw binary data into scaled data (input). Function performs ntohx() conversion and data word size recognition by itself.

**Note:**

This function can be used for off-line conversion (after `DqConvFormatPdcFromBinary`) or in the data acquisition program itself (after `DqConvGetDataConv` and after data has been acquired). This function can be used as a replacement for `DqAdvRawToScaleValue` and is more efficient in data conversion.

### 3.14.5 DqConvRaw2ScaleDev

**Syntax:**

```
int DAQLIB DqConvRaw2ScaleDev(int hd, int devn, uint32* chlst, uint32
chlstsz, uint32 bufsz, char* raw, double* scaled)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32* chlst` | input channel list for the device |
| `uint32 chlstsz` | channel list size |
| `uint32 bufsz` | buffer size in samples |
| `char* raw` | pointer to the raw buffer (where the binary data stored) |
| `double* scaled` | pointer to the double buffer where to store converted data |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_NOT_IMPLEMENTED` | error in data structure |
| `DQ_BAD_PARAMETER` | error in size parameter |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Converts a buffer of raw binary data into scaled data (input). Function performs `ntohx()` conversion and data word size recognition by itself.

**Note:**

This function is a version of the `DqConvRaw2ScalePdc()` but takes a handle to the IOM and a device number instead of the `pDATACONV pdc` (`DqConvRaw2ScalePdc` is called inside this function).

This function cannot be used for off-line data conversion.

### 3.14.6 DqConvScale2RawPdc

**Syntax:**

```
int DAQLIB DqConvScale2RawPdc(pDATACONV pdc, uint32* chlst,
uint32 chlstsz, uint32 bufsz, double* scaled, char* raw)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `pDATACONV* pdc` | pointer to the conversion structure |
| `uint32* chlst` | input channel list for the device |
| `uint32 chlstsz` | channel list size |
| `uint32 bufsz` | buffer size in samples |
| `double* scaled` | pointer to the double buffer where to store converted data |
| `char* raw` | pointer to the raw buffer (where the binary data stored) |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | error in `pdc` |
| `DQ_NOT_IMPLEMENTED` | `pdc` is NULL |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Converts a buffer of scaled into raw binary data (output). Function performs `ntohx()` conversion and data word size recognition by itself.

**Note:**

See notes for `DqConvRaw2ScalePdc().` This function can be used for off-line conversion. This function can be used as a replacement for `DqAdvScaleToRawValue` and is more efficient in data conversion.

### 3.14.7 DqConvScale2RawDev

**Syntax:**

```
int DAQLIB DqConvScale2RawDev(int hd, int devn, uint32* chlst,
uint32 chlstsz, uint32 bufsz, double* scaled, char* raw)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32* chlst` | input channel list for the device |
| `uint32 chlstsz` | channel list size |
| `uint32 bufsz` | buffer size in samples |
| `double* scaled` | pointer to the double buffer where to store converted data |
| `char* raw` | pointer to the raw buffer (where the binary data stored) |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_NOT_IMPLEMENTED` | error getting IOM data |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Converts a buffer of scaled data into raw binary data (output). Function performs ntohx() conversion and data word size recognition by itself.

**Note:**

This function is a version of the `DqConvScale2RawPdc` but takes a handle to the IOM and a device number instead of the `pDATACONV pdc` (`DqConvScale2RawPdc` is called inside this function).

This function cannot be used for off-line data conversion.

### 3.14.8 DqConvFillConvData

**Syntax:**

```
int DAQLIB DqConvFillConvData(int hd, int dev, int ss, uint32* chlst,
uint32 chlstsz)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ss` | Subsystem: either `DQ_SS0IN` or `DQ_SS0OUT` |
| `uint32* chlst` | input channel list for the device |
| `uint32 chlstsz` | channel list size |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | insufficient memory to perform command |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_NOT_IMPLEMENTED` | error in data structure |
| `DQ_BAD_DEVN` | device indicated by devn does not exist |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function fills `DATACONV` for `<handle>` IOM and `<devn>` device.

**Note:**

If `DqConv...()` routines are used to convert data while VMap/DMap is actively running this function is called automatically by DAQLIB to fill tables for internal data conversions. For ACB mode, you should call this function after the board has been completely configured including channel list setup and any specific clocking configuration (i.e. 1PPS or PTP synchronization). This is to ensure the correct calibration structure is used when converting data and timestamps.

### *3.14.9 DqConvFormatPdcFromBinary*

**Syntax:**

```
int DAQLIB DqConvFormatPdcFromBinary(pDATACONV pdc)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `pDATACONV* pdc` | pointer to the conversion structure |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function fills `DATACONV` with pointers to the device-specific conversion function.

**Note:**

When restoring calibration/configuration `DATACONV` data structure from the file for off-line conversion, you must call this function.

# 4 Layer specific functions

## 4.1 DNx-MF-101 layer

### 4.1.1 DqAdv101AIRead

**Syntax:**

```
int DqAdv101AIRead(int hd, int devn, int CLSize, uint32* cl,
uint32* bData, double* fData)
```

**Command:**

Retrieve analog input data.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Number of channels |
| `uint32* cl` | Pointer to channel list |

**Output:**

| | |
|---|---|
| `uint32* bData` | Calibrated binary data |
| `double* fData` | Converted voltage data (pass `NULL` if not required) |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, bdata is `NULL`, or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

When this function is called for the first time, the firmware stops any ongoing operation on the device specified and reprograms it according to the channel list supplied. The layer continuously acquires data and every subsequent call to the function returns the latest acquired data.

`<cl>`

The channel list (`cl`) is an array of uint32 elements with a length `CLSize` from 1 to 17. The DNx-MF-101 supports up to 16 single-ended or 8 differential analog input channels, plus an additional timestamp channel. Each element of the channel list contains a bit-packed word that defines the channel number, the gain, and whether the channel is single-ended or differential.

- Gain: To set the gain of a particular channel, use the `DQ_LNCL_CHANGAIN()`macro. For example,

```
c[1]=DQ_LNCL_CHANGAIN(5, DQ_MF101_AI_GAIN_4);
```

configures the second channel to read a single-ended input on pin AIn SE5 with a gain of 4x.

Available gain settings are:

```
DQ_MF101_AI_GAIN_1     // (+-10 V, or +-80 V with divider)
DQ_MF101_AI_GAIN_4     // (+-2.5 V, or +-20 V with divider)
DQ_MF101_AI_GAIN_16    // (+-0.625 V, or +-5 V with divider)
DQ_MF101_AI_GAIN_64    // (+-0.15625 V, or +-1.25 V with divider)
```

Input values should fall within the lower range when the 1/8th divider is turned off. To use the higher range, enable the divider using `DqAdv101AISetConfig()` .

- Single-ended vs Differential: To read inputs in differential mode, OR the channel list element with `DQ_LNCL_DIFF`. For example,

  ```
  c[2]=DQ_LNCL_CHANGAIN(0, DQ_MF101_AI_GAIN_4)|DQ_LNCL_DIFF;
  ```

  configures the third channel to read a differential input on pins AIn DF0+ and AIn DF0-. Differential channel numbers go from 0...7. Single-ended channel numbers go from 0...15. It is possible to combine single-ended and differential readings, for example having 2 differential channels on four pins and 12 single-ended channels on the remaining pins.

- Timestamps: To timestamp the data, add a channel and set it to `DQ_LNCL_TIMESTAMP`. The timestamp is reset by the firmware on the first read of a new channel list. There is some delay between the reset and the first read of the timestamp, so the first data point may not correspond to t=0. The timestamp channel's `fdata` contains the elapsed time in seconds.

`<bdata>`

Binary data is returned as:

- `Bit 31`, `DQ_MF101_AI_DATA_READY`: =1 means the data was updated since the last read
- `Bit 30`, `DQ_MF101_AI_DATA_PGA_EF`: =1 means the current sample exceeded the input range
- `Bits 17…0`: ADC data

**Notes:**

- For use with point-by-point DAQ mode only. The user cannot perform this function call when the layer is involved in any streaming or data mapping operations.
- If the user specifies a short timeout delay, this function can time out when called for the first time. This is because the function is executed as a pending command, and layer programming takes up to 10ms.

### *4.1.2 DqAdv101AISetConfig*

**Syntax:**

```
int DqAdv101AISetConfig(int hd, int devn, uint32 divider_mask,
uint32 moving_average)
```

**Command:**

Configure 1/8th voltage divider and moving average.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |

| | |
|---|---|
| `int devn` | Layer inside the IOM |
| `uint32 divider_mask` | Bitmask to enable 1/8th voltage divider on each channel (0=disable, 1=enable) |
| `uint32 moving_average` | Moving average window size; select one of:<br>`DQ_MF101_AI_MAV_1     //1 sample`<br>`DQ_MF101_AI_MAV_2     //2 samples`<br>`DQ_MF101_AI_MAV_4     //4 samples`<br>`DQ_MF101_AI_MAV_8     //8 samples`<br>`DQ_MF101_AI_MAV_16    //16 samples`<br>`DQ_MF101_AI_MAV_32    //32 samples`<br>`DQ_MF101_AI_MAV_64    //64 samples`<br>`DQ_MF101_AI_MAV_128   //128 samples`<br>`DQ_MF101_AI_MAV_156   //256 samples` |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | Value of `moving_average` is invalid |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function sets configuration parameters for an analog input channel on the MF-101.

`<divider_mask>`

When enabled, the voltage divider reduces the voltage on the channel by a factor of 8. The divider is enabled/disabled individually on each input pin. Therefore, to use the divider for differential channel 7, enable dividers for single-ended channels 14 and 15. For example, `divider_mask = 0xfff8` enables the divider on pins 15...3 and disables the divider on pins 2...0. An input signal of 80 V on pins 15...3 is divided down to 10 V.

`<moving_average>`

Enabling the moving average can smooth out noise from the sensor input line. This function sets the number of samples used for the moving average as a power of 2 (from 0 to 8). For example, 0 is off/every sample and 3 is every $2^3$=8 samples. The averaging buffer is filled at the `DQ_MF101_SS_AI` subsystem clock rate (2kHz default). The first value fills the entire buffer, and these values are replaced as more data points are acquired.

## *4.1.3 DqAdv101AOWrite*

**Syntax:**

```
int DqAdv101AOWrite(int hd, int devn, int CLSize, uint32* cl,
uint16* bData, double* fData)
```

**Command:**

Write floating point or raw data to output channels.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Size of channel list |
| `uint32* cl` | List of output channels |
| `uint16* bData` | Pointer to binary data array `bData[CLSize]`, or `NULL` if `fData` is used |
| `double* fData` | Pointer to voltage or current data array `fData[CLSize]`, or `NULL` if `bData` is used |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MF101_AO_CHAN`, `bData` and `fData` are `NULL`, or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

When this function is called for the first time, the firmware stops any ongoing operation on the device. The firmware then parses the channel list and writes the data one by one. If `bData` and `fData` are both non-NULL, `fData` is used.

The AO subsystem must be configured through `DqAdv101AOSetConfig()` before calling this function. The `fData` values will either be in volts or milliamps depending on the mode selected in `DqAdv101AOSetConfig()`.

**Notes:**

- For use with point-by-point DAQ mode only. The user cannot perform this function call when the layer is involved in any streaming or data mapping operations.

### *4.1.4 DqAdv101AOSetConfig*

**Syntax:**

```
int DqAdv101AOSetConfig(int hd, int devn, uint chan, uint32
range)
```

**Command:**

Set analog output mode and range.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 chan` | Channel number |
| `uint32 range` | Output mode and range, selected among `DQ_MF101_AO_RANGE_` defines |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | Invalid channel, mode or range combination |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function sets an individual analog output channel to one of the following modes:

```
DQ_MF101_AO_RANGE_PN_5V       // +-5V
DQ_MF101_AO_RANGE_PN_10V      // +-10V
DQ_MF101_AO_RANGE_0_20MA      // 0 to 20mA
DQ_MF101_AO_RANGE_4_20MA      // 4 to 20mA
DQ_MF101_AO_RANGE_N_1_P_22MA  // -1 to +22mA
DQ_MF101_AO_RANGE_OFF         // disable output channel
DQ_MF101_AO_RANGE_EN_PD       // enable 30kΩ pulldown resistor to ground on
                                 output
DQ_MF101_AO_RANGE_OVERRANGE_EN // OR with selected range to enable 20% overrange on
                                 voltage ranges. Overrange is uncalibrated and should
                                 only be used for diagnostics.
                                 // +-5V  -> +-6V
                                 // +-10V -> +-12V
```

Channels do not need to share the same mode.

## 4.1.5 DqAdv101AOEnableWForm

**Syntax:**

```
int DqAdv101AOEnableWForm(int hd, int devn, uint32 cfg)
```

**Command:**

Enable or disable waveform output.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |

| | |
|---|---|
| `int devn` | Layer inside the IOM |
| `uint32 cfg` | Bitmask to enable waveform on output channels (1=enable, 0=disable) |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function outputs the waveform data written by `DqAdv101AOWriteWForm().` Channels are enabled or disabled according to the `cfg` bitmask.

To output a repetitive waveform instead of a one-shot output, add a reload flag to the bitmask:

```
cfg |= DQ_MF101_AO_WFORM_RELOAD;
```

To set the waveform frequency, configure the `DQ_MF101_SS_AO` subsystem clock using `DqCmdSetClock().` Use `DQ_MF101_CLKID_SS` as the "clocksel" parameter in the clock configuration structure.

## *4.1.6 DqAdv101AOWriteWForm*

**Syntax:**

```
int DqAdv101AOWriteWForm(int hd, int devn, int CLSize, uint32*
cl, int datasize, uint16** bData, double** fData)
```

**Command:**

Write waveform data to memory.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Size of channel list |
| `uint32* cl` | List of output channels |
| `int datasize` | Size of data to write, max `DQ_MF101_AO_FIFO_SZ` (4096). |
| `uint16** bData` | Pointer to binary data arrays `bData[CLSize][datasize]`, or `NULL` if `fData` is used |
| `double** fData` | Pointer to voltage or current data arrays `fData[CLSize][datasize]`, or `NULL` if `bData` is used |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MF101_AO_CHAN`, `bData` and `fData` are `NULL`, or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function loads waveform data into the FIFO of the MF-101 AO subsystem. The AO subsystem must be configured through `DqAdv101AOSetConfig()` before calling this function. The `fData` values will either be in volts or milliamps depending on the mode selected in `DqAdv101AOSetConfig()`.

Data is not used until `DqAdv101AOEnableWForm()` enables the FIFO for desired channels.

**Notes:**

- If `bData` and `fData` are both non-NULL, `fData` is used.
- The `datasize` for both channels must be the same.

### 4.1.7 *DqAdv101AOReadAdc*

**Syntax:**

```
int DqAdv101AOReadAdc (int hd, int devn, int CLSize, uint32* cl,
uint16* bData, double* fData)
```

**Command:**

Read diagnostic data on output channels.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Number of diagnostic channels (max size = 9) |
| `uint32* cl` | Pointer to diagnostic channel list |

**Output:**

| | |
|---|---|
| `uint32* bData` | Calibrated binary data |
| `double* fData` | Converted voltage data (pass `NULL` if not required) |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF- |

| | |
|---|---|
| | 101 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, bdata is `NULL`, or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function retrieves ADC data for each diagnostic channel specified in `cl`, converts it, and stores data in raw and float formats. Each element of `cl` consists of the output channel number (either 0 or 1) OR'd with one of the following modes:

```
DQ_MF101_AO_ADC_CHAN_TEMP         // DAC main die temperature (degrees C)
DQ_MF101_AO_ADC_CHAN_VSENSE_P     // Voltage on AOut (volts)
DQ_MF101_AO_ADC_CHAN_VSENSE_N     // Voltage on AGnd (volts)
DQ_MF101_AO_ADC_CHAN_VDPC_P       // DAC supply voltage (volts)
```

For example, `cl[3] = DQ_MF101_AO_ADC_CHAN_TEMP | 1` sets the fourth diagnostic channel to read temperature on the DAC for AOut 1.

The ADCs are normally OFF at power-up. Because the ADCs start up when a changed channel list is presented, discard the first set of data read from a new channel list.

Both ADCs will continue to run until this function is called with CLSize = 1 and cl[0] = DQ_MF101_AO_ADC_DISABLE. Both ADCs are enabled/disabled together, though you can choose to read from only one of them when enabled.

Timestamps:

To timestamp the data, set a diagnostic channel to `DQ_LNCL_TIMESTAMP`. The timestamp is reset by the firmware on the first read of a new channel list. There is some delay between the reset and the first read of the timestamp, so t = 0 does not necessarily equal the first data point received. The timestamp's `fdata` contains the elapsed time in seconds.

**Notes:**

- If this function is called at a rate that does not allow enough time for the ADC to sample and convert new data, old data will be returned. New data is indicated by the presence of a '1' in the MSBit of `bdata`. The AO ADC sampling rate is fixed at 2kHz.
- For use with point-by-point DAQ mode only

### 4.1.8 DqAdv101DIRead

**Syntax:**

```
int DqAdv101DIRead (int hd, int devn, uint32* state, uint32*
debounced)
```

**Command:**

Read the current and debounced digital input state.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `uint32* state` | Bitmask of current state |
| `uint32* debounced` | Bitmask of debounced state |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

The function returns the current and debounced state of the 16 digital input channels:

- `state`: This is the result of comparing the current ADC voltage to the levels set by `DqAdv101DISetLevels()`.
- `debounced`: This logic level must have held steady over the number of samples defined by `DqAdv101SetDebouncer()`.

**Notes:**

- For use with point-by-point DAQ mode only

### *4.1.9 DqAdv101DIReadAdc*

**Syntax:**

```
int DqAdv101DIReadAdc (int hd, int devn, int CLSize, uint32* cl,
uint32* bData, double* fData)
```

**Command:**

Read the raw and float values on digital inputs.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Number of channels |
| `uint32* cl` | Pointer to channel list |

**Output:**

| | |
|---|---|
| `uint32* bData` | Calibrated binary data |
| `double* fData` | Converted voltage data (pass `NULL` if not required) |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, bdata is `NULL`, or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
This function retrieves ADC data for each digital input channel specified in `cl`, converts it, and stores data in raw and float formats.

To timestamp the data, add another channel to the channel list and set it `DQ_LNCL_TIMESTAMP`. The timestamp is reset by the firmware on the first read of a new subchannel list. There is some delay between the reset and the first read of the timestamp, so t = 0 does not necessarily equal the first data point received. The timestamp's `fdata` contains the elapsed time in seconds.

**Notes:**

- For use with point-by-point DAQ mode only
- To change the DI ADC sampling rate (default 400kHz), configure the `DQ_MF101_SS_GUARDIAN` subsystem clock using `DqCmdSetClock()`. Use `DQ_MF101_CLKID_SS` as the "clocksel" parameter in the clock configuration structure.

### *4.1.10 DqAdv101DISetDebouncer*

**Syntax:**

```
int DqAdv101DISetDebouncer (int hd, int devn, uint32 channel,
uint16 samples)
```

**Command:**
Configure debouncing interval for a single channel

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 channel` | Channel number |
| `uint16 samples` | Number of ADC samples before change of state |

**Output:**
none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |

| | |
|---|---|
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
This function sets the debouncing interval for a single digital input channel. The debouncer changes its state only when the input signal remains stable at the new level for the specified number of samples. The debouncing time can be calculated from the ADC sampling rate. The ADC sampling rate is either the default 400kHz or a custom `Dq_MF101_SS_GUARDIAN` rate configured using `DqCmdSetClock().`Use `DQ_MF101_CLKID_SS` as the “clocksel” parameter in the clock configuration structure.

### *4.1.11 DqAdv101DISetLevels*

**Syntax:**

```
int DqAdv101DISetLevels (int hd, int devn, uint32 channel, float
low, float high)
```

**Command:**
Set high and low logic levels.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 channel` | Channel number (0…15) |
| `float low` | Logic low level (between 0 to 55 V) |
| `float high` | Logic high level (between 0 to 55 V) |

**Output:**
none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | Channel number is too high |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
This function sets the low and high voltage levels for a single channel. The hysteresis is programmed to change the detected logic level from 0 to 1 when the measured input voltage is above the high limit. The logic level changes from 1 to 0 when the input voltage is below the low limit. If the signal is in between the low and high limits, the detector maintains the previous logic level. This feature increases immunity to noise on the input lines.

### *4.1.12 DqAdv101DOWrite*

**Syntax:**

```
int DqAdv101DOWrite (int hd, int devn, uint16 low_cfg, uint16
high_cfg)
```

**Command:**

Set digital output state to 0, 1, or turned off.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint16 low_cfg` | Bitmask to pull channel low (1=pull low, 0=ignore) |
| `uint16 high_cfg` | Bitmask to pull channel high (1=pull high, 0=ignore) |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function sets the state of the digital output channels. The `low_cfg` and `high_cfg` bitmasks control a channel's lower and upper FETs respectively. The LSB is DIO 0.

| `low_cfg` | `high_cfg` | Output state |
|---|---|---|
| 0 | 0 | off |
| 1 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 1 | off |

The logic prevents both FETs from being on concurrently. Thus, if `low_cfg` and `high_cfg` are both 1 or both 0 on a given channel, both FETs are turned off (non-conducting). Turning off the output configures the channel as input-only. Use `DqAdv101DOSetTermination()` to enable pull up/down resistors on the inputs.

**Notes:**

- For use with point-by-point DAQ mode only

### *4.1.13 DqAdv101DORead*

**Syntax:**

```
int DqAdv101DORead (int hd, int devn, uint16* low_state, uint16*
high_state)
```

**Command:**

Read the last state written to digital outputs.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `uint16* low_state` | Last state written to the low-side FETs |
| `uint16* high_state` | Last state written to the high-side FETs |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

Returns the most recent state written to the DO subsystem by `DqAdv101DOWrite()`. The states written to the low-side and high-side FETs correspond to the output state as follows:

| `low_state` | `high_state` | Output state |
|---|---|---|
| 0 | 0 | off |
| 1 | 0 | LOW |
| 0 | 1 | HIGH |
| 1 | 1 | off |

**Notes:**

- For use with point-by-point DAQ mode only

### *4.1.14 DqAdv101DOSetTermination*

**Syntax:**

```
int DqAdv101DOSetTermination (int hd, int devn, uint32 cfg)
```

**Command:**

Configure pull-up/down resistors.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 cfg` | Bitmask to enable pull up/down resistors (2 bits per channel) |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF- |

| | |
|---|---|
| | 101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
This function enables or disables the 98kohm pull-up and pull-down resistor on each digital channel. Each channel has 2 bits in the `cfg` bitmask: the lower bit controls the pull-down resistor (1=enable, 0=disable), and the higher bit controls the pull-up resistor. It is possible for one resistor to be enabled, both enabled, or both disabled. For example, `cfg=0x42` (01000010) sets ch0 as pull-up, sets ch3 as pull-down, and disables the pull-up/down resistors on all other channels.

### *4.1.15 DqAdv101DOSetPWM*

**Syntax:**

```
int DqAdv101DOSetPWM (int hd, int devn, int count, pMF101DOPWM
settings, float* act_duty_cycle, uint32* act_period_us)
```

**Command:**
Configure pulse width modulation on industrial digital outputs.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int count` | Number of channels, equal to size of `MF101DOPWM` array |
| `pMF101DOPWM settings` | Array of `MF101DOPWM` structures with settings for each PWM channel |

**Output:**

| | |
|---|---|
| `float* act_duty_cycle` | Actual duty cycle for each PWM channel |
| `uint32* act_period_us` | Actual period used for the layer |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | Invalid setting in `MF101DOPWM` structure |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

The MF-101 supports continuous PWM and soft start/stop digital outputs. This function is used in conjunction with `DqAdv101DOWrite()`. To specify the mode of operation for each channel, create an array of one or more `MF101DOPWM` structures.

```
    typedef struct {
       uint8 channel;        // channel to configure PWM
 uint8 pwm_mode;       // soft start/stop or base PWM mode
 uint8 output_mode;    // push, pull, or push-pull output mode
 uint32 period_us;     // PWM period in us
 float duty_cycle;     // duty cycle from 0.0-1.0 (0%-100%)
       uint32 soft_start_us; // soft start duration in us; unused in PWM_MODE and
                             PWM_MODE_GATED
 uint32 soft_stop_us;  // soft stop duration in us; unused in PWM_MODE and
                             PWM_MODE_GATED
    } MF101DOPWM, *pMF101DOPWM;
```

`MF101DOPWM` parameters:

- `<channel>`
  Select a channel number between 0 and 15.
- `<pwm_mode>`
  Select one of the following modes of operation:

| | |
|---|---|
| DQ_MF101_DO_PWM_DISABLED | PWM and soft start disabled |
| DQ_MF101_DO_PWM_SOFTSTART | PWM soft-start (output transitions from 0 to 1 over the `soft_start_us` time interval) |
| DQ_MF101_DO_PWM_SOFTSTOP | PWM soft-stop (output transitions from 1 to 0 over the `soft_stop_us` time interval) |
| DQ_MF101_DO_PWM_SOFTBOTH | Use PWM to soft-start and soft-stop |
| DQ_MF101_DO_PWM_MODE | Outputs continuous PWM according to `period_us` and `duty_cycle` parameters. `DqAdv101DOWrite()`value is ignored in this mode. |
| DQ_MF101_DO_PWM_MODE_GATED | Similar to `DQ_MF101_DO_PWM_MODE`, except PWM signal is only ON when `DqAdv101DOWrite()` writes a 1 to the output. |

- `<output_mode>`
  Select one of the following:

| | |
|---|---|
| DQ_MF101_DO_PWM_OFF | No PWM applied to high or low side FETs |
| DQ_MF101_DO_PWM_PULL | Switch low-side FET, ignore high-side FET |
| DQ_MF101_DO_PWM_PUSH | Switch high-side FET, ignore low-side FET |
| DQ_MF101_DO_PWM_PUSH_PULL | Switch both high-side and low-side FETs |

- `<period_us>`
  Sets the desired PWM period in microseconds (min 5, max 254200).
- `<duty_cycle>`
  Sets the duty cycle as a fraction of a period. For example, a value of 0.25 keeps the output high 25% of the time.
- `<soft_start_us>`

Defines the duration of the full soft-start cycle in microseconds. Whenever `DqAdv101DOWrite()`changes the channel state from low to high, there will be a PWM signal on the output. The duty cycle of this PWM signal gradually transitions from 0% to `duty_cycle` over `soft_start_us`. Therefore, `soft_start_us` should be longer than `period_us`. This parameter is only used when `pwm_mode` = `DQ_MF101_DO_PWM_SOFTSTART` or `DQ_MF101_DO_PWM_SOFTBOTH`.

- `<soft_stop_us>`
  Defines the duration of the full soft-stop cycle in microseconds. Whenever `DqAdv101DOWrite()`changes the channel state from high to low, there will be a PWM signal on the output. The duty cycle of this PWM signal gradually transitions from `duty_cycle` to 0% over `soft_start_us`. Therefore, `soft_stop_us` should be longer than `period_us`. This parameter is only used when `pwm_mode =` `DQ_MF101_DO_PWM_SOFTSTOP` or `DQ_MF101_DO_PWM_SOFTBOTH`.

**Notes:**

- This function implements the requested `period_us` and `duty_cycle` by counting off 66MHz clock cycles with a 24-bit counter. If the requested values do not convert into a whole number of clock cycles, there may be a small discrepancy between the requested values and the actual values.

### 4.1.16 DqAdv101TTLRead

**Syntax:**

`int DqAdv101TTLRead (int hd, int devn, uint32* read)`

**Command:**

Read the state of TTL and TRIGIN lines.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `uint32* read` | Bitmask of TTL state where 1=high, 0=low. Bits 0-3 are TTL lines 0-3, bit 4 is TRIGIN. |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function reads the state of all TTL lines and TRIGIN at the rate set by `DqCmdSetClock()`.

**Notes:**

- For use with point-by-point DAQ mode only

### *4.1.17 DqAdv101TTLSetConfig*

**Syntax:**

```
int DqAdv101TTLSetConfig (int hd, int devn, uint32 output_1_0,
uint32 output_3_2);
```

**Command:**

Enable TTL outputs in pairs.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 output_1_0` | =1 enables output on TTL1 and TTL0, =0 disables output |
| `uint32 output_3_2` | =1 enables output on TTL3 and TTL2, =0 disables output |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

Configures TTL lines as input or output in pairs (TTL0-1 and TTL2-3). Use `DqAdv101TTLWrite()` to set the digital state of the outputs.

### *4.1.18 DqAdv101TTLWrite*

**Syntax:**

```
int DqAdv101TTLWrite (int hd, int devn, uint32 state)
```

**Command:**

Set state of TTL outputs.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 state` | Bitmask of TTL state where 1=high, 0=low. Bits 0-3 control TTL lines 0-3, bit 4 controls TRIGOUT |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function sets the digital state of TTL outputs and TRIGOUT. Before calling this function, use `DqAdv101TTLSetConfig()` to set desired TTL lines as outputs.

**Notes:**

- For use with point-by-point DAQ mode only

### *4.1.19 DqAdv101CTSetSource*

**Syntax:**

```
int DqAdv101CTSetSource (int hd, int devn, int ct, uint32 input,
uint32 gate)
```

**Command:**

Connect DIO pins to CLKIN and GATE.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ct` | Counter to configure (0 or 1) |
| `uint32 input` | Source to use for CLKIN, selected from `DQ_MF101_CT_SOURCE_` defines |
| `uint32 gate` | Source to use for GATE, selected from `DQ_MF101_CT_SOURCE_` defines |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function sets the input and gate sources on an MF-101 counter. Available sources:

```
DQ_MF101_CT_SOURCE_FET15    // FET-based DIO pins
DQ_MF101_CT_SOURCE_FET14    //
DQ_MF101_CT_SOURCE_FET13    //
DQ_MF101_CT_SOURCE_FET12    //
DQ_MF101_CT_SOURCE_FET11    //
DQ_MF101_CT_SOURCE_FET10    //
DQ_MF101_CT_SOURCE_FET9     //
DQ_MF101_CT_SOURCE_FET8     //
DQ_MF101_CT_SOURCE_FET7     //
DQ_MF101_CT_SOURCE_FET6     //
DQ_MF101_CT_SOURCE_FET5     //
DQ_MF101_CT_SOURCE_FET4     //
DQ_MF101_CT_SOURCE_FET3     //
DQ_MF101_CT_SOURCE_FET2     //
DQ_MF101_CT_SOURCE_FET1     //
DQ_MF101_CT_SOURCE_FET0     //
DQ_MF101_CT_SOURCE_SYNC3    // SYNC pins (input only)
DQ_MF101_CT_SOURCE_SYNC2    //
DQ_MF101_CT_SOURCE_SYNC1    //
DQ_MF101_CT_SOURCE_SYNC0    //
DQ_MF101_CT_SOURCE_LTRIGGER // Layer trigger state (0-STOPPED/1-STARTED)
DQ_MF101_CT_SOURCE_TRIGOUT  // TTL TRIGOUT pin
DQ_MF101_CT_SOURCE_TRIGIN   // TTL TRIGIN pin (input only)
DQ_MF101_CT_SOURCE_TTL3     // TTL DIO pins
DQ_MF101_CT_SOURCE_TTL2     //
DQ_MF101_CT_SOURCE_TTL1     //
DQ_MF101_CT_SOURCE_TTL0     //
```

**Notes:**

- When using FET-based sources, be sure to configure input levels using `DqAdv101DISetLevels()` and start the DI ADC using `DqAdv101DIReadAdc()`.

### *4.1.20 DqAdv101CTSetOutput*

**Syntax:**

`int DqAdv101CTSetOutput (int hd, int devn, int ct, uint32 output)`

**Command:**

Connect digital outputs to CLKOUT.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ct` | Counter to configure (0 or 1) |
| `uint32 output` | Mask of sources to use for output |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |

| | |
|---|---|
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function routes one or more outputs to the CLKOUT line of counter `ct`. Only FET-based DIO, TTL, and TRIGOUT sources are available for output. Create the `output` mask by ORing in one or more of the following defines:

```
DQ_MF101_CT_OUTPUT_FET15    // FET-based DIO pins
DQ_MF101_CT_OUTPUT_FET14    //
DQ_MF101_CT_OUTPUT_FET13    //
DQ_MF101_CT_OUTPUT_FET12    //
DQ_MF101_CT_OUTPUT_FET11    //
DQ_MF101_CT_OUTPUT_FET10    //
DQ_MF101_CT_OUTPUT_FET9     //
DQ_MF101_CT_OUTPUT_FET8     //
DQ_MF101_CT_OUTPUT_FET7     //
DQ_MF101_CT_OUTPUT_FET6     //
DQ_MF101_CT_OUTPUT_FET5     //
DQ_MF101_CT_OUTPUT_FET4     //
DQ_MF101_CT_OUTPUT_FET3     //
DQ_MF101_CT_OUTPUT_FET2     //
DQ_MF101_CT_OUTPUT_FET1     //
DQ_MF101_CT_OUTPUT_FET0     //
DQ_MF101_CT_OUTPUT_TRIGOUT  // TTL TRIGOUT pin
DQ_MF101_CT_OUTPUT_TTL3     // TTL DIO pins
DQ_MF101_CT_OUTPUT_TTL2     //
DQ_MF101_CT_OUTPUT_TTL1     //
DQ_MF101_CT_OUTPUT_TTL0     //
```

The output state is controlled by Counter Register 0 (`cr0`) and Counter Register 1 (`cr1`), set by `DqAdv101CTConfigCounter()` or `DqAdv101CTCfgFor_()`. The output stays LOW while the counter counts from the Load Register (`lr`)value up to `cr0`. Upon reaching `cr0,` the output goes HIGH until `cr1`. The output signal may also be configured to invert.

**Notes:**

This function will overwrite the channel settings set by `DqAdv101DOWrite()` and `DqAdv101TTLSetConfig()`. If the counter is enabled, the counter will drive the output FETs and ignore the `DqAdv101DOWrite()` and `DqAdv101TTLSetConfig()` instructions.

### *4.1.21 DqAdv101CTStartCounter*

**Syntax:**

```
int DqAdv101CTStartCounter (int hd, int devn, int ct, int
enable)
```

**Command:**

Start or stop a counter.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |

| | |
|---|---|
| `int devn` | Layer inside the IOM |
| `int ct` | Counter to configure (0 or 1) |
| `int enable` | 1=enable, 0=disable counter |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `ct` is invalid |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

When a counter's `startmode = DQ_MF101_CT_SMSOFT`, the counter will not start until enabled by this function. Set `enable=0` to stop the counter.

### *4.1.22 DqAdv101CTClearCounter*

**Syntax:**

```
int DqAdv101CTClearCounter (int hd, int devn, int ct)
```

**Command:**

Clear the counter.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ct` | Counter to configure (0 or 1) |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | Counter number is invalid |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function stops the counter and sets it back to its initial state. If `startmode = DQ_MF101_CT_SMSOFT`, the counter will need to be started again using `DqAdv101CTStartCounter()`.

### *4.1.23 DqAdv101CTRead*

**Syntax:**

`int DqAdv101CTRead (int hd, int devn, int CLSize, uint32* cl, uint32* data)`

**Command:**

Read count values from counter.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Number of counters in `cl` |
| `uint32* cl` | Pointer to list of counters |

**Output:**

| | |
|---|---|
| `uint32* data` | Data from counters in `cl` |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function reads the current count values from the counters specified in the channel list. The returned `data` may contain up to three uint32s per counter, depending on the counter configuration.

**Notes:**

- For use with point-by-point DAQ mode only

### *4.1.24 DqAdv101CTWrite*

**Syntax:**

`int DqAdv101CTWrite (int hd, int devn, int CLSize, uint32* cl, uint32* data)`

**Command:**

Change CLKOUT signal by writing to CR0 and CR1.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Number of channels in `cl` |

| | |
|---|---|
| `uint32* cl` | Pointer to list of channels; each channel is set to the counter number OR'd with one of the following constants:<br>`DQ_MF101_CT_CHNLTYPE_CR0     //count register 0`<br>`DQ_MF101_CT_CHNLTYPE_CR1     //count register 1` |
| `uint32* data` | Pointer to data values to write (1 uint32 per channel) |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, `data` is NULL, or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
This function writes values to the compare registers specified in the channel list. Each channel corresponds to a compare register on one of the two MF-101 counters. For example, if `cl[3]= 0|DQ_MF_101_CT_CHNLTYPE_CR1` the value in `data[3]` will be written to Counter #0, CR1. The output signal CLKOUT is low while the count is between the load register value and CR0. CLKOUT is high while the count is between CR0 and CR1. In general, `lr` ≤ `cr0` ≤ `cr1`.

**Notes:**

- For use with point-by-point DAQ mode only

### 4.1.25 DqAdv101CTConfigCounter

**Syntax:**

```
int DqAdv101CTConfigCounter (int hd, int devn, int ss, int
counter, int startmode, int sampwidth, int ps, int pc, int cr0,
int cr1, int tbr, int dbg, int dbc, int iie, int gie, int oie,
int mode, int trs, int enc, int gated, int re, int end_mode, int
lr, int* cfg)
```

**Command:**
Configure counter settings.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ss` | Subsystem the counter belongs to:<br>`DQ_SS0IN //subsystem 0 input`<br>`DQ_SS0OUT //subsystem 0 output` |
| `int counter` | Counter to configure (0 or 1) |
| `int startmode` | Counter startup mode, one of `DQ_MF101_CT_SM` |

| | |
|---|---|
| | constants:<br>`DQ_MF101_CT_SMAUTO //counter starts immediately`<br>`DQ_MF101_CT_SMSOFT //counter starts by calling DqAdv101CTStartCounter()`<br>`DQ_MF101_CT_SMHARD //reserved` |
| `int sampwidth` | Currently unused on the MF-101, but must pass in one of:<br>`DQ_LN_DATASZ8   //one byte`<br>`DQ_LN_DATASZ16  //two bytes`<br>`DQ_LN_DATASZ24  //three bytes`<br>`DQ_LN_DATASZ32  //four bytes (default)` |
| `int ps` | 32-bit prescaler divides clock by this factor |
| `int pc` | Period count register, defines number of periods N as measured from the rising edge of CLKIN |
| `int cr0` | Compare register 0, CLKOUT=low between `lr` and `cr0` |
| `int cr1` | Compare register 1, CLKOUT=high between `cr0` and `cr1` |
| `int tbr` | Timebase register; 31-bit value specifies the number of 66MHz clock cycles for a timed measurement |
| `int dbg` | Input debouncing gate register; 19-bit value specifies the number of 66MHz clock cycles for GATE to be stable |
| `int dbc` | Input debouncing clock register; 19-bit value specifies the number of 66MHz clock cycles for CLKIN to be stable |
| `int iie` | TRUE to invert CLKIN pin |
| `int gie` | TRUE to invert GATE pin |
| `int oie` | TRUE to invert CLKOUT pin |
| `int mode` | Counter mode, one of the following `DQ_CM_` constants:<br>`DQ_CM_CT    //basic timer`<br>`DQ_CM_TCT   //triggered timer`<br>`DQ_CM_TPPM  //Timed Pulse Period Measurement`<br>`DQ_CM_TTPPM //triggered Timed Pulse Period Measurement`<br>`DQ_CM_ECT   //external event counter`<br>`DQ_CM_TECT  //triggered event counter`<br>`DQ_CM_HP    //half-period capture`<br>`DQ_CM_THP   //triggered half-period capture`<br>`DQ_CM_NP    //N-period capture`<br>`DQ_CM_TNP   //triggered N-period capture`<br>`DQ_CM_QE    //quadrature encoder` |
| `int trs` | TRUE to use GATE as trigger (only w/ a triggered mode) |
| `int enc` | TRUE to auto-clear counter after `end_mode` and await next trigger (only w/ a triggered mode) |
| `int gated` | TRUE for GATE to enable/disable counter (not supported in triggered modes) |
| `int re` | TRUE to restart counter after `end_mode` condition is met (continuous operation) |
| `int end_mode` | Count termination condition, one of `DQ_EM_` constants:<br>`DQ_EM_CR0    //CR=CR0`<br>`DQ_EM_CR1    //CR=CR1`<br>`DQ_EM_FFF    //CR=0xFFFFFFFF`<br>`DQ_EM_PC     //pc=0`<br>`DQ_EM_TBR    //tbr=0 (count is not returned until tbr=0)`<br>`DQ_EM_GT     //GATE goes from high to low` |

| | |
|---|---|
| `int lr` | Load register; sets the initial value of the counter |

**Output:**

| | |
|---|---|
| `int* cfg` | Configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `counter` number, `mode`, or `end_mode` is invalid, or `cfg` is NULL |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function provides advanced configuration options for an MF-101 counter. The counting `mode` determines what is returned by `DqAdv101CTRead()` and what input parameters are supported. Refer to the `DqAdv101CTCfgFor___()`functions to learn about the different modes and their applications.

| Cfg function | Counter mode |
|---|---|
| `GeneralCounting` | CT/TCT, ECT/TECT |
| `BinCounter` | CT/TCT, ECT/TECT |
| `PeriodMeasurement` | NP/TNP |
| `HalfPeriod` | HP/THP |
| `TPPM` | TPPM/TTPPM |
| `Quadrature` | QE |
| `PWM` | CT/TCT, ECT/TECT |
| `PWMTrain` | CT/TCT, ECT/TECT |

### *4.1.26 DqAdv101CTCfgForGeneralCounting*

**Syntax:**

```
int DqAdv101CTCfgForGeneralCounting (int hd, int devn, int
counter, int startmode, int ps, int cr0, int cr1, int tbr, int
dbg, int dbc, int iie, int gie, int oie, int extclk, int trig,
int trs, int enc, int gated, int re, int end_mode, int lr, int*
cfg)
```

**Command:**

Configure a counter for general counting.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter to configure (0 or 1) |

| | |
|---|---|
| `int startmode` | Counter startup mode, one of the following constants:<br>`DQ_MF101_CT_SMAUTO //counter starts immediately`<br>`DQ_MF101_CT_SMSOFT //counter starts by calling DqAdv101CTStartCounter()` |
| `int ps` | 32-bit prescaler divides CLKIN by this factor if `extclk=1`. If `extclk=0`, divides internal 66MHz clock. |
| `int cr0` | Compare register 0, CLKOUT=low between `lr` and `cr0` |
| `int cr1` | Compare register 1, CLKOUT=high between `cr0` and `cr1` |
| `int tbr` | Timebase register; 31-bit value specifies the number of 66MHz clock cycles for a timed measurement |
| `int dbg` | Input debouncing gate register; 19-bit value specifies the number of 66MHz clock cycles for GATE to be stable |
| `int dbc` | Input debouncing clock register; 19-bit value specifies the number of 66MHz clock cycles for CLKIN to be stable |
| `int iie` | TRUE to invert CLKIN pin |
| `int gie` | TRUE to invert GATE pin |
| `int oie` | TRUE to invert CLKOUT pin |
| `int extclk` | TRUE to use external clock (CLKIN) |
| `int trig` | TRUE to enable trigger |
| `int trs` | TRUE to use GATE as trigger (only if `trig=1`) |
| `int enc` | TRUE to clear counter to `lr` after `end_mode` and await next trigger (only if `trig=1` and `re=1`); call `DqAdv101StartCounter()` to reset trigger |
| `int gated` | TRUE for GATE to enable/disable counter (only if `trig=0`) |
| `int re` | TRUE to restart counter after `end_mode` condition is met |
| `int end_mode` | Stop the count under one of the following conditions:<br>`DQ_EM_CR0    //count reaches cr0 value`<br>`DQ_EM_CR1    //count reaches cr1 value`<br>`DQ_EM_FFF    //count reaches 0xFFFFFFFF`<br>`DQ_EM_TBR    //tbr counts down to 0`<br>`DQ_EM_GT     //GATE goes from high to low` |
| `int lr` | Load register; sets the initial value of the counter |

**Output:**

| | |
|---|---|
| `int* cfg` | Configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `counter` number or `end_mode` is invalid, or `cfg` is NULL |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
This function sets up an MF-101 counter as a general counter without period counting. `DqAdv101CTRead()` returns the current count value.

### 4.1.27 DqAdv101CTCfgForBinCounter

**Syntax:**

```
int DqAdv101CTCfgForBinCounter(int hd, int devn, int counter,
int startmode, int ps, int cr0, int cr1, int tbr, int dbg, int
dbc, int iie, int gie, int extclk, int trig, int trs, int enc,
int gated, int re, int end_mode, int lr, int* cfg)
```

**Command:**

Configure a counter for bin counting.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter to configure (0 or 1) |
| `int startmode` | Counter startup mode, one of the following constants:<br>`DQ_MF101_CT_SMAUTO //counter starts immediately`<br>`DQ_MF101_CT_SMSOFT //counter starts by calling DqAdv101CTStartCounter()` |
| `int ps` | 32-bit prescaler divides CLKIN by this factor if `extclk=1`. If `extclk=0`, divides internal 66MHz clock. |
| `int cr0` | Compare register 0, CLKOUT=low between `lr` and `cr0` |
| `int cr1` | Compare register 1, CLKOUT=high between `cr0` and `cr1` |
| `int tbr` | Time interval for binning, as number of 66MHz clock cycles |
| `int dbg` | Input debouncing gate register; 19-bit value specifies the number of 66MHz clock cycles for GATE to be stable |
| `int dbc` | Input debouncing clock register; 19-bit value specifies the number of 66MHz clock cycles for CLKIN to be stable |
| `int iie` | TRUE to invert CLKIN pin |
| `int gie` | TRUE to invert GATE pin |
| `int extclk` | TRUE to use external clock (CLKIN) |
| `int trig` | TRUE to enable trigger |
| `int trs` | TRUE to use GATE as trigger (only if `trig=1`) |
| `int enc` | TRUE to clear counter to `lr` after `end_mode` and await next trigger (only if `trig=1` and `re=1`); call `DqAdv101StartCounter()` to reset trigger |
| `int gated` | TRUE for GATE to enable/disable counter (only if `trig=0`) |
| `int re` | TRUE to restart counter after `end_mode` condition is met |
| `int end_mode` | Stop the count under one of the following conditions:<br>`DQ_EM_TBR //tbr counts down to 0` |
| `int lr` | Load register; sets the initial value of the counter |

**Output:**

| | |
|---|---|
| `int* cfg` | Configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |

| | |
|---|---|
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `counter` number or `end_mode` is invalid, or `cfg` is NULL |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
When a counter is configured for bin counting, `DqAdv101CTRead()` returns the number of counts after the `tbr` time interval has elapsed. The timebase divider is a 31-bit 66MHz counter, so the maximum `tbr` value is 0x7FFFFFFF (2147483647) corresponding to ~32.5 seconds.

### *4.1.28 DqAdv101CTCfgForPeriodMeasurement*

**Syntax:**

```
int DqAdv101CTCfgForPeriodMeasurement(int hd, int devn, int
counter, int startmode, int pc, int tbr, int dbg, int dbc, int
iie, int gie, int trig, int trs, int enc, int gated, int re, int
end_mode, int* cfg)
```

**Command:**

Configure a counter to measure period.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter to configure (0 or 1) |
| `int startmode` | Counter startup mode, one of the following constants:<br>`DQ_MF101_CT_SMAUTO //counter starts immediately`<br>`DQ_MF101_CT_SMSOFT //counter starts by calling DqAdv101CTStartCounter()` |
| `int pc` | Number of periods to measure |
| `int tbr` | reserved; set to 0 |
| `int dbg` | Input debouncing gate register; 19-bit value specifies the number of 66MHz clock cycles for GATE to be stable |
| `int dbc` | Input debouncing clock register; 19-bit value specifies the number of 66MHz clock cycles for CLKIN to be stable |
| `int iie` | TRUE to invert CLKIN pin |
| `int gie` | TRUE to invert GATE pin |
| `int trig` | TRUE to enable trigger |
| `int trs` | TRUE to use GATE as trigger (only if `trig=1`) |
| `int enc` | TRUE to stop counter after `end_mode` and await next trigger (only if `trig=1`); use `DqAdv101ClearCounter()` to clear count and reset trigger |
| `int gated` | reserved; set to 0 |

| | |
|---|---|
| `int re` | reserved; set to 1 |
| `int end_mode` | Stop the count under one of the following conditions:<br>`DQ_EM_PC //pc=0` |

**Output:**

| | |
|---|---|
| `int* cfg` | Configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `counter` number or `end_mode` is invalid, or `cfg` is NULL |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function configures the counter to measure the period of CLKIN. The counter stores the number of 66MHz clock cycles over which CLKIN is high (`crh`), as well as the number of 66MHz clock cycles over which CLKIN is low (`crl`). The counter continues to increment `crh` and `crl` until a total of `pc-1` number of periods has elapsed. Thus, the average period=(`crh`+`crl`)/[66000000*(`pc`+1)] seconds.

A call to `DqAdv101CTRead()`returns two pieces of data per channel:

1. total length of the positive part of the signal (`crh=data_read[0]`)
2. total length of the negative part of the signal (`crl=data_read[1]`)

The previous measurements are returned until `pc-1` periods have been counted again.

### *4.1.29 DqAdv101CTCfgForHalfPeriod*

**Syntax:**

```
int DqAdv101CTCfgForHalfPeriod(int hd, int devn, int counter,
int startmode, int tbr, int dbg, int dbc, int iie, int gie, int
trig, int trs, int enc, int gated, int re, int end_mode, int*
cfg)
```

**Command:**

Configure a counter to measure pulse width.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter to configure (0 or 1) |
| `int startmode` | Counter startup mode, one of the following constants:<br>`DQ_MF101_CT_SMAUTO //counter starts immediately`<br>`DQ_MF101_CT_SMSOFT //counter starts by calling` |

| | |
|---|---|
| `int tbr` | reserved; set to 0 |
| `int dbg` | Input debouncing gate register; 19-bit value specifies the number of 66MHz clock cycles for GATE to be stable |
| `int dbc` | Input debouncing clock register; 19-bit value specifies the number of 66MHz clock cycles for CLKIN to be stable |
| `int iie` | TRUE to invert CLKIN pin |
| `int gie` | TRUE to invert GATE pin |
| `int trig` | TRUE to enable trigger |
| `int trs` | TRUE to use GATE as trigger (only if `trig=1`) |
| `int enc` | TRUE to stop counter after `end_mode` and await next trigger (only if `trig=1`); use `DqAdv101ClearCounter()` to clear count and reset trigger |
| `int gated` | reserved; set to 0 |
| `int re` | reserved; set to 1 |
| `int end_mode` | reserved; set to `DQ_EM_PC` |

**Output:**

| | |
|---|---|
| `int* cfg` | Configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `counter` number or `end_mode` is invalid, or `cfg` is NULL |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function configures the counter to measure the pulse width of CLKIN. Calling `DqAdv101CTRead()` returns the number of 66MHz clock cycles over which CLKIN is high (`crh`).

### *4.1.30 DqAdv101CTCfgForTPPM*

**Syntax:**

```
int DqAdv101CTCfgForTPPM(int hd, int devn, int counter, int
startmode, int tbr, int dbg, int dbc, int iie, int gie, int enc,
int re, int end_mode, int* cfg)
```

**Command:**

Configure a counter to measure period over a specific time interval.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

| | |
|---|---|
| `int counter` | Counter to configure (0 or 1) |
| `int startmode` | Counter startup mode, one of the following constants:<br>`DQ_MF101_CT_SMAUTO //counter starts immediately`<br>`DQ_MF101_CT_SMSOFT //counter starts by calling DqAdv101CTStartCounter()`<br>`DQ_MF101_CT_SMHARD //reserved` |
| `int tbr` | Time interval for period measurement, specified as the number of 66MHz clock cycles |
| `int dbg` | Input debouncing gate register; 19-bit value specifies the number of 66MHz clock cycles for GATE to be stable |
| `int dbc` | Input debouncing clock register; 19-bit value specifies the number of 66MHz clock cycles for CLKIN to be stable |
| `int iie` | TRUE to invert CLKIN pin |
| `int gie` | TRUE to invert GATE pin |
| `int re` | reserved; set to 1 |
| `int end_mode` | Stop the count under one of the following conditions:<br>`DQ_EM_TBR     //tbr counts down to 0` |

**Output:**

| | |
|---|---|
| `int* cfg` | Configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `counter` number or `end_mode` is invalid, or `cfg` is NULL |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function configures an MF-101 counter for Timed Pulse Period Measurement (TPPM) mode. TPPM counts the number of rising CLKIN edges (or falling edges if `iie=1`) over the `tbr` time interval. The timebase divider is a 31-bit 66MHz counter, so the maximum `tbr` value is 0x7FFFFFFF (2147483647) corresponding to ~32.5 seconds.

When configured in TPPM mode, a call to `DqAdv101CTRead()` returns two pieces of data per channel:

1. total number of 66MHz clock cycles between the first rising edge to the last rising edge (`counts = data_read[0]`)
2. total number of rising edges (`pulses=data_read[1]`)

Since the number of periods is one less than the number of rising edges, the average period=(`counts`)/[66000000*(`pulses-`1)] seconds. At least 2 rising edges are required to measure a period. If less than 2 rising edges occur in the specified time interval, a count of 0 will be returned.

### *4.1.31 DqAdv101CTCfgForQuadrature*

**Syntax:**

```
int DqAdv101CTCfgForQuadrature(int hd, int devn, int counter,
int startmode, int tbr, int dbg, int dbc, int iie, int gie, int
end_mode, int* cfg)
```

**Command:**

Configure a counter as a quadrature encoder.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter to configure (0 or 1) |
| `int startmode` | Counter startup mode, one of the following constants:<br>`DQ_MF101_CT_SMAUTO //counter starts immediately`<br>`DQ_MF101_CT_SMSOFT //counter starts by calling DqAdv101CTStartCounter()` |
| `int tbr` | Timebase register; 31-bit value specifies the number of 66MHz clock cycles for a timed measurement |
| `int dbg` | Input debouncing gate register; 19-bit value specifies the number of 66MHz clock cycles for GATE to be stable |
| `int dbc` | Input debouncing clock register; 19-bit value specifies the number of 66MHz clock cycles for CLKIN to be stable |
| `int iie` | TRUE to invert CLKIN pin |
| `int gie` | TRUE to invert GATE pin |
| `int end_mode` | Stop the count under one of the following conditions:<br>`DQ_EM_CR0 //count reaches cr0 value (0)`<br>`DQ_EM_CR1 //count reaches cr1 value (0xFFFFFFFF)`<br>`DQ_EM_TBR //tbr counts down to 0` |
| `int lr` | Sets quadrature midpoint, usually 0x80000000 |

**Output:**

| | |
|---|---|
| `int* cfg` | Configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `counter` number or `end_mode` is invalid, or `cfg` is NULL |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function configures an MF-101 counter for quadrature encoding. The counter counts the incoming pulses on CLKIN using the GATE line as the direction: counts up if GATE=1, down if GATE=0. In this mode, `DqAdv101CTRead()` returns the current count value.

### *4.1.32 DqAdv101CTCfgForPWM*

**Syntax:**

```
int DqAdv101CTCfgForPWM (int hd, int devn, int counter, int
startmode, int sampwidth, int ps, int cr0, int cr1, int dbg, int
dbc, int iie, int gie, int oie, int extclk, int trig, int trs,
int enc, int gated, int re, int end_mode, int lr, int* cfg)
```

**Command:**

Configure a counter for PWM output.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter to configure (0 or 1) |
| `int startmode` | Counter startup mode, one of the following constants:<br>`DQ_MF101_CT_SMAUTO //counter starts immediately`<br>`DQ_MF101_CT_SMSOFT //counter starts by calling DqAdv101CTStartCounter()` |
| `int sampwidth` | Currently unused on the MF-101, but must pass in one of:<br>`DQ_LN_DATASZ8 //one byte`<br>`DQ_LN_DATASZ16 //two bytes`<br>`DQ_LN_DATASZ24 //three bytes`<br>`DQ_LN_DATASZ32 //four bytes (default)` |
| `int ps` | 32-bit prescaler divides CLKIN by this factor if `extclk=1`. If `extclk=0`, divides internal 66MHz clock. |
| `int cr0` | Compare register 0, CLKOUT=low between `lr` and `cr0` |
| `int cr1` | Compare register 1, CLKOUT=high between `cr0` and `cr1` |
| `int dbg` | Input debouncing gate register; 19-bit value specifies the number of 66MHz clock cycles for GATE to be stable |
| `int dbc` | Input debouncing clock register; 19-bit value specifies the number of 66MHz clock cycles for CLKIN to be stable |
| `int iie` | TRUE to invert CLKIN pin |
| `int gie` | TRUE to invert GATE pin |
| `int oie` | TRUE to invert CLKOUT pin |
| `int extclk` | TRUE to use external clock (CLKIN) |
| `int trig` | TRUE to enable trigger |
| `int trs` | TRUE to use GATE as trigger (only if `trig=1`) |
| `int enc` | TRUE to clear counter to `lr` after `end_mode` and await next trigger (only if `trig=1` and `re=1`); call `DqAdv101StartCounter()` to reset trigger |
| `int gated` | TRUE for GATE to enable/disable counter (only if `trig=0`) |
| `int re` | TRUE to restart counter after `end_mode` condition is met |
| `int end_mode` | Stop the count under one of the following conditions: |

```
DQ_EM_CR0    //count reaches cr0 value
DQ_EM_CR1    //count reaches cr1 value (default for PWM)
DQ_EM_FFF    //count reaches 0xFFFFFFFF
DQ_EM_GT     //GATE goes from high to low
```

| | |
|---|---|
| `int lr` | Load register; sets the initial value of the counter |

**Output:**

| | |
|---|---|
| `int* cfg` | Configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `counter` number or `end_mode` is invalid, or `cfg` is NULL |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function sets up an MF-101 counter for PWM output. Output remains low while the count value is between `lr` and `cr0`. Output is high while the count value is between `cr0` and `cr1`. A In other words, if `lr=0`, the PWM period is `cr1` and its duty cycle is given by (`cr1`-`cr0`)/`cr1`. After the counter is started, `cr0` and `cr1` may be updated on the fly using `DqAdv101CTWrite()`. `DqAdv101CTRead()` returns the current count value.

### 4.1.33 DqAdv101CTCfgForPWMTrain

**Syntax:**

```
int DqAdv101CTCfgForPWMTrain (int hd, int devn, int counter, int
startmode, int sampwidth, int ps, int n_pulses, int cr0, int
cr1, int dbg, int dbc, int iie, int gie, int oie, int extclk,
int trig, int trs, int enc, int gated, int re, int end_mode, int
lr, int* cfg)
```

**Command:**

Configure a counter to output a set number of PWM pulses.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter to configure (0 or 1) |
| `int startmode` | Counter startup mode, one of the following constants: |

```
DQ_MF101_CT_SMAUTO //counter starts immediately
DQ_MF101_CT_SMSOFT //counter starts by calling
                     DqAdv101CTStartCounter()
```

| | |
|---|---|
| `int sampwidth` | Currently unused on the MF-101, but must pass in one of: |

```
DQ_LN_DATASZ8   //one byte
DQ_LN_DATASZ16  //two bytes
```

| | |
|---|---|
| | `DQ_LN_DATASZ24  //three bytes`<br>`DQ_LN_DATASZ32  //four bytes (default)` |
| `int ps` | 32-bit prescaler divides CLKIN by this factor if `extclk=1`. If `extclk=0`, divides internal 66MHz clock. |
| `int n_pulses` | Number of pulses to generate |
| `int cr0` | Compare register 0, CLKOUT=low between `lr` and `cr0` |
| `int cr1` | Compare register 1, CLKOUT=high between `cr0` and `cr1` |
| `int dbg` | Input debouncing gate register; 19-bit value specifies the number of 66MHz clock cycles for GATE to be stable |
| `int dbc` | Input debouncing clock register; 19-bit value specifies the number of 66MHz clock cycles for CLKIN to be stable |
| `int iie` | TRUE to invert CLKIN pin |
| `int gie` | TRUE to invert GATE pin |
| `int oie` | TRUE to invert CLKOUT pin |
| `int extclk` | TRUE to use external clock (CLKIN) |
| `int trig` | TRUE to enable trigger |
| `int trs` | TRUE to use GATE as trigger (only if `trig=1`) |
| `int enc` | TRUE to clear counter to `lr` after `end_mode` and await next trigger (only if `trig=1` and `re=1`); call `DqAdv101StartCounter()` to reset trigger |
| `int gated` | TRUE for GATE to enable/disable counter (only if `trig=0`) |
| `int re` | TRUE to restart counter after `end_mode` condition is met |
| `int end_mode` | Stop the count under one of the following conditions:<br>`DQ_EM_CR0    //count reaches cr0 value`<br>`DQ_EM_CR1    //count reaches cr1 value (default for PWM)`<br>`DQ_EM_FFF    //count reaches 0xFFFFFFFF`<br>`DQ_EM_GT     //GATE goes from high to low` |
| `int lr` | Load register; sets the initial value of the counter |

**Output:**

| | |
|---|---|
| `int* cfg` | Configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `counter` number or `end_mode` is invalid, or `cfg` is NULL |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function sets up an MF-101 counter to output a set number of pulses. Output remains low while the count value is between `lr` and `cr0`. Output is high while the count value is between `cr0` and

`cr1`. The counter stops after the desired number of pulses have been generated. Call `DqAdv101StartCounter()` to re-trigger the counter.
After the counter is started, `cr0` and `cr1` may be updated on the fly using `DqAdv101CTWrite()`. `DqAdv101CTRead()` returns the current count value.

**Notes:**

- If using the internal 66MHz clock, `cr1` should be ≥ 5.
- If using an external clock, the value loaded to `cr0` should correspond to >75.75ns based on the expected CLKIN frequency.

### *4.1.34 DqAdv101SerialSetConfig*

**Syntax:**

`int DqAdv101SerialSetConfig(int hd, int devn, pMF101SERIALCFG pCfg)`

**Command:**

Configure the MF-101 serial port.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `pMF101SERIALCFG pCfg` | Pointer to `MF101SERIALCFG` configuration structure |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `term_buffer` or `term_length` exceeds 128 characters |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

Serial port settings are defined using the `MF101SERIALCFG` structure:

```
typedef struct {
    uint32 flags;                     //OR in flags to change associated parameters

    // DQ_MF101_SERIAL_CFG_BAUD flag
    uint32 baud_rate;                 // desired baud rate

    // DQ_MF101_SERIAL_CFG_CHAN flag
    uint32 mode;                      // RS-232, 422, or 485
    uint32 loopback;                  // =1 enable internal loopback
    uint32 stop_bits;                 // number of stop bits
    uint32 parity;                    // type of parity bit
    uint32 width;                     // Number of bits in each character
    uint32 error_en;                 // reserved
    uint32 break_en;                 // =1 sets serial output to logical 0
    uint32 term_fs_tx_rx;            // =1 enables RS-485 termination resistors

    // DQ_MF101_SERIAL_CFG_CHAR_DELAY flag
    float char_delay_us;             // delay between each character sent to FIFO
    uint32 char_delay_src;           // clock source for char_delay_us

    // DQ_MF101_SERIAL_CFG_FRAME_DELAY flag
    float frame_delay_us;            // delay between minor frames
    uint32 frame_delay_src;          // clock source for frame_delay_us
    uint32 frame_delay_mode;         // defines the minor frame for frame_delay_us
    uint32 frame_delay_length;       // number of characters in minor frame; only for
                                        FIXEDLEN delay mode
    float frame_delay_repeat_us;     // repeat time between major frames

    // DQ_MF101_SERIAL_CFG_TERM_STRING flag
    char* term_buf;                  // termination string
    uint32 term_length;              // length of term_buf to use

    // DQ_MF101_SERIAL_CFG_TIMEOUT flag
    uint32 timeout;                  // number of clocks without receiving data
                                        before timeout
    uint32 timeout_clock;            // units for timeout

    // DQ_MF101_SERIAL_TX_WM flag
    uint32 tx_watermark;             // reserved

    // DQ_MF101_SERIAL_RX_WM flag
    uint32 rx_watermark;             // RX FIFO watermark for data flow control

    // DQ_MF101_SERIAL_CFG_EXT flag
    uint32 suppress_hd_echo;         // =1 suppresses echo in RS-422 mode
    uint32 add_ts_on_idle;           // =1 adds timestamp to RX FIFO during idle
                                        state
} MF101SERIALCFG, *pMF101SERIALCFG;
```

`MF101SERIALCFG` parameters:

- `<flags>`

  OR in one or more of the following flags:

| | |
|---|---|
| DQ_MF101_SERIAL_CFG_CLEAR | Resets all parameters to defaults and clears FIFOs. ORing in other flags will overwrite the defaults. |
| DQ_MF101_SERIAL_CFG_BAUD | Allows baud rate to be reconfigured. |
| DQ_MF101_SERIAL_CFG_CHAN | Allows general channel settings to be reconfigured. |
| DQ_MF101_SERIAL_CFG_CHAR_DELAY | Allows character delay to be reconfigured. |
| DQ_MF101_SERIAL_CFG_FRAME_DELAY | Allows frame delay to be reconfigured. |
| DQ_MF101_SERIAL_CFG_TERM_STRING | Allows termination string to be defined. |
| DQ_MF101_SERIAL_CFG_TIMEOUT | Allows timeout to be reconfigured. |
| DQ_MF101_SERIAL_CFG_TX_WM | Allows TX watermark level to be reconfigured. |
| DQ_MF101_SERIAL_CFG_RX_WM | Allows RX watermark level to be reconfigured. |
| DQ_MF101_SERIAL_CFG_EXT | Allows extended configuration settings. |

The `MF101SERIALCFG` structure definition lists the parameters associated with each flag. For example, to change the baud rate and the number of stop bits:

```
cfg.flags = DQ_MF101_SERIAL_CFG_BAUD|DQ_MF101_SERIAL_CFG_CHAN;
```

If a flag is not set, its parameters will be ignored and replaced with default values.

- `<baud_rate>`
  Baud rate is specified in bits per second (bps), set to 57600 by default. The minimum supported value is 300 bps. RS-232 mode supports rates up to 256 kbps, while RS-422/485 mode supports rates up to 2 Mbps. The requested baud rate may differ from the actual baud rate by no more than 0.01%. To set the `baud_rate`, users must OR `DQ_MF101_SERIAL_CFG_BAUD` into `flags`.

- `<mode>`
  Select one of the following:

| | |
|---|---|
| `DQ_MF101_SERIAL_CFG_MODE_232` | RS-232 mode (default) |
| `DQ_MF101_SERIAL_CFG_MODE_485F` | RS-485 full duplex mode, compatible with RS-422 |
| `DQ_MF101_SERIAL_CFG_MODE_485H` | RS-485 half duplex mode |

  To switch the `mode`, users must OR `DQ_MF101_SERIAL_CFG_CHAN` into `flags`.

- `<loopback>`
  Loopback is disabled (=0) by default. When enabled (1), RX and TX signals are connected internally and external signals are no longer generated.
  To enable `loopback`, users must OR `DQ_MF101_SERIAL_CFG_CHAN` into `flags`.

- `<stop_bits>`
  Select one of the following:

| | |
|---|---|
| `DQ_MF101_SERIAL_CFG_STOP_1` | Include one STOP bit (default) |
| `DQ_MF101_SERIAL_CFG_STOP_1_5` | Assert STOP for duration of 1.5 bits |
| `DQ_MF101_SERIAL_CFG_STOP_2` | Include two STOP bits |

  To configure `stop_bits`, users must OR `DQ_MF101_SERIAL_CFG_CHAN` into `flags`.

- `<parity>`
  Select one of the following:

| | |
|---|---|
| `DQ_MF101_SERIAL_CFG_PARITY_NONE` | No parity bit (default) |
| `DQ_MF101_SERIAL_CFG_PARITY_EVEN` | Parity bit is 0 if data contains even number of 1’s. |
| `DQ_MF101_SERIAL_CFG_PARITY_ODD` | Parity bit is 0 if data contains odd number of 1’s. |
| `DQ_MF101_SERIAL_CFG_PARITY_SPACE` | Parity bit is set to 0 |
| `DQ_MF101_SERIAL_CFG_PARITY_MARK` | Parity bit is set to 1 |

  To configure `parity`, users must OR `DQ_MF101_SERIAL_CFG_CHAN` into `flags`.

- `<width>`
  While each character is always stored as a byte in the FIFO, users may change the number of data bits transferred with each frame. Select one of the following character widths:

| | |
|---|---|
| `DQ_MF101_SERIAL_CFG_WIDTH_8` | 8 data bits in each frame (default) |
| `DQ_MF101_SERIAL_CFG_WIDTH_7` | 7 data bits in each frame (for true ASCII) |
| `DQ_MF101_SERIAL_CFG_WIDTH_6` | 6 data bits in each frame (rarely used) |
| `DQ_MF101_SERIAL_CFG_WIDTH_5` | 5 data bits in each frame (for Baudot code) |

  To configure `width`, users must OR `DQ_MF101_SERIAL_CFG_CHAN` into `flags`.

- `<break_en>`

The TX line is held at logic low when this parameter is enabled (=1). Users must also OR `DQ_MF101_SERIAL_CFG_CHAN` into `flags` to activate `break_en`. The break is disabled (=0) by default but may be enabled momentarily using `DqAdv101SerialSendBreak()`.

- `<term_fs_tx_rx>`

  OR in one or more of the following termination resistors for RS-485 full or half-duplex networks. They are all disabled by default.

| | |
|---|---|
| DQ_MF101_SERIAL_CFG_TERM_FS | Enables fail-safe biasing. When the bus is idle, voltage dividers pull the differential TX voltage to a logic high. Auto-disables when TX FIFO is empty. |
| DQ_MF101_SERIAL_CFG_TERM_TX | Enables 91 Ω terminator between TX+/TX-. |
| DQ_MF101_SERIAL_CFG_TERM_RX | Enables 91 Ω terminator between RX+/RX-. |

  To configure `term_fs_tx_rx`, users must OR `DQ_MF101_SERIAL_CFG_CHAN` into `flags`.

- `<char_delay_us>`

  Defines the time between when each character is written to the TX FIFO (default=0). Units are microseconds if using an internal clock source. For an external source, units are in clock cycles. If enabled, the delay should be longer than the duration of the transmitted frame (start+data+parity+stop). Also, a clock source must be selected in `char_delay_src` and `DQ_MF101_SERIAL_CFG_CHAR_DELAY` OR'd into `flags`.

- `<char_delay_src>`

  Select one of the following clock sources to use for `char_delay_us`:

| | |
|---|---|
| DQ_MF101_SERIAL_CFG_DELAY_DISABLED | No character delay (default) |
| DQ_MF101_SERIAL_CFG_DELAY_INTERNAL | Internal subsystem clock |

  To select a clock source, users must OR `DQ_MF101_SERIAL_CFG_CHAR_DELAY` into `flags`.

- `<frame_delay_us>`

  Specifies the time from the start of one minor frame to the start of the next minor frame. Units are microseconds if using an internal clock source. For an external source, units are in clock cycles.

  The minor frame is defined by `frame_delay_mode` (default=0). Make sure that if the delay is enabled, it is longer than the duration of the minor frame. Also, a clock source must be selected in `frame_delay_src` and `DQ_MF101_SERIAL_CFG_FRAME_DELAY` OR'd into `flags`.

- `<frame_delay_src>`

  Select one of the following clock sources to use for `frame_delay_us`:

| | |
|---|---|
| DQ_MF101_SERIAL_CFG_DELAY_DISABLED | No delay (default) |
| DQ_MF101_SERIAL_CFG_DELAY_INTERNAL | Internal subsystem clock |

  To select a clock source, users must OR `DQ_MF101_SERIAL_CFG_FRAME_DELAY` into `flags`.

- `<frame_delay_mode>`

Select one of the following moes to define how characters are grouped into minor frames. The minor frames are separated by `frame_delay_us.`

| | |
|---|---|
| `DQ_MF101_SERIAL_CFG_FRAMEDELAY_DISABLED` | No frame delay (default) |
| `DQ_MF101_SERIAL_CFG_FRAMEDELAY_FIXEDLEN` | `frame_delay_length` number of characters |
| `DQ_MF101_SERIAL_CFG_FRAMEDELAY_VMAP_LEN` | All characters in a single `DqAdv101SerialWriteTXFIFO()`call. Can be used with a repeat flag to maintain a delay between VMap writes. |
| `DQ_MF101_SERIAL_CFG_FRAMEDELAY_ZERO_CHAR` | End of minor frame is indicated by an ASCII NUL character (0x00). Zero is transmitted when it's the last character in a write. |
| `DQ_MF101_SERIAL_CFG_FRAMEDELAY_VARLEN` | Minor frame size is defined by an extra character preceding the data characters.<br>**Example:**<br>`data[0]=3` // three characters in frame A<br>`data[1]=0xa` //frame A data<br>`data[2]=0xa` //frame A data<br>`data[4]=0xa` //frame A data<br>`data[5]=2` //two characters in frame B<br>`data[6]=0xb` //frame B data<br>`data[7]=0xb` //frame B data<br><br>Transmitted: 0xa, 0xa, 0xa, delay, 0xb, 0xb |

In addition, a repeat condition may be logical OR'd into the delay mode selected above.

| | |
|---|---|
| `DQ_MF101_SERIAL_CFG_FRAMEDELAY_REPEAT` | Output the last written group of characters every `frame_delay_repeat_us.` |

To set `frame_delay_mode`, users must OR `DQ_MF101_SERIAL_CFG_FRAME_DELAY` into `flags`.

- `<frame_delay_length>`
  Defines the number of characters in a minor frame when `frame_delay_mode = DQ_MF101_SERIAL_CFG_FRAMEDELAY_FIXEDLEN`. To set `frame_delay_length`, users must also OR `DQ_MF101_SERIAL_CFG_FRAME_DELAY` into `flags`.

- `<frame_delay_repeat_us>`
  Specifies the period for writing major frames to the TX FIFO (default=0). Units are microseconds if using an internal clock source. For an external source, units are in clock cycles.

  This parameter is only active when `frame_delay_mode` includes the `DQ_MF101_SERIAL_CFG_FRAMEDELAY_REPEAT` condition. The major frame consists of the characters last written by `DqAdv101SerialWriteTxFIFO()`. Make sure that if the delay is enabled, it is longer than the duration of the major frame. To set `frame_delay_repeat_us`, users must also OR `DQ_MF101_SERIAL_CFG_FRAME_DELAY` into `flags`.

- `<term_buf>`
  This buffer contains the termination string (128 characters maximum). `DqAdv101SerialReadRxFIFO()` stops reading after the termination string has been found. To use `term_buf`, users must OR `DQ_MF101_SERIAL_CFG_TERM_STRING` into `flags` and set `term_length`. Only the first `term_length` number of characters in the termination string are used.

- `<term_length>`
  Trims the termination string in `term_buf` to a new length. `Term_length` should be ≤ the length of the string in `term_buf`. To set `term_length`, users must OR `DQ_MF101_SERIAL_CFG_TERM_STRING` into `flags`.

- `<timeout>`
  Defines the timeout period when no data is seen on the RX line. This parameter is configured as a number of `timeout_clock` cycles. To set `timeout`, users must OR `DQ_MF101_SERIAL_CFG_TIMEOUT` into `flags`.

- `<timeout_clock>`
  Select one of the following clock sources to use for `timeout`:

| | |
|---|---|
| DQ_MF101_SERIAL_CFG_TIMEOUT_1MS | 1ms clock |
| DQ_MF101_SERIAL_CFG_TIMEOUT_100US | 100us clock |
| DQ_MF101_SERIAL_CFG_TIMEOUT_10US | 10us clock |
| DQ_MF101_SERIAL_CFG_TIMEOUT_1US | 1us clock |

  To select a clock source, users must OR `DQ_MF101_SERIAL_CFG_TIMEOUT` into `flags`.

- `<rx_watermark>`
  The RX FIFO holds 2048 characters, so `rx_watermark` ranges between 0 (default) and 2047. Use `DqAdv101SerialFlowControl() to` configure what happens when the number of characters in the RX FIFO crosses `rx_watermark`. To set `rx_watermark`, users must OR `DQ_MF101_SERIAL_RX_WM` into `flags`.

- `<suppress_hd_echo>`
  This parameter is disabled (=0) by default and only applies to RS-485 half-duplex mode. Because TX and RX lines are internally connected in RS-485 half-duplex mode, the receiver will see all data being transmitted. When set to 1, the receiver is disabled and no longer reads the “echo” of the transmitted character. To set `suppress_hd_echo`, users must OR `DQ_MF101_SERIAL_CFG_EXT` into `flags`.

- `<add_ts_on_idle>`
  Adds a timestamp to the RX FIFO after no data has been received for the `timeout` period. Once the timestamp is added to the FIFO, data needs to be received at least once before another timestamp can be added.

### *4.1.35 DqAdv101SerialClearFIFO*

**Syntax:**

```
int DqAdv101SerialClearFIFO(int hd, int devn, int action)
```

**Command:**

Clear the RX and/or TX FIFOs.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int action` | Which FIFO to clear:<br>`DQ_MF101_SERIAL_CLEAR_FIN //input FIFO (RX)`<br>`DQ_MF101_SERIAL_CLEAR_FOUT //output FIFO (TX)`<br>`DQ_MF101_SERIAL_CLEAR_FINFOUT //RX and TX FIFOs` |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function clears the input and/or output FIFOs of the MF-101 serial subsystem.

### *4.1.36 DqAdv101SerialEnable*

**Syntax:**

```
int DqAdv101SerialEnable(int hd, int devn, int enable)
```

**Command:**

Enable or disable serial port.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int enable` | Enable=1, Disable=0 |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |

| | |
|---|---|
| Other negative values | Low level IOM error |

**Description:**
This function enables or disables the MF-101 serial port. Before an enable, configure the port using `DqAdv101SerialSetConfig()` and clear the FIFOs with `DqAdv101SerialClearFIFO()`.

### *4.1.37 DqAdv101SerialReadRxFIFO*

**Syntax:**

```
int DqAdv101SerialReadRxFIFO(int hd, int devn, int requested,
uint8* data, int* returned, int* remained)
```

**Command:**

Read data from the receive FIFO.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int requested` | Number of characters to read from receive FIFO |

**Output:**

| | |
|---|---|
| `uint8* data` | Received characters |
| `int* returned` | Number of bytes returned (each character is stored as a byte) |
| `int* remained` | Bytes of unread data remaining in receive FIFO |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `requested` or `data` is NULL |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
This function reads characters from the RX FIFO. `Returned` may be less than `requested` if the RX FIFO is short on data or if the termination string has been found. If the device is configured for `DQ_MF101_SERIAL_RTS_AUTOFLOW_NORMAL_TIMING` or `DQ_MF101_SERIAL_RTS_AUTOFLOW_GATED_TIMING`, user should read data fast enough to prevent the RX FIFO from filling. If RX FIFO is filled, serial configuration will have to be reset.

**Notes:**

- Only for use with point-by-point/immediate DAQ mode
- The termination string can span across multiple function calls. If one call returns the beginning of the termination string, the next call will watch for the remainder of the string.

### *4.1.38 DqAdv101SerialReadRxFIFOEx*

**Syntax:**

`int DqAdv101SerialReadRxFIFOEx(int hd, int devn, int requested, uint16* data, int* returned, int* remained)`

**Command:**

Read data, timestamps, and status from the receive FIFO.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int requested` | Number of characters to read from receive FIFO |

**Output:**

| | |
|---|---|
| `uint16* data` | Received characters and status data |
| `int* returned` | Number of bytes returned (each character is stored as a byte) |
| `int* remained` | Bytes of unread data remaining in receive FIFO |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `requested` or `data` is NULL |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function reads data from the RX FIFO including status bits. Data format is a uint16, where the lower 8 bits contain the received character and the upper 8 bits contain the status:

```
DQ_MF101_SERIAL_DATA_TS     (1L<<15)   // Data is timestamp (add_ts_on_idle must be
                                          enabled)
DQ_MF101_SERIAL_DATA_FAE    (1L<<11)   // =1 FIFO almost empty, i.e. this data is
                                          the last remaining in the FIFO
DQ_MF101_SERIAL_DATA_PE   (1L<<10)   // =1 parity error detected with this data
DQ_MF101_SERIAL_DATA_FE     (1L<<9)    // =1 frame error detected with this data
DQ_MF101_SERIAL_DATA_BI     (1L<<8)    // =1 indicates break condition (no data)
```

`Returned` may be less than `requested` if the RX FIFO is short on data or if the termination string has been found. If the device is configured for `DQ_MF101_SERIAL_RTS_AUTOFLOW_NORMAL_TIMING` or `DQ_MF101_SERIAL_RTS_AUTOFLOW_GATED_TIMING`, user should read data fast enough to prevent the RX FIFO from filling. If RX FIFO is filled, serial configuration will have to be reset.

**Notes:**

- Only for use with point-by-point/immediate DAQ mode

- The termination string can span across multiple function calls. If one call returns the beginning of the termination string, the next call will watch for the remainder of the string.

### *4.1.39 DqAdv101SerialWriteTxFIFO*

**Syntax:**

```
int DqAdv101SerialWriteTxFIFO(int hd, int devn, uint32
requested, uint8* data, int* written, int* remained_free)
```

**Command:**

Write data to the transmit FIFO.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 requested` | Number of characters to write to TX FIFO |
| `uint8* data` | Buffer of message to send |

**Output:**

| | |
|---|---|
| `int* written` | Number of bytes written (each character stored as a byte) |
| `int* remained` | Number of bytes remaining free in the TX FIFO |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `data` or `written` is NULL |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function is only for use with Point-by-Point DAQ mode. It writes a block of data to the TX FIFO on the MF-101 serial port. If `requested == written`, all data fit into the FIFO.

**Notes:**

### *4.1.40 DqAdv101SerialSendBreak*

**Syntax:**

```
int DqAdv101SerialSendBreak(int hd, int devn, uint32
duration_ms)
```

**Command:**

Transmit a break of a specified duration.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 duration_ms` | Duration of break in milliseconds |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function is for Point-by-Point DAQ mode only. It transmits a break state (holds logic level low) for the time specified by `duration_ms`. No data is written while the break state is active. Usually `duration_ms` is set to longer than the time needed for transmitting the Start Bit + Data Bits. That way, the receiver can detect the break condition when the expected Stop Bit does not arrive.

**Notes:**

- A break is sent regardless of the `break_en` configuration in `DqAdv101SerialSetConfig()`.

### *4.1.41 DqAdv101SerialFlowControl*

**Syntax:**

```
int DqAdv101SerialFlowControl(int hd, int devn, uint32 cts_cfg,
uint32 rts_cfg, uint32* cts_state)
```

**Command:**

Configure RS-232 flow control.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 cts_cfg` | CTS configuration constant |
| `uint32 rts_cfg` | RTS configuration constant |

**Output:**

| | |
|---|---|
| `uint32* cts_state` | CTS pin state returned in LSB. (0=negative voltage, 1=positive voltage) |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `cts_cfg` or `rts_cfg` is not one of the defined constants |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |

| | |
|---|---|
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
This function configures RTS/CTS flow control and returns the state of the CTS pin.

Input constants:

- `<cts_config>`

| | |
|---|---|
| `DQ_MF101_SERIAL_CTS_NO_CHANGE` | Keep current CTS configuration |
| `DQ_MF101_SERIAL_CTS_IGNORE` | UART transmitter sends data regardless of the CTS state (default) |
| `DQ_MF101_SERIAL_CTS_AUTOFLOW` | UART transmitter checks state of CTS pin before sending a frame; positive voltage on CTS enables data flow |

- `<rts_config>`

| | |
|---|---|
| `DQ_MF101_SERIAL_RTS_NO_CHANGE` | Keep current RTS configuration |
| `DQ_MF101_SERIAL_RTS_PIN_LOW` | Set RTS pin to -5V; may be used to disable other RS232 devices |
| `DQ_MF101_SERIAL_RTS_PIN_HIGH` | Set RTS pin to +5V (default); may be used to enable other RS232 devices |
| `DQ_MF101_SERIAL_RTS_AUTOFLOW_NORMAL_TIMING` | RTS pin is de-asserted (set to -5V) when the RX FIFO reaches the `rx_watermark` level. RTS returns to the high state (+5V) once the RX FIFO empties to 0 characters. In normal timing, RTS goes low after the first data bit of the watermark frame is present on the serial input line. |
| `DQ_MF101_SERIAL_RTS_AUTOFLOW_GATED_TIMING` | RTS pin is de-asserted (set to -5V) when the RX FIFO reaches the `rx_watermark` level. RTS returns to the high state (+5V) once the RX FIFO empties to 0 characters. In gated timing, RTS goes low after the middle of the stop bit of the watermark frame. |

**Notes:**

- If the RX FIFO overflows when RTS Autoflow is enabled, the receiver stops receiving data until a hard reset is performed.

### *4.1.42 DqAdv101I2CSetConfig*

**Syntax:**

```
int DqAdv101I2CSetConfig(int hd, int devn, pMF101I2CCFG pCfg,
pMCTPARAM pMctprm)
```

**Command:**

Configure the I$^2$C master and slave.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `pMF101I2CCFG pCfg` | Pointer to `MF101I2CCFG` configuration structure |
| `pMCTPARAM pMctprm` | Pointer to `MCTPARAM` custom clock rate configuration structure, 0 if not required |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_DEVICE_BUSY` | Device is in use |
| `DQ_DATA_ERROR` | Layer returned invalid data |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

I$^2$C master and slave settings are contained in the `MF101I2CCFG` structure:

```
typedef struct {
    // General parameters
     uint32 flags;                 // select active parameters to set/change
    uint32 clock;                  // clock frequency
    uint32 tx_lines;               // reserved

    // Master configuration
    uint32 master_cfg;             // select active master settings
    uint32 master_idle_delay;      // delay before acquiring bus in MM mode, per 15ns
    uint32 master_byte_delay;      // delay between bytes sent by master (µs)
    uint32 master_max_sync_delay;  // max time that a slave can delay the clock (µs)
    uint32 master_datasz_unfifo;   // reserved
    uint32 master_to_cfg;          // max timeout before releasing bus (µs)
    uint32 master_wait_bm_fifo_ms; // reserved
    uint32 master_xdcp_device_type; // reserved

    // Slave configuration
    uint32 slave_cfg;              // select active slave settings
    uint32 slave_addr;             // set 7 or 10-bit slave address
    uint32 slave_data;             // data to send when slave TX FIFO is empty
    uint32 slave_sync_dly;         // time slave delays clock during ACK, per 15ns
    uint32 slave_ack_dly;          // time to wait for the ACK clock, per 15ns
    uint32 slave_max_ack;          // max # of RX words before issuing NACK
    uint32 slave_tx_reg_size;      // number of bytes sent in UNFIFO mode
} MF101I2CCFG, *pMF101I2CCFG;
```

All members must be of uint32 size, although some values are only defined up to 12 or 16 bits.

`MF101CFGCFG` general parameters:

- `<flags>`

  Marks which parameters should be set. If a flag is not set, its parameters will be ignored and replaced with default values. OR in one or more of the following flags:

| | |
|---|---|
| `DQ_MF101_I2C_CFG_CLEAR` | Resets all parameters to defaults and clears FIFOs. ORing in other flags will overwrite the defaults. |
| `DQ_MF101_I2C_CFG_CLOCK` | Allows clock frequency to be reconfigured. |
| `DQ_MF101_I2C_CFG_TERM_LOOP` | reserved |
| `DQ_MF101_I2C_CFG_MASTER_VALID` | Allows master settings to be reconfigured. |
| `DQ_MF101_I2C_CFG_SLAVE_VALID` | Allows slave settings to be reconfigured. |
| `DQ_MF101_I2C_CFG_SDATA_ADDR` | Allows slave address and register-based data to be reconfigured. |

- `<clock>`

  To change `clock`, `flags` must have `DQ_MF101_I2C_CFG_CLOCK` OR'd in. Choose one of the following clock rate options:

| | |
|---|---|
| `DQ_MF101_I2C_CFG_CLOCK_100k` | 100kHz |
| `DQ_MF101_I2C_CFG_CLOCK_400k` | 400 kHz (default) |
| `DQ_MF101_I2C_CFG_CLOCK_1M` | 1 MHz |
| `DQ_MF101_I2C_CFG_CLOCK_CUST` | Custom rate between 2kHz…100kHz, from `MCTPARAM`. Generate `MCTPARAM` using the helper function `Dqdv101I2CCalcCustomTiming()` |

  Note that both the slave and master modules run at the same clock rate.

`MF101CFGCFG` master configuration:

To change any of the master parameters, `flags` must have `DQ_MF101_I2C_CFG_MASTER_VALID` OR'd in.

- `<master_cfg>`

  One or more of the following settings can be OR'd into `master_cfg`.

| | |
|---|---|
| `DQ_MF101_I2C_MCFG_MMASTER` | Enables multi-master mode. |
| `DQ_MF101_I2C_MCFG_LONG_IDLE_DLY` | In multi-master mode, master waits for `master_idle_delay` before taking over the bus. `DQ_MF101_I2C_MCFG_MMASTER` must also be selected. |
| `DQ_MF101_I2C_MCFG_BYTE_DELAY` | Master inserts `master_byte_delay` between bytes (extra delay after ACK). |
| `DQ_MF101_I2C_MCFG_CLK_SYNC` | Allows clock synchronization (stretching) by slaves. Max delay is given by `master_max_sync_delay`. |
| `DQ_MF101_I2C_MCFG_RAWMODE` | Switches master into raw mode. In raw mode, the master TX FIFO is written with `DqAdv101MasterWriteTxPhyFIFO()`. |
| `DQ_MF101_I2C_MCFG_XDCP` | reserved |

  None are selected by default (=0).

- `<master_idle_delay>`

  In Multi-Master mode, the bus must be idle for `master_idle_delay` amount of time before the master can acquire the bus. This delay is a 32-bit value programmed as a number of 15.15ns clocks (default =0). To activate `master_idle_delay,` users must OR both `DQ_MF101_I2C_MCFG_MMASTER | DQ_MF101_I2C_MCFG_LONG_IDLE_DLY` into `master_cfg`.

- `<master_byte_delay>`
  Specifies the delay between bytes sent by master, i.e. the time between the falling edge of the ACK clock to the rising edge of the next clock. This delay is programmed with 1µs resolution (default =0), up to a max of 490µs. To configure `master_byte_delay`, users must OR `DQ_MF101_I2C_MCFG_BYTE_DELAY` into `master_cfg`.

- `<master_to_cfg>`
  Master releases the bus if SDA or SCL have been stuck LOW for `master_to_cfg` amount of time. This helps return the bus to a valid idle state. The timeout duration is a 16-bit value programmed with 1µs resolution (default =0).

- `<master_max_sync_delay>`
  Specifies the maximum time that any slave on the bus can stretch the clock. If a slave is still holding SCL low after `master_max_sync_delay`, the master stops sending data and returns the bus to an idle state. This delay is a 16-bit value programmed with 1µs resolution (default =500µs). In order for this timeout to occur, users must OR `DQ_MF101_I2C_MCFG_CLK_SYNC` into `master_cfg` and increase the overall master timeout such that `master_to_cfg` ≥ `master_max_sync_delay.`

`MF101CFGCFG` slave configuration:

- `<slave_cfg>`
  To change any of the `slave_cfg` parameters, `flags` must have `DQ_MF101_I2C_CFG_SLAVE_VALID` OR'd in. One or more of the following settings can be OR'd into `slave_cfg`.

| | |
|---|---|
| DQ_MF101_I2C_SCFG_ENABLE_BM | Sets slave to Bus Monitor (BM) mode. |
| DQ_MF101_I2C_SCFG_SCKSYN_AD | Allows slave to stretch the clock when evaluating the address from the master. Slave holds SCL low for `slave_sync_dly` between receiving the address and issuing an ACK. |
| DQ_MF101_I2C_SCFG_SCKSYN_TX | Allows slave to stretch the clock when sending data to the master. Slave holds SCL low for `slave_sync_dly` between receiving the master's ACK and sending the next byte. |
| DQ_MF101_I2C_SCFG_SCKSYN_RX | Allows slave to stretch the clock when processing data from the master. Slave holds SCL low for `slave_sync_dly` between receiving a byte and issuing an ACK. |
| DQ_MF101_I2C_SCFG_ACKLEN | Configures the slave to wait `slave_ack_dly` amount of time for the master to clock the ACK. |
| DQ_MF101_I2C_SCFG_ACK_BM | Allows acknowledge (ACK) generation in BM mode. |
| DQ_MF101_I2C_SCFG_10BIT | Configures the slave address as a 10-bit value (instead of the 7-bit default). |
| DQ_MF101_I2C_SCFG_MAXACK | Configures the slave to issue a NACK after receiving `slave_max_ack` number of words. |

| | |
|---|---|
| `DQ_MF101_I2C_SCFG_SLAVE_UNFIFO` | Configures the slave to send `slave_tx_reg_size` number of bytes. TX FIFO data is transmitted first, padded by `slave_data` if FIFO data is unavailable. |
| `DQ_MF101_I2C_SCFG_FIFO_SRXDO` | Each 12-bit RX word includes one of the following 4-bit bus conditions. If this bit is set, the RX FIFO will only store the 8-bits of data that follow bus condition 6, 7, or 8.<br>`0 RSV:            Reserved`<br>`1 START:          I2C Start condition`<br>`2 RESTART:        I2C Restart condition`<br>`3 STOP:           I2C Stop condition`<br>`4 Address + ACK:  I2C Address + ACK`<br>`5 Address + /ACK: I2C Address + NACK`<br>`6 Data + ACK:     I2C Data + ACK`<br>`7 Data + /ACK:    I2C Data + NACK`<br>`8 All data received + /ACK:`<br>`I2C Last data byte + NACK`<br>`9 Clock stretching error:`<br>`I2C Error occurred in relation to clock stretching` |

- `<slave_addr>`

  Sets the 7/10-bit slave address. To set this parameter, `flags` must have `DQ_MF101_I2C_CFG_SDATA_ADDR` OR'd in. To set a 10-bit addresses, users must also OR `DQ_MF101_I2C_SCFG_10BIT` into `slave_cfg`.

- `<slave_data>`

  This data is loaded into the slave's 32-bit TX register. This data is automatically sent when the slave TX FIFO is empty. Additionally, this is the data that gets sent in `DQ_MF101_I2C_SCFG_SLAVE_UNFIFO` mode. To set this parameter, `flags` must have `DQ_MF101_I2C_CFG_SDATA_ADDR` OR'd in.

- `<slave_sync_dly>`

  Slave extends the ACK time by `slave_sync_dly` when clock stretching is enabled. This delay is a 12-bit value programmed as a number of 15.15ns clocks (default =0). To activate `slave_sync_dly`, users must OR at least one of `DQ_MF101_I2C_SCFG_SCKSYN_AD`, `DQ_MF101_I2C_SCFG_SCKSYN_TX`, or `DQ_MF101_I2C_SCFG_SCKSYN_RX` into `slave_cfg`.

- `<slave_ack_dly>`

  Sets the maximum number of clocks allowed between the last falling edge of the data or address byte and the falling edge of the ACK clock. Data transmission stops if the master fails to clock the ACK in time. This delay is a 12-bit value programmed as a number of 15.15 ns clocks (default =0). To configure `slave_ack_dly`, users must OR `DQ_MF101_I2C_SCFG_ACKLEN` into `slave_cfg`.

- `<slave_max_ack>`

  Sets the slave RX max count register. The slave will issue a NACK when it has received `slave_max_ack` number of words from the master. (Use `DqAdv101I2CBuildCmdData()`

to set the TX/RX count registers on the master.) To configure `slave_max_ack`, users must OR `DQ_MF101_I2C_SCFG_MAXACK` into `slave_cfg`.

### *4.1.43 DqAdv101I2CFlush*

**Syntax:**

`int DqAdv101I2CFlush (int hd, int devn, int flags)`

**Command:**

Clear the I$^2$C port FIFO(s).

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int flags` | Bitmask with one or more of the following OR'd in:<br>`DQL_IOCTL101_FLUSH_SRX_I2C // flush slave RX FIFO`<br>`DQL_IOCTL101_FLUSH_STX_I2C // flush slave TX FIFO`<br>`DQL_IOCTL101_FLUSH_MRX_I2C // flush master RX FIFO`<br>`DQL_IOCTL101_FLUSH_MTX_I2C // flush master TX FIFO`<br>`DQL_IOCTL101_FLUSH_MRST_I2C // reset master module`<br>`DQL_IOCTL101_FLUSH_SRST_I2C // reset slave module` |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function flushes receive and/or transmit FIFOs on the slave and/or master depending on the selected `flags`. Flushing the FIFOs allows new data transfers to start from a known empty state.

### *4.1.44 DqAdv101I2CGetStatus*

**Syntax:**

`int DqAdv101I2CGetStatus(int hd, int devn, int flags, pMF101I2CSTS pSts)`

**Command:**

Return the status of I$^2$C subsystem.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int flags` | Either NULL or set to:<br>`DQL_IOCTL101_STATUS_CLR //clears hw_stat bits 31…16` |

**Output:**

| | |
|---|---|
| `pMF101I2CSTS pSts` | User-allocated `MF101I2CSTS` structure which stores the status information |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function requests the current status of the I2C interface and stores the information in a MF101I2CSTS structure:

```
typedef struct {    // all members must be of uint32 size
    uint32 hw_stat;     // hardware status
    uint32 pin_stat;    // pin status
    uint32 drv_stat;    // reserved
  } MF101I2CSTS, *pMF101I2CSTS;
```

Bitmapping for MF101I2CSTS members:

`<hw_stat>`

Bits 31..16 are "sticky" and will accumulate unless cleared by the `DQL_IOCTL101_STATUS_CLR` flag. Bits 15..0 return the current state.

```
DQ_MF101_I2C_STS_HW_ACKERR  (1L<<30)   // =1 master ack error has occurred
DQ_MF101_I2C_STS_HW_PIERR   (1L<<29)   // =1 PHY idle wait timeout error
DQ_MF101_I2C_STS_HW_PTOERR  (1L<<28)   // =1 PHY command wait while bus is active
                                                timeout error
DQ_MF101_I2C_STS_HW_CSERR   (1L<<27)   // =1 slave clock synchronization error
DQ_MF101_I2C_STS_HW_MDONE   (1L<<26)   // =1 master completes execution of the last
                                                command
DQ_MF101_I2C_STS_HW_MSCH    (1L<<25)   // =1 master command or data sequence was
                                                received
DQ_MF101_I2C_STS_HW_MCRCERR (1L<<24)   // reserved
---------------------------------------------------------------------------------
DQ_MF101_I2C_STS_HW_MCLOERR (1L<<23)   // =1 error in CLO-->master sequence
DQ_MF101_I2C_STS_HW_MMERR   (1L<<22)   // =1 multi-master arbitration was lost
DQ_MF101_I2C_STS_HW_STXDONE (1L<<21)   // =1 slave TX data sent (non-continuous
                                                mode)
DQ_MF101_I2C_STS_HW_SRXDTR  (1L<<20)   // =1 slave RX data ready (non-continuous
                                                mode)
DQ_MF101_I2C_STS_HW_SRXFFS  (1L<<19)   // =1 slave RX FIFO was full
DQ_MF101_I2C_STS_HW_STXFES  (1L<<18)   // =1 slave TX FIFO was empty
DQ_MF101_I2C_STS_HW_RXFFS   (1L<<17)   // =1 master RX FIFO was full
DQ_MF101_I2C_STS_HW_TXFES   (1L<<16)   // =1 master TX FIFO was empty
---------------------------------------------------------------------------------
     DQ_MF101_I2C_STS_HW_BBSY    (1L<<15)   // =1 when bus is busy
     DQ_MF101_I2C_STS_HW_CSWAIT  (1L<<14)   // =1 indicates that master waits for the
```

```
                                               slave to release SCL
DQ_MF101_I2C_STS_HW_MCLORDY (1L<<13)   // =1 when channel can accept writes
DQ_MF101_I2C_STS_HW_MWAIT   (1L<<12)   // =1 master waiting for command/data release
DQ_MF101_I2C_STS_HW_CLOMWT  (1L<<11)   // =1 when output port list SM waits for the
                                               master to become available
DQ_MF101_I2C_STS_HW_SRXFHF  (1L<<10)   // =1 slave RX FIFO is above watermark
DQ_MF101_I2C_STS_HW_SRXFF   (1L<<9)    // =1 slave RX FIFO is full
DQ_MF101_I2C_STS_HW_STXFHF  (1L<<8)    // =1 slave TX FIFO is below watermark
----------------------------------------------------------------------------------
DQ_MF101_I2C_STS_HW_STXFE   (1L<<7)    // =1 slave TX FIFO is empty
DQ_MF101_I2C_STS_HW_SBSY    (1L<<6)    // =1 when slave state machine is busy
DQ_MF101_I2C_STS_HW_EMAS    (1L<<5)    // =1 when external master owns the bus
DQ_MF101_I2C_STS_HW_MBSY    (1L<<4)    // =1 when master state machine is busy
DQ_MF101_I2C_STS_HW_RXFHF   (1L<<3)    // =1 master RX FIFO is above watermark
DQ_MF101_I2C_STS_HW_RXFF    (1L<<2)    // =1 master RX FIFO is full
DQ_MF101_I2C_STS_HW_TXFHF   (1L<<1)    // =1 master TX FIFO is below watermark
DQ_MF101_I2C_STS_HW_TXFE    (1L<<0)    // =1 master TX FIFO is empty
```

`<pin_stat>`

Returns the current state of the SDA and SCL pins.

```
DQ_MF101_I2C_STS_PIN_SDAM  (1L<<3)    // Master SDA current value
DQ_MF101_I2C_STS_PIN_SCLM  (1L<<2)    // Master SCL current value
DQ_MF101_I2C_STS_PIN_SDAS  (1L<<1)    // Slave SDA current value
DQ_MF101_I2C_STS_PIN_SCLS  (1L<<0)    // Slave SCL current value
```

### *4.1.45 DqAdv101I2CEnable*

**Syntax:**

`int DqAdv101I2CEnable(int hd, int devn, int enable, int flags)`

**Command:**

Enable or disable the I$^2$C port.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int enable` | 0=Disable, 1=Enable |
| `int flags` | reserved |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_DEVICE_BUSY` | Device is in use |
| `DQ_DATA_ERROR` | Layer returned invalid data |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
This function enables or disables the MF-101 I$^2$C port. Users should configure the port using `DqAdv101I2CSetConfig()` prior to an enable.

### *4.1.46 DqAdv101I2CBuildCmdData*

**Syntax:**

```
int DqAdv101I2CBuildCmdData(int hd, int devn, uint32 flags,
uint32 command, int wrrd_rx_size, int size, char* data, int*
tx_size, uint32* tx_data)
```

**Command:**

Build master command.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 flags` | reserved |
| `uint32 command` | I$^2$C command OR'd with slave address |
| `int wrrd_rx_size` | Number of `data` bytes to read in WRRD command (max 255) |
| `int size` | Number of `data` bytes to send for WRITE and WRRD command or to receive for READ command (max 255) |
| `char* data` | Data to send in a WRITE or WRRD command |

**Output:**

| | |
|---|---|
| `int* tx_size` | Number of uint32 words in `tx_data` |
| `uint32* tx_data` | User-allocated array to store built command words (should be at least 4+`size`/3) |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_BAD_PARAMETER` | `size` is greater than 255 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
This function builds the I$^2$C master command. The resulting command words are stored in `tx_data` and the number of words returned in `tx_size`. The command can then be written to the master TX FIFO by calling `DqAdv101I2CMasterWriteTxFIFO()`.

The `command` parameter consists of an $I^2C$ command OR'd with optional behavior bit(s) and Or'd with the slave address:

`command =` $I^2C$_command | $I^2C$_behavior_bit | $I^2C$_address

- $I^2C$_command

| | |
|---|---|
| `DQ_MF101_I2C_CMD_TDELAY` | Insert a time delay (NOP command) |
| `DQ_MF101_I2C_CMD_STOP` | STOP - issue a stop condition once bus is available |
| `DQ_MF101_I2C_CMD_ST_WRITE` | START+WRITE, including write multiple |
| `DQ_MF101_I2C_CMD_ST_READ` | START+READ, including read multiple |
| `DQ_MF101_I2C_CMD_ST_WRRD` | START+WRITE+RESTART+READ, including read multiple |
| `DQ_MF101_I2C_CMD_XDCP_READ` | START+WRITE+READ<br>(for Renesas XDCP protocol - ex. X9119) |
| `DQ_MF101_I2C_CMD_XDCP_WRITE` | START+WRITE+WRITE<br>(for Renesas XDCP protocol - ex. X9259) |

- $I^2C$_behavior_bit

  OR in these bits to modify the default behavior of the master command:

| | |
|---|---|
| `DQ_MF101_I2C_CMD_A_BIT` | use 10-bit address (0 == 7-bit address) |
| `DQ_MF101_I2C_CMD_N_BIT` | ignore NACK received after slave address (violates $I^2C$ specification) |
| `DQ_MF101_I2C_CMD_P_BIT` | do not issue STOP; next valid command with START must follow (ignored for `DQ_MF101_I2C_CMD_STOP` command) |

- $I^2C$_address

  OR in one of the following to set the $I^2C$ address for the master→slave command:

| | |
|---|---|
| `DQ_MF101_I2C_CMD_ADDR7(N)` | replace N with a 7-bit slave address |
| `DQ_MF101_I2C_CMD_ADDR10(N)` | replace N with a 10-bit slave address |

**Notes:**

- User should allocate enough space in `tx_data` to hold the command: at least 4+`size`/3 for WRRD, 2+`size`/3 for WRITE, or 2 for READ. The command information is stored in the first four uint32 words for WRRD or the first two uint32 words for WRITE and READ. Each subsequent word packs in 3 data bytes for transmission. For example, `tx_size`=3 when building a WRITE command with `size`= 1, 2, or 3 data bytes.

### 4.1.47 DqAdv101I2CCalcCustomTiming

**Syntax:**

```
int DqAdv101I2CCalcCustomTiming(int hd, float divider, pMCTPARAM
pMctprm)
```

**Command:**

Calculate timing parameters for custom clock rate.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `float divider` | Divider of 100kHz base clock (from 1…50, corresponding to rates from 100 kHz to 2 kHz) |

**Output:**

| | |
|---|---|
| `pMCTPARAM pMctprm` | User-allocated `MCTPARAM` structure which stores the custom timing parameters |

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER_1` | `divider` is out of range |
| `DQ_SUCCESS` | Successful completion |

**Description:**

This function calculates timing parameters for the requested clock rate and stores them in the `MCTPARAM` structure.

```
typedef struct {
    uint32 mctidle;     // time interval before IDLE is detected
    uint32 mctstart;    // SDA low to SCL low for START condition
    uint32 mctstop;     // SCL high to SDA high for STOP condition
    uint32 mctdbnc;     // debounce time
    uint32 mctsudat;    // data setup time
    uint32 mcthigh;     // SCL high time
} MCTPARAM, *pMCTPARAM;
```

This structure can be passed as a parameter into `DqAdv101I2CSetConfig()`. To make use of the custom timing parameters, ensure that the `MF101CFGCFG` structure has its `DQ_MF101_I2C_CFG_CLOCK` flag enabled and `clock` set to `DQ_MF101_I2C_CFG_CLOCK_CUST`.

### 4.1.48 DqAdv101I2CMasterReadRxFIFO

**Syntax:**

```
int DqAdv101I2CMasterReadRxFIFO(int hd, int devn, uint32 flags,
int size, uint16* data, int* received, int* available)
```

**Command:**

Read data from the master RX FIFO.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 flags` | Reserved |
| `int size` | Number of words requested to read (max 255) |

**Output:**

| | |
|---|---|
| `uint16* data` | Array of data read from the master RX FIFO |
| `int* received` | Number of words successfully read from the FIFO |
| `int* available` | Number of unread words in the FIFO at the moment of return |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |

| | |
|---|---|
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
This function requests `size` number of words from the master RX FIFO and returns the number of words read and still available in the FIFO. This FIFO contains data transmitted by the slave upon a READ command.

**Notes:**
Each word in the master RX FIFO contains a byte of data (bits [0:7]) plus a “STOP” bit (bit [8]). The “STOP” bit is set for the last word retrieved by a READ command.

Example of received `data`:

data[0]=0x0dd
data[1]=0x0dd
data[2]=0x0dd
data[3]=0x1dd //end of 4-byte READ command
data[4]=0x0dd
data[5]=0x0dd
data[6]=0x1dd //end of 3-byte READ command

In this example, “dd” is the byte sent by the slave and a 1 on the 9th bit indicates the end of a READ.

### *4.1.49 DqAdv101I2CMasterWriteTxFIFO*

**Syntax:**

```
int DqAdv101I2CMasterWriteTxFIFO(int hd, int devn, uint32 flags,
int size, uint32* data, int* transmitted, int* available)
```

**Command:**
Write I$^2$C commands to the master TX FIFO.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 flags` | reserved |
| `int size` | Number of uint32 words in `data` to write (max 255) |
| `uint32* data` | Array of “built” command words to write |

**Output:**

| | |
|---|---|
| `int* transmitted` | Number of “built” command words successfully written into master TX FIFO |
| `int* available` | Space still available in the FIFO at the moment of return |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
This function writes master commands into the master TX FIFO, for transmission to the slave. It writes `size` number of words from `data` and returns the number of words written and still available in the FIFO. Before calling this function, use `DqAdv101I2CBuildCmdData()`to construct the `data` array.

**Notes:**

- The board logic allows each function call to write at most 255 words. This function may be called multiple times in order to write up to 1024 words into the master TX FIFO.
- The `transmitted` output parameter refers to the number of uint32 words built by `DqAdv101I2CBuildCmdData()`, not the number of data bytes written to the slave.

### 4.1.50 DqAdv101I2CMasterWriteTxPhyFIFO

**Syntax:**

```
int DqAdv101I2CMasterWriteTxPhyFIFO(int hd, int devn, uint32
flags, int size, uint16* data, int* transmitted, int* available)
```

**Command:**
Write low-level instructions to the master TX FIFO.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 flags` | Reserved |
| `int size` | Number of uint16 words in `data` to write (max 255) |
| `uint16* data` | Array of low-level instructions |

**Output:**

| | |
|---|---|
| `int* transmitted` | Number of words successfully written into master TX FIFO |
| `int* available` | Space still available in the FIFO at the moment of return |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |

| | |
|---|---|
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
This function provides low-level access to the master TX FIFO, also known as “raw mode”. It writes `size` number of words from `data` and returns the number of words written and still available in the FIFO. The `data` array should be pre-loaded with a sequence of 12-bit words, where the upper bits[11:8] contain the raw mode command and bits[7:0] contain the data if applicable. Construct words using the following macros:

```
I2C_MRAW_PHY_TX1(B)      // Set SDA to 1 for one clock (use B=0)
I2C_MRAW_PHY_TX0(B)      // Set SDA to 0 for one clock (use B=0)
I2C_MRAW_PHY_RX(B)       // Set SDA to 1 for one clock, return SDA at the falling
                           edge of the clock (use B=0)
I2C_MRAW_PHY_START(B)    // START condition on the bus (use B=0)
I2C_MRAW_PHY_STOP(B)     // STOP condition on the bus (use B=0)
I2C_MRAW_PHY_RELEASE(B)  // Release bus without creating START or STOP conditions
                            (use B=0)
I2C_MRAW_DLY_2NUS(B)     // Delay for 2^B uS (B=0…31)
I2C_MRAW_DLY_NX8US(B)    // Delay for B*8 uS (B=0…255)
I2C_MRAW_BYTE_SEND(B)    // Transmit data byte B to the bus (B=0…255)
I2C_MRAW_BYTE_RECEIVE(B) // Read byte of data from the bus and save to master RX
                            FIFO (use B=0)
I2C_MRAW_ACK_WAIT(B)     // Wait for the ACK; NACK ends sequence (use B=0)
I2C_MRAW_SEQ_END(B)      // Ends sequence; bus is idle until this command
                            initiates the write (use B=0)
```

**Notes:**

- In the majority of these macros, parameter B is not used and should be set to zero.
- Before calling this function, configure the port with `DQ_MF101_I2C_MCFG_RAWMODE` enabled in the `master_cfg` field.

### 4.1.51 DqAdv101I2CSlaveReadRxFIFO

**Syntax:**

```
int DqAdv101I2CSlaveReadRxFIFO(int hd, int devn, uint32 flags,
int size, uint16* data, int* received, int* available)
```

**Command:**
Read data from the slave RX FIFO.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 flags` | Reserved |
| `int size` | Number of uint16 words requested to read (max 255) |

**Output:**

| | |
|---|---|
| `uint16* data` | Array of data read from the slave RX FIFO |
| `int* received` | Number of words successfully read from the FIFO |
| `int* available` | Number of unread words in the FIFO at the moment of |

return

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
This function requests `size` number of words from the slave RX FIFO and returns the number of words read and still available in the FIFO. This FIFO contains data bytes and command bytes transmitted by the master, as well as bus condition information.

**Notes:**
By default, each word in the slave RX FIFO contains 12-bits: a byte of data transmitted by the master (bits [0:7]) and a 4-bit bus condition (bits [8:11]).

| **Bus condition** | **Word Content** |
|---|---|
| 1 | START |
| 2 | ReSTART |
| 3 | STOP |
| 4 | Address + ACK |
| 5 | Address + NACK |
| 6 | Data + ACK |
| 7 | Data + NACK |
| 8 | Last data byte + NACK |
| 9 | Clock stretching error |

<u>Example of received `data` during WRITE:</u>

data[0]=0x100 // start command
data[1]=0x4aa // "aa" is the 7-bit slave address + WRITE bit
data[2]=0x6dd // data byte "dd"
data[3]=0x8dd // last data byte "dd"
data[4]=0x300 // stop command

<u>Example of received `data` during READ:</u>

data[0]=0x100 // start command
data[1]=0x4aa // "aa" is the 7-bit slave address + WRITE bit
data[2]=0x700 // slave transmitted data but stores the NACK sent by the master
data[3]=0x300 // stop command

To suppress bus conditions and only store data bytes in the FIFO, configure the port with `DQ_MF101_I2C_SCFG_FIFO_SRXDO` enabled in the `slave_cfg` field.

### *4.1.52 DqAdv101I2CSlaveWriteTxFIFO*

**Syntax:**

```
int DqAdv101I2CSlaveWriteTxFIFO(int hd, int devn, uint32 flags,
int size, uint16* data, int* transmitted, int* available)
```

**Command:**

Write data to the slave TX FIFO.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 flags` | Reserved |
| `int size` | Number of uint16 words in `data` to write (max 255) |
| `uint16* data` | Array of uint16 words to write. Data is in the 8 LSB. Upper bits are reserved for future use. |

**Output:**

| | |
|---|---|
| `int* transmitted` | Number of words successfully written into slave TX FIFO |
| `int* available` | Space still available in the FIFO at the moment of return |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function writes `size` number of words from `data` into the slave TX FIFO. It returns the number of words written and still available in the FIFO. Data is transmitted from this FIFO when the slave receives a READ command from the master.

**Notes:**

- When this FIFO becomes empty, data will be taken from the slave TX register. Write data to the register by filling the `slave_data` field in the `MF101CFGCFG` configuration structure and calling `DqAdv101I2CSetConfig()`.
- The board logic allows each function call to write at most 255 words. This function may be called multiple times in order to write up to 512 words into the slave TX FIFO.

### *4.1.53 DqAdv101ConfigEvents*

**Syntax:**

```
int DqAdv101ConfigEvents(int hd, int devn, int flags, event101_t
event, uint32* param)
```

**Command:**

Configure asynchronous events.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int flags` | One or more of the following behavior modifications:<br>`DQEVENT_NOREPLY //do not reply to this event`<br>`DQEVENT_NOSTORE //do not store UDP port to generate events` |
| `event101_t event` | Event type to monitor |
| `uint32* param` | Up to 2 additional parameters depending on event type |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not an MF-101 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function configures the firmware to send event notification packets according to the selected `event` criteria.

`<event>`

- `EV101_CLEAR`
  Clears all events that have been previously set along with all runtime variables.

- `EV101_DI_CHANGE`
  Returns an event packet whenever a configured digital input line changes state. This event type accepts two uint32 parameters:
  1. `param[0]`: =1 resets the `EV101_PERIODIC` event timer when an `EV101_DI_CHANGE` event occurs; =0 disables the reset.
  2. `param[1]:` This 16-bit mask determines which DIO pins trigger an `EV101_DI_CHANGE` event. The event triggers on both positive and negative edges.

- `EV101_PERIODIC`
  Continuously returns an event packet at the specified rate. This event type accepts two uint32 parameters:

1. `param[0]`: Divides 66MHz clock to set the periodic event rate. For example, a value of 66,000,000 configures the firmware to return event data every second. UEI recommends rates periods >100 ms.
2. `param[1]:` =1 fills event packet with all status data (see `EV101_ID` structure below); =0 leaves the `data` field empty.

`<flags>`

- `DQEVENT_NOREPLY`

  `DqAdv101ConfigEvents()` does not wait for a confirmation reply from the firmware. This comes in handy when the user wants to change event parameters (such as the change-of-state line mask) on the fly. However, disabling replies decreases reliability and is therefore only recommended for high traffic applications.

- `DQEVENT_NOSTORE`

  This flag is designed for multi-threaded applications. It programs the firmware to reply on the same UDP port as previous `DqAdv101ConfigEvents()` function calls. In the example below, the second `DqAdv101ConfigEvents()` call ignores the “a_hd2” parameter. Both `EV101_DI_CHANGE` and `EV101_PERIODIC` events will be returned on port 6344.

```
DqAddIOMPort(hd, a_hd1, 6344, timeout); //create handle to port 6344
DqAddIOMPort(hd, a_hd2, 6345, timeout); //create handle to port 6345
DqAdv101ConfigEvents(a_hd1, DEVN, 0, EV101_DI_CHANGE, ev_params);
DqAdv101ConfigEvents(a_hd2, DEVN, DQEVENT_NOSTORE, EV101_PERIODIC, ev_params);
```

Initially all events should be cleared with the following call:

```
ret = DqAdv101ConfigEvents(a_handle, devn, 0, EV101_CLEAR, 0);
```

Enable asynchronous events using `DqRtAsyncEnableEvents()` and retrieve event packets using `DqCmdReceiveEvent()`. When an event happens, the firmware sends back a `DQEVENT` packet:

```
typedef struct {
    uint8   dev;       // device
    uint8   ss;        // subsystem
    uint16  size;      // size of DQEVENT structure
    uint32  event;     // event type, e.g. EV101_DI_CHANGE
    uint8   data[];    // EV101_ID data structure
} DQEVENT, *pDQEVENT;
```

The `EV101_ID` data structure is as follows:

```
typedef struct {
    uint32 chan;        // reserved
    uint32 evtype;      // event  type, e.g. EV101_DI_CHANGE
    uint32 tstamp;      // timestamp of event
    uint32 size;        // size of the following data in bytes
    uint32 data[];      // event content, if applicable
} EV101_ID, *pEV101_ID;
```

`EV101_ID tstamp:`

- For `EV101_DI_CHANGE`, `tstamp` represents the time when the change of state occurred on the DI pin.

- For `EV101_PERIODIC`, `tstamp` represents the time when the packet was sent.

`EV101_ID data[]:` flexible structure containing up to four status words

- `data[0]`: Current debounced state of all 16 DIO lines.
- `data[1]`: Low-to-high mask; bit is set to 1 when a rising edge event has occurred on the corresponding pin.
- `data[2]`: High-to-low mask; bit is set to 1 when a falling edge event has occurred on the corresponding pin.
- `data[3]`: Change-of-state mask; bit is set to 1 when either a rising or falling edge event has occurred on the corresponding pin.

`DqCmdReceiveEvent()` automatically converts `DQEVENT` fields from network data format (big endian) to the host computer's format (little endian on Intel-based processors). However, because `DqCmdReceiveEvent()` is a generic function for all board types, it does not automatically convert the fields within `EV101_ID`. You can use the `DqHtonl()` macro to convert from big endian to little endian. For example:

```
//convert timestamp from big endian to little endian representation
uint32 timestamp = DqHtonl(hd, pEv101->tstamp);
```

**Notes:**

- If configuring for digital input events, you should also configure the digital input ADCs using `DqAdv101DISetLevels().`
- If you need to verify the sequence of received events, you can use `DqCmdReceiveEventPkt()` instead of the `DqCmdReceiveEvent()` API to retrieve the packet ID counter (`dqCounter`).

## 4.2 DNA-AI-201/202 layers

### 4.2.1 DqAdv201Read

**Syntax:**

```
int DqAdv201Read(int hd, int devn, int CLSize, uint32 *cl,
uint16 *bData, double *fData)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | number of channels |
| `uint32 *cl` | pointer to channel list, with optional timestamp request |
| `uint32 *bData` | pointer to store calibrated binary data |
| `double *fData` | pointer to store calibrated voltage data (`NULL` if not required) |

**Output:**

| | |
|---|---|
| `uint32 *cl` | channel list, channel number and gain code |
| `uint16 *bData` | calibrated binary ADC data |
| `double *fData` | calibrated voltage data |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-201 or AI-202 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, `bData` is NULL, or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function works using underlying `DqReadAIChannel()` but converts data using internal knowledge of input range and gain of every channel. It uses the `DQCMD_IOCTL` command with the `DQIOCTL_CVTCHNL` function under the hood.

The gain codes are located in powerdna.h, search key DQ_AI201_GAIN_1 .

When this function is called for the first time, the firmware stops any ongoing operation on the device specified and reprograms it in accordance with the channel list supplied. This function uses the preprogrammed CL update frequency – 10Hz. One can reprogram the update frequency by calling the `DqCmdSetClk()` command after the first call to `DqAdv201Read()`.

Thus, the user cannot call this function when the layer is involved in any streaming or data mapping operations.

If the user specifies a short timeout delay, this function can time out when called for the first time because it is executed as a pending command and layer programming takes up to 10ms.

Once this function is called, the layer continuously acquires data and every next call function returns the latest acquired data.

If one would like to cancel ongoing sampling, call the same function with 0xffffffff as the first channel number.

**Note:**

None

## 4.3 DNA-AI-204 layer

### 4.3.1 DqAdv204Read

**Syntax:**

```
int DqAdv204Read(int hd, int devn, int CLSize, uint32 *cl,
uint32 *fstatus, uint32 *bData, double *fData)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | number of channels in channel list |
| `uint32 *cl` | pointer to channel list with optional timestamp request |
| `uint32 *fstatus` | pointer to diagnostic data in the form of switch fault status. Requires array of CLSize uint32's (`NULL` if not required) |
| `uint32 *bData` | pointer to store calibrated binary data (`NULL` if not required) |
| `double *fData` | pointer to store calibrated current data (`NULL` if not required) |

**Output:**

| | |
|---|---|
| `uint32 *fstatus` | diagnostic data in the form of switch fault status. |
| `uint32 *bData` | raw binary data received from device (`NULL` if not required) |
| `double *fData` | current data expressed in milliamps or voltage data expressed in Volts (`NULL` if not required) |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-204 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE` or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

When this function is called for the first time, the firmware stops any ongoing operation on the device specified and reprograms it according to the channel list supplied. The user cannot perform this function call when the layer is involved in any streaming or data mapping operations.

If the user specifies a short timeout delay, this function can time out when called for the first time because it is executed as a pending command and layer programming takes up to 10ms.

Once this function is called, the layer continuously acquires data and every call to the function returns the latest available data. Bit 31 of bData will be 1 if bData has been updated since last read.

Available gain settings are:

| | | |
|---|---|---|
| #define DQ_AI204_GAIN_1 | (0) | +-25.0mA |
| #define DQ_AI204_GAIN_2 | (1) | +-12.5mA |
| #define DQ_AI204_GAIN_5 | (2) | +- 5.0mA |
| #define DQ_AI204_GAIN_10 | (3) | +- 2.5mA |

Gain settings (current ranges) are combined with the channel number and placed in the channel list. See macros DQ_LNCL_GAIN() and DQ_LNCL_CHANGAIN() in powerdna.h.

If one would like to cancel ongoing sampling, call this function with `0xffffffff` as the first channel number in the channel list.

**Note:**

### 4.3.2 DqAdv204CBStatus

**Syntax:**

```
int DqAdv204CBStatus(int hd, int devn, int reengage, int*
status)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int reengage` | 24-bit mask to reset tripped breakers, 1 = do reset |

**Output:**

| | |
|---|---|
| `int* status` | 24-bit mask to show CB tripped status, 1 = tripped (NULL if not required) |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-204 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Shows breaker status and/or resets tripped breakers.

**Note:**

### 4.3.3 DqAdv204EnableCB

**Syntax:**

```
int DqAdv204EnableCB(int hd, int devn, int chmask, int onoff,
int * state)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ch_mask` | bitwise mask to determine with channels are to have their circuit breaker ON/OFF status changed |
| `int onoff` | circuit breaker ON or OFF, use defined constants<br>DQ_AI204_ENABLE_CB_OFF or<br>DQ_AI204_ENABLE_CB_ON or<br>DQ_AI204_ENABLE_CB_TRIP |

**Output:**

| | |
|---|---|
| `int* state` | returned bitwise mask to indicate which channels have their CB enabled , NULL if not required |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-204 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

`int onoff` sets the CB (circuit breaker) function ON or OFF for the channels selected by the `ch_mask` parameter (bit0 selects ch0, bit1 selects ch1, etc.). The maximum current level before the CB trips is 1LSB less than the maximum current for the current range (gain setting) presently selected by the channel list. A channel has to remain in the overcurrent state for 100ms for a break (trip) to occur .

Use one of the following #defined constants for `onoff`

DQ_AI204_ENABLE_CB_OFF - always connected, circuit breaker never breaks
DQ_AI204_ENABLE_CB_ON - turns circuit breaking action ON, overcurrent breaks
DQ_AI204_ENABLE_CB_TRIP - test mode, will force the selected channels to enter the break (tripped) state after 100ms.

`Int* state` may be used to retrieve a 24-bit bitwise mask to indicate the onoff state of the breakers. A '1' will appear in the corresponding bit position if the circuit breaking action is turned ON or if it is being forced into the tripped state with DQ_AI204_ENABLE_CB_TRIP.

**Note:**

### *4.3.4 DqAdv204SetAutozero*

**Syntax:**

`int DqAdv204SetAutozero(int hd, int devn, int* onoff)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int* onoff` | autozero on or off, 0=off, 1=on |

**Output:**

| | |
|---|---|
| `int* onoff` | previous onoff state |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |

| | |
|---|---|
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-204 |
| `DQ_BAD_PARAMETER` | `onoff` pointer is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets the Autozero function in point-by-point mode to be either ON or OFF.
By default the autozeroing is ON. It is normally never turned OFF.

**Note:**

### *4.3.5 DqAdv204GetAutoZero*

**Syntax:**

```
int DqAdv204SetAutozero(int hd, int devn)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer/Board inside the IOM |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-204 |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads autozero value of channels and saves them in DAQLib for later use.
This function should be called in RTDMAP and RTVMAP mode prior to starting acquisition.

**Note:**

### *4.3.6 DqAdv204SetMovAvg*

**Syntax:**

```
int DqAdv204SetMovAvg(int hd, int devn, int onoff)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int onoff` | set moving average ON or OFF, use either #defined constant DQ_AI204_MOV_AVG_OFF or DQ_AI204_MOV_AVG_ON |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-204 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

`int onoff` sets the moving average ON or OFF .

This function enables or disables data smoothing for the data that is acquired by using the DqAdv204Read() function.

**Note:**

## *4.4 DNA-AI-205 layer*

### *4.4.1 DqAdv205Read*

**Syntax:**

```
int DqAdv205Read(int hd, int devn, int CLSize, uint32 *cl,
uint32 *bData, double *fData)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | number of channels |
| `uint32 *cl` | pointer to channel list |
| `uint32 *bData` | pointer to store calibrated binary data |
| `double *fData` | pointer to store calibrated voltage data (`NULL` if not required) |

**Output:**

| | |
|---|---|
| `uint32 *cl` | channel list |
| `uint32 *bData` | calibrated binary data |
| `double *fData` | calibrated voltage data |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-205 |

| | |
|---|---|
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, `bData` is NULL, or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function works using the underlying `DqReadAIChannel()`, but converts data using internal knowledge of input range and gain of every channel. It uses the `DQCMD_IOCTL` command with the `DQIOCTL_CVTCHNL` function under the hood.

When this function is called for the first time, the firmware stops any ongoing operation on the device specified and reprograms it in accordance with the channel list supplied. The function uses the preprogrammed CL update frequency – 10Hz. One can reprogram the update frequency by calling `DqCmdSetClk()`after the first call to `DqAdv205Read()`.

Thus, the user cannot perform this function call when the layer is involved in any streaming or data mapping operations.

If the user specifies a short timeout delay, this function can time out when called for the first time because it is executed as a pending command and layer programming takes up to 10ms.

Once this function is called, the layer continuously acquires data and every call to the function returns the latest acquired data.

If one would like to cancel ongoing sampling, call the same function with 0xffffffff as a channel number.

**Note:**

None

### *4.4.2 DqAdv205LoadCoeff*

**Syntax:**

```
int DqAdv205LoadCoeff(int hd, int devn, int fir, int channel,
int decrat, int tapsize, double *data)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int fir` | fir stage number |
| `int channel` | channel number |
| `int decrat` | decimation ratio (1 = no decimation) |
| `int tapsize` | number of taps in the filter (0 = pass-thru mode) |
| `double *data` | filter taps data |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | `fir`, `channel`, `decrat`, or `tapsize` is negative or exceeds maximum possible value |
| `DQ_NO_MEMORY` | error allocating memory |

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-205 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function loads the coefficient table.

**Note:**

Filter taps should be converted into double precision format in [-1.0 ..1.0] range.
`DqAdv205LoadCoeff()` can be used in streaming mode (ACB, VMAP, RTDMAP).

### *4.4.3 DqAdv205SetFilterMode*

**Syntax:**

`DqAdv205SetFilterMode(int hd, int devn, int fir, int channel, uint32 mode)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int fir` | fir stage number |
| `int channel` | channel number |
| `uint32 mode` | mode of operation |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | `fir`, `channel`, or `mode` is invalid |
| `DQ_NO_MEMORY` | error allocating memory |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-205 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets the FIR mode of a particular channel.

**Note:**

The following FIR modes are defined:

```
#define AI205_FIR_DISABLED       0      // filter disabled
#define AI205_FIR_DEFAULT        1      // set default parameters
#define AI205_FIR_DECRAT_ONLY    2      // program decimation ratio only
#define AI205_FIR_PROGRAMMED     6      // program filter and decimation
```

## 4.5 DNA-AI-207 layer

### 4.5.1 DqAdv207Read

**Syntax:**

```
int DqAdv207Read(int hd, int devn, int CLSize, uint32 *cl, uint32
*bData, double *fData)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Number of channels |
| `uint32 *cl` | Pointer to channel list with optional timestamp request |
| `uint32 *bData` | Pointer to raw data received from device |
| `double *fData` | Pointer to store converted voltage data (`NULL` if not required) |

**Output:**

| | |
|---|---|
| `uint32 *cl` | Channel list |
| `uint32 *bData` | Received binary data |
| `double *fData` | Received voltage data |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-207 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, `bData` is NULL, or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function uses `DqReadAIChannel()` but converts data using internal knowledge of input range and gain of every channel.

When this function is called for the first time, the firmware stops any ongoing operation on the device specified and reprograms it in accordance with the channel list supplied. This function uses the preprogrammed CL update frequency – 10 Hz. One can reprogram the update frequency by calling the `DqCmdSetClk()` command after the first call to `DqAdv207Read()`.

Thus, the user cannot call this function when the layer is involved in any streaming or data mapping operations.

If the user specifies a short timeout delay, this function can time out when called for the first time because it is executed as a pending command and layer programming takes up to 10 ms.

Once this function is called, the layer continuously acquires data and every next call function returns the latest acquired data.

To cancel ongoing sampling, call the same function with 0xffffffff as a channel number.

**Note:**

When reading thermocouples, please refer to the AI-207 thermocouple sample code for converting the voltage reading from `DqAdv207Read()`; thermocouple samples provide a TCConversion.h header, which contains voltage-to-temperature conversion functions.

If you are reading a thermocouple, note the following:

1. Make sure to set the gain properly on the AI-207 to measure thermocouples. Leaving a gain at 1 will result in noisy, inaccurate measurements.
2. You do need a CJC to measure thermocouples. Our terminal panel, STP-AI-207TC, provides the necessary CJC (and sample code provided with the installation implements a conversion based on this CJC on our STP panel).

### *4.5.2 DqAdv207ReadChannel*

**Syntax:**

```
int DqAdv207ReadChannel(int hd, int devn, uint32 measurement, uint32
clentry, int *samples, uint32 *data, double *volts)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | device number |
| `uint32 measurement` | what to measure |
| `uint32 clentry` | channel and gain settings |
| `int *samples` | number of samples requested; must be between 1 and 120 |
| `uint32 *data` | pointer to buffer for retrieved binary data (18-bit) |
| `double *volts` | pointer to buffer for value in volts - can be `NULL` |

**Output:**

| | |
|---|---|
| `int *samples` | number of samples retrieved |
| `uint32 *data` | pointer to retrieved binary data (18-bit) |
| `double *volts` | pointer to value in volts |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-207 |
| `DQ_BAD_PARAMETER` | measurement is not one of the valid `DQL_IOCTL208_READ` constants, `clentry` is not a valid channel number, `samples` is NULL or requested value is not between 1 and 120, or `data` is NULL |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function performs “raw” measurements of the following values:

- `DQL_IOCTL208_READ_AGND`: connect both differential inputs of the PGA to analog ground
- `DQL_IOCTL208_READ_REF`: read 2.5V voltage reference
- `DQL_IOCTL208_READ_Rs`: measure switch resistance *Rs*
- `DQL_IOCTL208_READ_Rx`: measure multiplexer resistance
- `DQL_IOCTL208_READ_Ra`: measure shunt resistor *Ra*
- `DQL_IOCTL208_READ_Rb`: measure shunt resistor *Rb*
- `DQL_IOCTL208_READ_SS`: measure *S+* to *S-*
- `DQL_IOCTL208_READ_PP`: measure *P+* to *AGND*
- `DQL_IOCTL208_READ_PS`: measure *PS+* to *AGND*
- `DQL_IOCTL208_READ_5k`: measure 5k resistance
- `DQL_IOCTL208_READ_PSM`: measure S+ thru the PS+ line

Because the resistance can differ from channel to channel (because current is flowing through different channels of the same multiplexer which can have different resistances) one should set up the channel number to be used. This function returns the number of samples requested to perform averaging. Data is returned in raw format.

**Note:**

None

## *4.5.3 DqAdv207SetAutozero*

**Syntax:**

```
int DqAdv207SetAutozero(hd, int devn, int* onoff)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | device number |
| `int32 *onoff` | pointer to boolean |

**Output:**

| | |
|---|---|
| `int32 *onoff` | previous on/off state of the Autozero |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-207 or AI-208 |
| `DQ_BAD_PARAMETER_2` | `onoff` is `NULL` |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |

|  |  |
|---|---|
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function turns the autozeroing capability of the AI-207 on or off. By default the autozeroing on the AI-207 is OFF.

**Note:**

## *4.6 DNA-AI-208 layer*

### *4.6.1 DqAdv208Read*

**Syntax:**

```
int DqAdv208Read(int hd, int devn, int CLSize, uint32 *cl,
uint32 *bData, double *fData)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Number of channels |
| `uint32 *cl` | Pointer to channel list with optional timestamp request |
| `uint32 *bData` | Pointer to raw data received from device |
| `double *fData` | Pointer to store converted voltage data (`NULL` if not required) |

**Output:**

| | |
|---|---|
| `uint32 *cl` | Channel list |
| `uint32 *bData` | Received binary data |
| `double *fData` | Received voltage data |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-208 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, `bData` is NULL, or a channel number in `cl` is invalid |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function uses `DqReadAIChannel()` but converts data using internal knowledge of input range and gain of every channel.

When this function is called for the first time, the firmware stops any ongoing operation on the device specified and reprograms it in accordance with the channel list supplied. This function uses the preprogrammed CL update frequency – 10Hz. One can reprogram the update frequency by calling the `DqCmdSetClk()` command after the first call to `DqAdv208Read()`.

Thus, the user cannot call this function when the layer is involved in any streaming or data mapping operations.

If the user specifies a short timeout delay, this function can time out when called for the first time because it is executed as a pending command and layer programming may take up to 10ms to complete.

Once this function is called, the layer continuously acquires data and every next call function returns the latest acquired data.

If one would like to cancel ongoing sampling, call the same function with 0xffffffff as a channel number.

**Note:**

None

## 4.6.2 DqAdv208SetAutozero

**Syntax:**

`int DqAdv208SetAutozero(hd, int devn, int* onoff)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int32 *onoff` | Pointer to boolean |

**Output:**

| | |
|---|---|
| `int32 *onoff` | Previous on/off state of the Autozero |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-207 or an AI-208 |
| `DQ_BAD_PARAMETER_2` | `onoff` is `NULL` |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function turns the autozeroing capability of the AI-208 on or off.

By default the autozeroing on the AI-208 is OFF.

**Note:**

### *4.6.3 DqAdv208SetControl*

**Syntax:**

```
int DqAdv208SetControl(int hd, int devn, uint32 control,
uint32 value, uint32 *data)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 control` | What to set |
| `uint32 value` | Value to write |
| `uint32 *data` | Pointer to output data (pass `NULL` if you don't need it):<br>For `DQL_IOCTL208_SET_Ra` and `DQL_IOCTL208_SET_Rb`, to switch shunt calibration on, set `*data` to `TRUE`. |

**Output:**

| | |
|---|---|
| `uint32 *data` | Output data (generally, the value written or undefined data.) |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-208 |
| `DQ_BAD_PARAMETER` | `control` is not one of the valid `DQL_IOCTL208_SET` constants |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function allows setting up different internal parameters.

The following sub-functions are available, selected by `control`:

- `DQL_IOCTL208_SET_SSPERCHAN`: set samples per channel ACB/DMap mode
- `DQL_IOCTL208_SET_Ra`: set value for shunt calibration resistor A in 256 steps (*P*+ to *S*+)
- `DQL_IOCTL208_SET_Rb`: set value for shunt calibration resistor B in 256 steps (*S*+ to *P*-)
- `DQL_IOCTL208_SET_EXC_A`: set excitation DAC A
- `DQL_IOCTL208_SET_EXC_B`: set excitation DAC B
- `DQL_IOCTL208_SET_EXC_CH`: switch on/off excitation channels

**Note:**

Set `*data` to `TRUE` - to switch shunt calibration on when `control` is `DQL_IOCTL208_SET_Ra` or `DQL_IOCTL208_SET_Rb`

### 4.6.4 DqAdv208SetExcVoltage

**Syntax:**

```
int DqAdv208SetExcVoltage(int hd, int devn, ExcVoltA, float ExcVoltB,
uint32 chA, uint32 chB, float *readbackA, float *readbackB)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `float ExcVoltA` | Excitation voltage for excitation DAC A |
| `float ExcVoltB` | Excitation voltage for excitation DAC B |
| `int chA` | Channel to measure excitation voltage for DAC A |
| `int chB` | Channel to measure excitation voltage for DAC B |
| `float *readbackA` | Pointer to buffer for actual excitation voltage read back from DAC A |
| `float *readbackB` | Pointer to buffer for actual excitation voltage read back from DAC B |

**Output:**

| | |
|---|---|
| `float *readbackA` | Actual excitation voltage read back from DAC A |
| `float *readbackB` | Actual excitation voltage read back from DAC B |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-208 |
| `DQ_BAD_PARAMETER` | illegal `ExcVoltA`, `ExcVoltB`, `chA` and/or `chB` value, or `readbackA` or `readbackB` is NULL |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_DATA_ERROR` | device data error; see NOTE below |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Set excitation voltage for excitation sources A and B and measure it back using specified channels. The AI-208 layer is capable of providing two sources of excitation voltage.

Excitation A is connected to even channels and B is connected to odd channels. The excitation voltage can be selected and set at any level from 1.5V to 10V. This function sets up excitation voltage as close as possible to the requested level and reads it back from the selected channels.

The user can select either channels 0x10 through 0x17 to read the excitation voltage from the *Px+* terminal (four-wire connection), or channels from 0x20 thru 0x27 to read the excitation voltage from *PSx+* terminals (six-wire connection). All readings are performed relative to *AGND*.

The user must use the read-back excitation voltage from the terminal because of DACs; there is a voltage drop in the strain gage leads and DAQ output quantization error amounts to 1/1024 of the range.

**Note:**

Please ensure your sensor is properly wired before using this function. Without the proper wiring, this function can return a `DQ_DATA_ERROR` due to the inability to properly compensate for line losses.

This function must be called before starting acquisition or reading channels to set up a proper excitation voltage source.

### 4.6.5 DqAdv208ReadChannel

**Syntax:**

```
int DqAdv208ReadChannel(int hd, int devn, uint32 measurement,
uint32 clentry, int *samples, uint32 *data, double *volts)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 measurement` | What to measure |
| `uint32 clentry` | Channel and gain settings |
| `int *samples` | Number of samples requested; must be between 1 and 120 |
| `uint32 *data` | Pointer to buffer for retrieved binary data (18-bit) |
| `double *volts` | Pointer to buffer for value in volts - can be `NULL` |

**Output:**

| | |
|---|---|
| `int *samples` | Number of samples retrieved |
| `uint32 *data` | Pointer to retrieved binary data (18-bit) |
| `double *volts` | Pointer to value in volts |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-208 |
| `DQ_BAD_PARAMETER` | measurement is not one of the valid `DQL_IOCTL208_READ` constants, `clentry` is not a valid channel number, `samples` is NULL or requested value is not between 1 and 120, or `data` is NULL |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function performs “raw” measurements of the following values:

- `DQL_IOCTL208_READ_AGND`: connect both differential inputs of the PGA to analog ground
- `DQL_IOCTL208_READ_REF`: read 2.5V voltage reference
- `DQL_IOCTL208_READ_Rs`: measure switch resistance *Rs*
- `DQL_IOCTL208_READ_Rx`: measure multiplexer resistance
- `DQL_IOCTL208_READ_Ra`: measure shunt resistor *Ra*
- `DQL_IOCTL208_READ_Rb`: measure shunt resistor *Rb*
- `DQL_IOCTL208_READ_SS`: measure *S+* to *S-*
- `DQL_IOCTL208_READ_PP`: measure *P+* to *AGND*
- `DQL_IOCTL208_READ_PS`: measure *PS+* to *AGND*
- `DQL_IOCTL208_READ_5k`: measure 5k resistance
- `DQL_IOCTL208_READ_PSM`: measure S+ thru the PS+ line

Because the resistance can differ from channel to channel (because current is flowing through different channels of the same multiplexer which can have different resistances) one should set up the channel number to be used. This function returns the number of samples requested to perform averaging. Data is returned in raw format.

**Note:**

None

### *4.6.6 DqAdv208MeasureParams*

**Syntax:**

```
int DqAdv208MeasureParams(int hd, int devn, pDQ208CCOND pcond,
pDQ208CPRM pprm)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `pDQ208CCOND pcond` | Pointer to buffer for measurement conditions |
| `pDQ208CPRM pprm` | Pointer to buffer for measured parameters |

**Output:**

| | |
|---|---|
| `pDQ208CCOND pcond` | Measurement conditions |
| `pDQ208CPRM pprm` | Measured parameters |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-208 |
| `DQ_BAD_PARAMETER` | `pcond` or `pprm` is NULL |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |

| | |
|---|---|
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function is used to measure a variety of AI-208 front-end parameters (see channel equivalent diagram in the *PowerDNA User Manual*):

| | |
|---|---|
| *Vref* | Reference voltage, Volts |
| *Vexc* | Excitation voltage, Volts |
| *Vs* | Vmeas for *Rs*, Volts |
| *Rs* | Switch resistance, Ohms |
| *Vx* | Vmeas for *Rx*, Volts |
| *Rx* | Mux resistance, Ohms |
| *Va* | Vmeas for *Ra*, Volts |
| *Ra* | Resistance of shunt resistor Ra (plus 5k constant!), Ohms |
| *Vb* | Vmeas for *Rb*, Volts |
| *Rb* | Resistance of shunt resistor *Rb* (plus 5k constant!), Ohms |

Before the function can measure these parameters, the user should specify measurement conditions:

| | |
|---|---|
| *channel* | Channel used for measurements |
| *ExcA* | Excitation level A (even channels, 16 bit) |
| *ExcB* | Excitation level B (odd channels, 16 bit) |
| *Ra* | Shunt A level (8 bit, 256 positions from 0 to 200k) |
| *Rb* | Shunt B level (8 bit, 256 positions from 0 to 200k) |

Here are the structures used:

```
// calibration measurements
typedef struct {
    double Vref;    // reference voltage (Volts)
    double Vexc;    // excitation voltage (Volts)
    double Vs;      // Vmeas for Rs (Volts)
    double Rs;      // switch resistance (Ohms)
    double Vx;      // Vmeas for Rx (Volts)
    double Rx;      // mux resistance (Ohms)
    double Va;      // Vmeas for Ra (Volts)
    double Ra;      // Resistance of shunt resistor Ra (plus 5k
                    // constant!) (Ohms)
    double Vb;      // Vmeas for Rb (Volts)
    double Rb;      // Resistance of shunt resistor Rb (plus 5k
                    // constant!) (Ohms)
} DQ208CPRM, *pDQ208CPRM;

// calibration measurements parameters
typedef struct {
    uint32 channel; // channel to measure at
    uint16 ExcA;    // excitation level A (even channels, 16 bit)
    uint16 ExcB;    // excitation level B (odd channels, 16 bit)
    uint8 Ra;       // Shunt A level (8 bit, 256 positions)
    uint8 Rb;       // Shunt B level (8 bit, 256 positions)
} DQ208CCOND, *pDQ208CCOND;
```

**Note:**

The device should be in configuration mode; these measurements cannot be done along with acquisition.

### *4.6.7 DqAdv208ReadAutogain*

**Syntax:**

```
int DqAdv208ReadAutogain(int hd, int devn, uint32 measurement,
uint32 clentry, int* samples, uint32* dt, double* df)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 measurement` | What to measure |
| `uint32 clentry` | Channel and gain settings |
| `int *samples` | Number of samples requested |
| `uint32 *dt` | Pointer to buffer for retrieved binary data (18-bit) |
| `double *df` | Pointer to buffer for values in volts |

**Output:**

| | |
|---|---|
| `int *samples` | Number of samples retrieved |
| `uint32 *dt` | Retrieved binary data (18-bit) |
| `double *df` | Values in volts |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-208 |
| `DQ_BAD_PARAMETER` | measurement is not one of the valid `DQL_IOCTL208_READ` constants, `clentry` is not a valid channel number, `samples` is NULL or requested value is not between 1 and 64, or `dt` or `df` is NULL |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads data and automatically selects the proper gain.

**Notes:**

None

### *4.6.8 DqAdv208ShuntCal*

**Syntax:**

```
int DqAdv208ShuntCal(int hd, int devn, uint32 channel,
uint32 shunt_set, int cycles, double excitation, double r_shunt,
double* r_actual, double* v_exc, double* v_gage, double* v_shunt)
```

**Command:**

None

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 channel` | What channel to measure |
| `uint32 shunt_set` | Type of shunt to apply:<br>`DQL_IOCTL208_READ_Ra`<br>`DQL_IOCTL208_READ_Rb` |
| `int cycles` | Number of measurement cycles to average (1..100) |
| `double excitation` | Requested excitation voltage: 2.0V..9.0V |
| `double r_shunt` | Requested shunt value: 10k..170k |

**Output:**

| | |
|---|---|
| `double* r_actual` | Actual shunt resistance |
| `double* v_exc` | Actual excitation voltage |
| `double* v_gage` | Actual voltage between S+ and S- |
| `double* v_shunt` | Actual voltage between S+ and S- with shunt applied |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-208 |
| `DQ_BAD_PARAMETER` | one of parameters supplied are bad |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| `DQ_EXECUTION_ERROR` | Error while working with shunt cal, most likely connections |
| Other negative values | low level IOM error |

**Description:**

The function performs shunt calibration measurements.

**Notes:**

1. The function takes approximately 5 seconds to execute with 10 cycles.
2. PS+ should be connected to S+ on the screw terminal. Otherwise, the shunt cannot be measured and only a four-wire interface is supported.
3. Shunt A is applied between S+ and P+; shunt B between S+ and AGND
4. Measurement range of r_shunt can be from 10k to 170k (shunt has 5k 0.1% resistor and 200k digital potentiometer with good ppm/C but with 30% tolerance). Some layers may be capable of providing shunt calibration with a resistance exceeding 170k.

## 4.7 DNA-AI-211 layer

### 4.7.1 DqAdv211Read

**Syntax:**

```
int DqAdv211Read(int hd, int devn, int CLSize, uint32 *cl,
uint16 *bData, double *fData)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | number of channels |
| `uint32 *cl` | pointer to channel list with optional timestamp request |
| `uint32 *bData` | pointer to raw data received from device, including LED status bits |
| `double *fData` | pointer to store converted voltage data (`NULL` if not required) |

**Output:**

| | |
|---|---|
| `uint32 *bData` | raw data received from device |
| `double *fData` | raw data converted to voltage |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-211 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, `bData` is NULL, or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

When this function is called for the first time, the firmware stops any ongoing operation on the device specified and reprograms it according to the channel list supplied. This function uses a fixed CL update frequency – 1953Hz. Thus, the user cannot perform this function call when the layer is involved in any streaming or data mapping operations.

If the user specifies a short timeout delay, this function can time out when called for the first time because it is executed as a pending command and layer programming takes up to 10ms.

Once this function is called, the layer continuously acquires data and every call to the function returns the latest acquired data.

If one would like to cancel ongoing sampling, call the same function with `0xffffffff` as a channel number.

The DNA-STP-211 has on-board LEDs which indicate whether there is an open, short or OK circuit status. Bits 26 and 27 in `<bData>` indicate which state the circuit is currently in, explained in the table below

| Bits 27&26 | Circuit Status | LED Status |
|---|---|---|
| 00 | Ok | Green |
| 01 | Short | Blinking Red |
| 10 | Open | Red |
| 11 | Invalid* | Red |

*Invalid state can be reached during while board is being configured. Subsequent reads should set the status to one of the valid states.

### *4.7.2 DqAdv211SetCfgChannel*

**Syntax:**

`int DqAdv211SetCfgChannel(int hd, int devn, pDQCFGCH_211 cdata)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `pDQCFGCH_211 cdata` | Pointer to 211 channel config structure, see below |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-211 |
| `DQ_BAD_PARAMETER` | Illegal value in DQCFGCH_211 structure |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up the channel configuration parameters for the AI-211. The user may configure as many parameters as he chooses. Any parameter that is not enabled by a <mask> bit will retain its present value.

To accomplish AI-211 channel configuration the user must allocate, initialize, and pass a pointer to a `pDQCFGCH_211` structure. This structure is defined as:

```
typedef struct {
    uint16 channels;    // channel select bits
    uint16 mask;        // bitwise indicates which of the following struct fields are valid
    uint16 biasdrive;   // set drive current for iepe sensors, range 0-255 approx = 0-8mA
    uint16 biasonoff;   // DQ_211_BIAS_ON  or  DQ_211_BIAS_OFF
    uint16 comphi;      // 12 bit comparison value for open sensor detection
    uint16 complo;      // 12 bit comparison value for shorted sensor detection
    uint16 alarmctrl;   // led alarm control, see defines below
    uint16 hpf;         // high pass filters. see defines below
    uint16 offset;      // test mode.
    uint16 anafilt;     // 48kHz anti-alias filter. DQ_211_ANALOG_FILTER_ON or
                        //  DQ_211_ANALOG_FILTER_OFF
    uint16 main_enb;    // enable dataflow from main ADC, see defines below
    uint16 sec_enbs;    // secondary enables. 0= sec. off, 1= secondary ON
    uint16 secn;        // determine update rate for secondary (led comparison) converter
} DQCFGCH_211, *pDQCFGCH_211;
```

`<channels>` flags indicate which channels should be written. The following flags are defined:

```
  #define DQ_AI211_SEL_CHAN_0        (0x01)
  #define DQ_AI211_SEL_CHAN_1        (0x02)
  #define DQ_AI211_SEL_CHAN_2        (0x04)
  #define DQ_AI211_SEL_CHAN_3        (0x08)
  #define DQ_AI211_SEL_CHAN_ALL      (0x0f)
```

More than one channel may be specified by oring multiple channel flags together.

`<mask>` flag bits that select which parameters will be written. The following flags are defined:

```
#define DQAI211_CFGCH_DEFAULTSET (1L<<11) // if=1 set all values to default state
#define DQAI211_BIASDRIVESET     (1L<<0)  // =1 to set "biasdrive" parameter
#define DQAI211_BIASONOFFSET     (1L<<1)  // =1 to set "biasonoff" parameter
#define DQAI211_COMPHISET        (1L<<2)  // =1 to set "comphi" parameter
#define DQAI211_COMPLOSET        (1L<<3)  // =1 to set "complo" parameter
#define DQAI211_ALARMCTRLSET     (1L<<4)  // =1 to set "alarmctrl" parameter
#define DQAI211_HPFSET           (1L<<5)  // =1 to set "hpf" parameter
#define DQAI211_OFFSETSET        (1L<<6)  // =1 to set "offset" parameter
#define DQAI211_ANAFILTSET       (1L<<7)  // =1 to set "anafilt" parameter
#define DQAI211_MAINENBSET       (1L<<8)  // =1 to set "main_enb" parameter
#define DQAI211_SECENBSSET       (1L<<9)  // =1 to set "secenbs" parameter
#define DQAI211_SECNSET          (1L<<10) // =1 to set "secn" parameter
```

The `DQAI211_CFGCH_DEFAULTSET` flag bit is used to easily set all of the channel configuration values to their default state. Before setting up a custom configuration , it is recommended to first set all channels to their default state by setting `<channels>` to `AI211_SEL_CHAN_ALL`, setting `<mask>` to `DQAI211_CFGCH_DEFAULTSET` and calling `DqAdv211SetCfgChannel().`

`< biasdrive >` Sets the amount of drive current for the iepe sensor. When the `DQAI211_BIASDRIVESET` flag bit is set in `<mask>`, the `<biasdrive>` value is set. The following scaling macro is defined to allow for easy translation to the correct values.

```
    #define DQ211_DRIVE_CURRENT(I) ((I/8.0)*255) //drive current scaling macro.
```

The driver accepts values from zero to 255 which map to drive currents from zero to 8mA. Values less than zero or greater than 8.0 will return an error.

`< biasonoff >` Turns the drive current for the iepe sensor on or off . When the `DQAI211_BIASDRIVESET` flag bit is set in `<mask>`, The value in `<biasonoff>` is set. The following two values are defined for this purpose:

```
     #define DQ_211_BIAS_ON          (1)
     #define DQ_211_BIAS_OFF         (0)
```

`<comphi>` Sets the 12 bit comparison value for the open sensor detector. When the `DQAI211_COMPHISET` flag bit is set in `<mask>`, The value in `<comphi>` is set. Two defines are provided:

```
  #define DQ_211_COMP_HI_STD     (0xfa0)  // standard value for doing comparison
  #define DQ_211_COMP_HI_DEFAULT (0xfff)  // default value ,disables comparison
```

It is suggested to keep the values between 0xf00 and 0xff0. Setting a value of 0xfff will turn the comparison off.

<complo> Sets the 12 bit comparison value for the shorted sensor detector. When the `DQAI211_COMPLOSET` flag bit is set in <mask>, The value in <complo> is set. Two defines are provided:

```
  #define DQ_211_COMP_LO_STD     (0xc0) // standard value for doing comparison
  #define DQ_211_COMP_LO_DEFAULT (0x0)  // default value ,disables comparison
```

It is suggested to keep the values between 0x30 and 0xe0. Setting a value of zero will turn the comparison off.

< alarmctrl> Sets the control settings for the visual alarm LED. When the `DQAI211_ALARMCTRLSET` flag bit is set in <mask>, The value in < alarmctrl > is set. The following defines are provided:

```
   #define DQ_211_ALARM_ON      (0x3)  // LEDs are controlled by comparison
                                       // registers and secondary adc
   #define DQ_211_ALARM_OFF     (0)    // LEDs are off
   #define DQ_211_ALARM_RED     (0x4)  // LED controlled by program, red on
   #define DQ_211_ALARM_GREEN   (0x8)  // LED controlled by program, green on
   #define DQ_211_ALARM_ORANGE  (0x0c) // LED controlled by program, orange on
```

<hpf> Sets the control settings for high pass filtering or DC coupling. When the `DQAI211_HPFSET` flag bit is set in <mask>, The value in <hpf> is set. The following defines are provided:

```
   #define DQ_211_HPF_DC        (1L<<0)     // DC coupling
   #define DQ_211_HPF_POINT1_HZ (1L<<1)     // 0.1 Hz cutoff high pass filter
   #define DQ_211_HPF_1_HZ      (1L<<2)     // 1.0 Hz cutoff high pass filter
   #define DQ_211_HPF_10_HZ     (1L<<3)     //  10 Hz cutoff high pass filter
```

< offset> Sets the control settings for a test mode. This is not normally set by a user. When the `DQAI211_OFFSETSET` flag bit is set in <mask>, The value in <offset> is set. The following defines are provided:

```
   #define DQ_211_OFFSET_TEST_ON   (1)         // test mode on
   #define DQ_211_OFFSET_TEST_OFF  (0)         // test mode off - default value
```

When the combination of DQ_211_BIAS_OFF and DQ_211_OFFSET_TEST_ON occurs, the system internally grounds the sensor input for adjustment purposes. No sensor connections are allowed at this time.

< anafilt> Sets the 48KHz analog anti-aliasing filter ON or OFF. When the `DQAI211_ANAFILTSET` flag bit is set in <mask>, The value in <anafilt> is set. The following defines are provided:

```
   #define DQ_211_ANALOG_FILTER_ON   (1)  // leave  anti-aliasing filter ON
   #define DQ_211_ANALOG_FILTER_OFF  (0)
```

< main_enb> Provides additional control over the main A/D converter for special applications. Control for the main converter is normally provided automatically by the DqAdv211Read() or the

ACB and DMap control functions. When the `DQAI211_MAINENBSET` flag bit is set in <mask>, The value in <main_enb> is set. The following defines are provided:

```
   #define DQ_211_MAIN_FLOW_OFF    (0)
   #define DQ_211_MAIN_FLOW_ON     (1
```

< sec_enbs> Provides control over the secondary A/D converter used by the visual alarm LED function. When the `DQAI211_SECENBSSET` flag bit is set in <mask>, The value in <sec_enbs> is set. The following defines are provided:

```
   #define DQ_211_SEC_ENB_OFF    (0)      // SECondary converter OFF
   #define DQ_211_SEC_ENB_LED    (0x1)    // SEC converter updates LED comparison
```

The converter must be turned on using the DQ_211_SEC_ENB_LED value in order for the visual alarm LED to function.

< secn> This value sets the number of main converter reads per secondary converter read. When the `DQAI211_SECNSET` flag bit is set in <mask>, The value in <secn> is set. The following defines are provided:

```
   #define DQ_211_SEC_N_STD   (2400)    // Number of main reads per SECondary read
   #define DQ_211_SEC_N_OFF   (0)       // Disable secondary data flow
```

The secondary converter's monitoring of the status of the iepe sensor connection for the setting of the visual alarm LED does not need to occur at the same rate as the primary A/D converter. This setting allows the user to set the rate at which the secondary converter does this monitoring. The secondary data is also transferred across the isolation barrier at this same rate to be made available for some special functions. Setting the value to DQ_211_SEC_N_OFF disables the transfer of this data but does not inhibit the functioning of the visual alarm LED. The DQ_211_SEC_N_STD define is the recommended setting for this value, but it may be set to any value from zero to 32,767

### *4.7.3 DqAdv211SetCfgLayer*

**Syntax:**

```
int DqAdv211SetCfgLayer(int hd, int devn, pDQCFGLAYER_211 ldata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `pDQCFGLAYER_211 ldata` | Pointer to 211 layer config structure, see below |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by devn does not exist or is not an AI-211 |
| `DQ_BAD_PARAMETER` | One of the values in the pDQCFGLAYER_211 structure is set to an illegal value |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |

| | |
|---|---|
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up advanced layer configuration parameters for the AI-211. These settings apply to all channels on the layer. In the normal case, these configuration settings are set automatically by the DqAcbInitOps() function. Use these settings to override standard behavior for special applications. The user may configure as many parameters as he chooses. Any parameter that is not enabled by a <mask> bit will retain its present value.

To accomplish AI-211 layer configuration the user must allocate, initialize, and pass a pointer to a `pDQCFGLAYER_211` structure. This structure is defined as:

```
typedef struct {
    uint16 mask;    // bitwise indicate which of the following fields are valid
    uint16 clksrc;  // select clock source for divider.
    uint16 clkdiv;  // clock divider. output (1MHz max)
                    // clkdiv=0 passes clksrc w/o change.
                    // Do not set values less than 65 or 23 when 66MHz or 24MHz
selected.
                    // Sample rate is:  clksrc/((clkdiv+1)*8).
    uint16 fmtr;    // reduced precision data format 1= reduced, 0 = normal.
    uint16 avg_factor; //Set the averaging factor. 0=1, 1=2, 2=4, 3=8, etc.
    uint16 dec_factor; // Set decimation factor
} DQCFGLAYER_211, *pDQCFGLAYER_211
```

`<mask>` flag bits that select which parameters will be written. The following flags are defined:

```
   #define DQAI211_CFGLAYER_DEFAULTSET    (1L<<0) // =1  to set default state
   #define DQAI211_CLKSRCSET    (1L<<1)  // =1 if "clksrc" contains valid data
   #define DQAI211_CLKDIVSET    (1L<<2)  // =1 if "clkdiv" contains valid data
   #define DQAI211_FMTRSET      (1L<<3)  // =1 if "fmtr" contains valid data
   #define DQAI211_AVGFACTORSET (1L<<4)  // =1 if "avg_factor" contains valid data
   #define DQAI211_DECFACTORSET   (1L<<8)   // =1 if dec_factor has valid data
   #define DQAI211_FIR_BY_DECFACTOR (1L<<5)   // Set default FIR to follow sample
rate determined by decimation factor
```

The `DQAI211_CFGLAYER_DEFAULTSET` flag bit is used to easily set all of the layer configuration values to their default state. Before setting up a custom configuration , it is recommended to first set all values to their default state by setting <`mask`> to `DQAI211_CFGLAYER_DEFAULTSET` and calling `DqAdv211SetCfgLayer(). When DQAI211_CFGLAYER_DEFAULTSET` is set, all other <`mask`> flag bits are ignored.

`<clksrc>` This value selects the source of the clock to be used to pace the A/D conversion on the layer. This clock source is routed to the clock divider that is set by `<clkdiv>` below. When the `DQAI211_CLKSRCSET` flag bit is set in <`mask`>, the value in <`clksrc`> is set. The following defines are provided:

```
#define DQ_211_CLK_66MHZ            (0)
#define DQ_211_CLK_24MHZ            (0x10)
#define DQ_211_CLK_SYNC2            (0x18)
```

```
#define DQ_211_CLK_SYNC0_BUS        (0x8)
#define DQ_211_CLK_SYNC1_BUS        (0x9)
#define DQ_211_CLK_SYNC2_BUS        (0xa)
#define DQ_211_CLK_SYNC3_BUS        (0xb)
```

`<clkdiv>` This value sets the clock divider used to set the rate of the A/D conversion. The maximum allowable frequency to pace the main A/D is 1Mhz.
Do not use values less than 65 when 66MHz is selected or less than 23 when 24MHz is selected. The output frequency is clksrc/(clkdiv+1). Max value is 1023. When the `DQAI211_CLKDIVSET` flag bit is set in `<mask>`, the value in `<clkdiv>` is set. The default value is 65.

`<fmtr>` This value sets a reduced precision mode, reserved for special applications. When the `DQAI211_FMTRSET` flag bit is set in `<mask>`, the value in `<fmtr>` is set. The following defines are provided:

```
      #define DQ_211_FMTR_NORMAL          (0)
      #define DQ_211_FMTR_REDUCED         (1)
```

The default value is `DQ_211_FMTR_NORMAL`.

`<avg_factor>` This value sets the amount of averaging performed. The number of samples averaged together is always a power of 2. Setting `<avg_factor>` to zero gives no averaging. Setting `<avg_factor>` to 1 gives 2 samples averaged, setting 2 averages 4 samples, 3 averages 8 , etc. The maximum value for `<avg_factor>` is 15, which averages 32,768 samples. When the `DQAI211_AVGFACTORSET` flag bit is set in `<mask>`, the value in `<avg_factor>` is set.

`<dec_factor>` This value sets the amount of decimation performed. The decimation ratio is always a power of 2. Setting `<dec_factor>` to zero gives no decimation. Setting `<dec_factor>` to 1 retains 1 of every 2 samples, setting 2 retains 1 of every 4 samples, 3 retains 1 of 8 , etc. The maximum value for `<dec_factor>` is 16, which retains 1 out of every 65536 samples. When the `DQAI211_DECFACTORSET` flag bit is set in `<mask>`, the value in `<dec_factor>` is set.

`The DQAI211_FIR_BY_DECFACTOR flag bit also uses the data in <dec_factor>. If the` `DQAI211_FIR_BY_DECFACTOR` bit is set and `DQAI211_DECFACTORSET` is also set,
the default FIR filter coefficients will be adjusted along with the decimation factor. This maintains the correct low-pass filter response. As the decimation factor decreases the data rate by a factor of 2 with each increment, the cutoff frequency of the FIR filter be reduced correspondingly.

### *4.7.4 DqAdv211SetFIR*

**Syntax:**

```
int DqAdv211SetFIR(int hd, int devn, int channel, int mask, int
decrat, int tapsize, double* data)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |

| | |
|---|---|
| `int channel` | bit field to select channel, see below |
| `int mask` | bit field to select which FIR setting functions to perform, see below |
| `int decrat` | desired decimation ratio (`NULL` if not required), see below |
| `int tapsize` | number of taps in filter, length of the following *data (`NULL` if not required), see below |
| `double *data` | pointer to filter taps data (`NULL` if not required), see below |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-211 |
| `DQ_BAD_PARAMETER` | No channel specified, decimation ratio is illegal value, tapsize is illegal value or *data is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function can be used to perform FIR configuration and control functions for the AI-211 when the user desires to specify his own filter coefficients and decimation rates. In the normal case, these configuration settings are set automatically by the `DqAcbInitOps()` function. Use these settings to override standard behavior for special applications. The user may configure as many parameters as he chooses. Any parameter that is not enabled by a <mask> bit will retain its present value.

`<channel>` parameter indicates which channels will get their FIR configuration changed. The following flags are defined:

```
#define DQ_AI211_SEL_CHAN_0        (0x01)
#define DQ_AI211_SEL_CHAN_1        (0x02)
#define DQ_AI211_SEL_CHAN_2        (0x04)
#define DQ_AI211_SEL_CHAN_3        (0x08)
#define DQ_AI211_SEL_CHAN_ALL      (0x0f)
```

More than one channel may be specified by oring multiple channel flags together.

<mask> parameter flag bits that select which parameters will be written. The following flags are defined:

```
#define DQ_AI211_FIR_SET_DEFAULT      (0x8) //set and enable the default filter
#define DQ_AI211_FIR_COEFF_LOAD       (0x4) // load taps and coefficients
#define DQ_AI211_FIR_SET_DECIMATION_RATE (0x2) // set decimation rate
#define DQ_AI211_FIR_ENABLE           (0x1) // enable fir filter
#define DQ_AI211_FIR_DISABLE          (0x0) // disable fir filter
#define DQ_AI211_FIRFIRST_ENABLE      (0x0) // perform FIR before averaging
#define DQ_AI211_FIRFIRST_DISABLE     (0x10)// perform FIR after averaging
#define DQ_AI211_FIR_SET_SKIP         (0x20)// set skip count by user
```

The `DQ_AI211_FIR_SET_DEFAULT` flag bit is used to clear all of the FIR configuration values to their unprogrammed state. Before setting up a completely custom configuration , it is recommended to first set all channels to their default state by setting <channel> to `DQ_AI211_SEL_CHAN_ALL`, setting <mask> to `DQ_AI211_FIR_SET_DEFAULT` and calling `DqAdv211SetFIR()`. When the `DQ_AI211_FIR_SET_DEFAULT` flag is set, all of the other <`mask`> parameter bits are ignored. After using the `DQ_AI211_FIR_SET_DEFAULT` flag, additional calls to `DqAdv211SetFIR()`must be made to load the coefficients, set the decimation and enable the filters.

The `DQ_AI211_FIR_ENABLE` and `DQ_AI211_FIR_DISABLE` defines are used to enable or bypass the FIR filter when a custom FIR filter has been loaded. When the user does not make any calls to `DqAdv211SetFIR()`the standard settings are used and the system will automatically load and enable the FIR filter. No additional enabling is required.

The `DQ_AI211_FIRFIRST_ENABLE` and `DQ_AI211_FIRFIRST_DISABLE` defines are used to set the order in which the FIR filtering and the averaging takes place. When standard default settings are used, the system will automatically set the FIR filtering to occur first.

`<decrat>` parameter sets the decimation rate. Zero = no decimation. Setting a value of 1 discards one reading for every one kept, two discards two readings for every one kept, etc. For example, to decimate an 80kHz sample rate to 10kHz, use a value of 7. When the `DQ_AI211_FIR_SET_DECIMATION_RATE` flag bit is set in <`mask`> parameter, The value in <`decrat`> is set.
The default value is zero. Include the `DQ_AI211_FIR_ENABLE` flag with `DQ_AI211_FIR_SET_DECIMATION_RATE` when changing rates.

`<tapsize> and <*data>` parameters are used to load user defined filter coefficients to the selected FIR filters. `<tapsize>` is the number of coefficients expected in the array of double precision coefficients pointed to by `<*data>`. The coefficients should be in the [-1.0…1.0] range. The sum of all the taps should equal 1.0 . When the `DQ_AI211_FIR_COEFF_LOAD` flag bit is set in the<`mask`> parameter, the <`tapsize`> and <*data> values are set. It is recommended to set the desired decimation with the same function call that is used to set the filter coefficients.

### *4.7.5 DqAdv211SetPll*

**Syntax:**

```
int DqAdv211SetPll(int hd, double samplerate, double* sr_actual,
int* dec_fact, int line)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `double samplerate` | Desired sampling rate |
| `int line` | Sync line to assign signal |

**Output:**

| | |
|---|---|
| `double* sr_actual` | actual sample rate |
| `int *dec_fact` | Required decimation factor for AI-211 layer |

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | requested sample rate is too high or too low |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:** The PLL clock circuit on the CPU layer will be programmed and routed by this function. The PLL may be routed by way of the sync[1] or sync[3] lines by using the `DQ_EXT_SYNC1` or `DQ_EXT_SYNC3` constants for the 'line' parameter.

The AI-211 uses clock rates that are higher than the sampling rate and also uses decimation to reduce the effective sampling rate. This function will calculate the correct clock and decimation factor for the AI-211.

The returned decimation factor should be sent to the AI-211 layers using the DqAdv211SetCfgLayer() function as follows:

```
 DQCFGLAYER_211 ldata;
    ldata.mask = (DQAI211_DECFACTORSET | DQAI211_FIR_BY_DECFACTOR);
    ldata.dec_factor = (uint16)decm_factor;
    DqAdv211SetCfgLayer(hd0,DEVN,&ldata);
```

```
The Config word used by the second parameter of DqAcbInitOps() must enable the PLL
clock by specifying the DQ_LN_CVCKSRC1 constant. For example:
 #define CFG211        (DQ_LN_ENABLED \
                        |DQ_LN_ACTIVE \
                        |DQ_LN_GETRAW \
                        |DQ_LN_IRQEN \
                        |DQ_LN_CVCKSRC1 \
                        |DQ_LN_STREAMING \
                        |DQ_FIFO_MODEFIFO)
```

## 4.8 DNA-AI-212 layer

### 4.8.1 DqAdv212Read

**Syntax:**

```
int DqAdv212Read(int hd, int devn, int CLSize, uint32 *cl,
uint32 *bData, double *fData)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Number of entries in the channel list |
| `uint32 *cl` | Pointer to channel list with optional timestamp request |
| `uint32 *bData` | Pointer to raw data received from device (`NULL` if not required), min length = `CLSize` |
| `double *fData` | Pointer to store converted voltage data (`NULL` if not required), min length = `CLSize` |

**Output:**

| | |
|---|---|
| `uint32 *bData` | raw data received from device |
| `double *fData` | converted voltage data |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-212 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

When this function is called for the first time, the firmware stops any ongoing operation on the device specified and reprograms it according to the channel list supplied. The user cannot perform this function call when the layer is involved in any streaming or data mapping operations. Subsequent calls acquire data on the specified channels.

Channel list entries have following format:

```
bits 10:8 <gain setting>
bits  4:0 <channel number>
```

Macros `DQ_AI212_GAIN()` and `DQ_AI212_CHANGAIN()` may be used to assemble the channel list.

Additionally, users can set up channels with a bias voltage by ORing the `DQ_AI212_BIAS_ON_FLAG` flag with channels in the `cl` channel list or a burnout detection current source by ORing in the `DQ_AI212_BURNOUT_ON_FLAG` flag in `cl`:

`DQ_AI212_BIAS_ON_FLAG`:

Users can enable an optional thermocouple bias voltage on each of the AI-212 Ain(-) inputs. This feature is disabled by default. When turned ON, it connects mid-supply voltage to the Ain- terminal. This is used when measuring floating (ungrounded) thermocouples. The DNA-STP-AI-212 terminal panel provides a 100K resistance to ground, which negates the need for the bias feature. In applications where the DNA-STP-AI-212 terminal panel is not used, the `DQ_AI212_BIAS_ON_FLAG` may be bitwise ORed with the channel list entries to enable it.

`DQ_AI212_BURNOUT_ON_FLAG`:

Users have the option of enabling an optional burnout detection current source. Burnout detection should be used prior to normal operation and disabled once the test is complete. Refer to the `DqAdv212SetBurnoutDetectCurrent() API` for more information.

**Note:**

If the user specifies a short timeout delay, this function can time out when called for the first time because it is executed as a pending command and layer programming takes up to 10 ms.

Once this function is called, the layer continuously acquires data and every call to the function returns the latest acquired data. If one would like to cancel ongoing sampling, call the same function with `0xffffffff` as the first channel number in the channel list.

### *4.8.2 DqAdv212SetBurnoutDetectCurrent*

**Syntax:**

```
int DqAdv212SetBurnoutDetectCurrent(int hd, int devn, uint32
chan_mask, uint32 burnout)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 chan_mask` | bits 0..11 correspond to channels 0..11 0 = no change to burnout detect current, 1 = set burnout detect current per value in `burnout` |
| `uint32 burnout` | One of the #defined constants, see below. |

**Output:**

None**.**

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-212 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets the burnout detect current for the channels specified in `chan_mask.`

```
#define DQ_AI212_BURNOUT_SRC_OFF    (0x00)      // burnout current source off
#define DQ_AI212_BURNOUT_SRC_HALFUA (0x40)      // burnout current source 0.5uA
#define DQ_AI212_BURNOUT_SRC_2UA    (0x80)      // burnout current source 2.0uA
#define DQ_AI212_BURNOUT_SRC_10UA   (0xC0)      // burnout current source 10uA
```

The default burnout setting at system power-up is `DQ_AI212_BURNOUT_SRC_OFF` for all channels.

**Note:**

To enable burnout detection, you must first set the burnout on flag for that channel in the read channel list.

```
cl[channel] |= DQ_AI212_BURNOUT_ON_FLAG;
```

Then program it with an initial call to `DqAdv212Read()` with the configured channel list:

```
DqAdv212Read(hd, DEVN, CHANNELS, cl, NULL, NULL)
```

Burnout detection should be used prior to normal operation. The burnout current flows through a 5kΩ resistor as well as the thermocouple. If the thermocouple is open circuit, the AI-212 input will go to the rails while valid channels will have some offset due to the resistor voltage drop. After detecting open circuits, users should turn current source off (`DQ_AI212_BURNOUT_SRC_OFF)` and reset channel list without the burnout flag (`DQ_AI212_BURNOUT_ON_FLAG)` enabled.

### *4.8.3 DqAdv212SetCjcRate*

**Syntax:**

```
int DqAdv212SetCjcRate(int hd, int devn, double *newrate,
double *oldrate)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `double *newrate` | New CJC sensor rate (Hz) |

**Output:**

| | |
|---|---|
| `double *oldrate` | previous CJC sensor rate (NULL if not required) |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-212 |
| `DQ_BAD_PARAMETER` | `*newrate` is NULL, or `newrate` value is less than 0.25 or greater than 1500 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets a new conversion rate for the digital CJC temperature sensors. The default rate is 4 readings per second.

## 4.9 DNA-AI-217, DNA-AI-218, DNA-AI-219 and DNA-AI-228 layers

AI-217, AI-218, AI-219 and AI-228 function applicability table:

| API function: | AI-217 | AI-218 | AI-219 | AI-228 |
|---|---|---|---|---|
| `DqAdv217Read` | ✓ | ✓ | ✓ | ✓ |
| `DqAdv217GetPgaStatus` | ✓ | ✓ | ✓ | ✓ |
| `DqAdv217SetCfgLayer` | ✓ | ✓ | ✓ | ✓ |
| `DqAdv217SetCjcAvg` | ✓ | | | |
| `DqAdv217SetFIR` | ✓ | ✓ | ✓ | ✓ |
| `DqAdv217SetPll` | ✓ | ✓ | | ✓ |
| `DqAdv218ConfigDio` | | ✓ | ✓ | ✓ |
| `DqAdv218WriteDioOut` | | ✓ | ✓ | ✓ |
| `DqAdv218ReadDioIn` | | ✓ | ✓ | ✓ |
| `DqAdv218SetBITMux` | | ✓ | ✓ | ✓ |

The AI-217-804 uses the same function calls as the AI-217.

### 4.9.1 DqAdv217Read

**Syntax:**

```
int DqAdv217Read(int hd, int devn, int CLSize, uint32 *cl,
uint32 *bData, double *fData)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Number of channels |
| `uint32 *cl` | Pointer to channel list with optional timestamp request |
| `uint32 *bData` | Pointer to raw data received from device (`NULL` if not required) |
| `double *fData` | Pointer to store converted voltage data (`NULL` if not required) |

**Output:**

| | |
|---|---|
| `uint32 *bData` | Raw data received from device |
| `double *fData` | Converted voltage data |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-217, AI-218, AI-219 or AI-228 |

| | |
|---|---|
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE` or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

When this function is called for the first time, the firmware stops any ongoing operation on the device specified and reprograms it according to the channel list supplied. The user cannot perform this function call when the layer is involved in any streaming or data mapping operations.

If the user specifies a short timeout delay, this function can time out when called for the first time because it is executed as a pending command and layer programming takes up to 10 ms.

Once this function is called, the layer continuously acquires data and every call to the function returns the latest acquired data.

If one would like to cancel ongoing sampling, call the same function with `0xffffffff` as a channel number.

## *4.9.2 DqAdv217GetPgaStatus*

**Syntax:**

```
int DqAdv217GetPgaStatus(int hd, int devn, uint32* errchan, uint32*
newdata, uint32* pgadata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `uint32* errchan` | Bit field that indicates which PGA channels have errors |
| `uint32* newdata` | Bit field that indicates which PGA channels have new data since the last read by this function |
| `uint32* pgadata` | Array of 16 PGA status readings. See below for description |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-217, AI-218, AI-219 or AI-228 |

| | |
|---|---|
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads the PGA status of the AI-217/218/219/228.

Use the following defines to identify errors:

```
DQ_AI217_PGAERR_CHKERR  (1UL<<7) // checksum error in SPI; only active if
checksum enabled
DQ_AI217_PGAERR_IARERR  (1UL<<6) // Input Amplifier saturated to supply rail
DQ_AI217_PGAERR_BUFA    (1UL<<5) // Buffer active indication
DQ_AI217_PGAERR_ICAERR  (1UL<<4) // input clamp conduction error
DQ_AI217_PGAERR_ERRFLAG (1UL<<3) // Error flag
DQ_AI217_PGAERR_OUTERR  (1UL<<2) // Output Amplifier clipping or over-current
DQ_AI217_PGAERR_GAINERR (1UL<<1) // Gain network overload
DQ_AI217_PGAERR_IOVERR  (1UL<<0) // Input over-voltage error
```

Updates will occur 10 times per second. Use the `newdata` variable to qualify the data if reads are made faster than this rate.

### *4.9.3 DqAdv217SetCfgLayer*

**Syntax:**

```
int DqAdv217SetCfgLayer(int hd, int devn, int channels, int function,
int data)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channels` | Bitwise mask that determines which AI channels are to be changed |
| `int function` | Constant indicating the configuration type that will be performed |
| `int data` | Configuration data |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by devn does not exist or is not an AI-217, AI-218, AI-219 or AI-228 |
| `DQ_BAD_PARAMETER` | One of the parameters is set to an illegal value |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function enables advanced configuration of the device.

`<function>` parameter:

- `DQ_AI217_SET_CFG_LAYER_PGA` - access the wire disconnect reporting feature or access the extended error reporting features of the PGA280 in the layer's front-end.

- `DQ_AI217_SET_CFG_LAYER_ADC` – set the clocking and decimation algorithm for the ADC.

`<channels>` parameter is a bit mask that determines which channels will be configured:

- '1' in a bit position will enable the corresponding channel for update. (e.g. Bit(1L<<0) for AI channel 0 ... Bit(1L<<15) for AI channel 15).
- Use the defined constant `DQ_AI217_SETCFG_ALL_CHAN` to update all channels.

`channels` programming applies to the `DQ_AI217_SET_CFG_LAYER_PGA` function only. When the `DQ_AI217_SET_CFG_LAYER_ADC` function is selected the `channels` parameter is ignored

`<data>` is the data that used for configuration:

- When `function` is `DQ_AI217_SET_CFG_LAYER_PGA`, `data` is a 16-bit value that will be used to configure the TI PGA280 IC's at the front end of each of the AI channels.
  This value combines chip addressing, register selection and register contents.
  For example, the value 0x470C is used to set wire break detect current sources ON. The '4' is the chip addressing, '7' is the number of the PGA register that will be written, and '0C' is the data that will be put into register 7.
- When `function` is `DQ_AI217_SET_CFG_LAYER_ADC`, `data` is either

```
#define DQ_AI217_SET_ADC_DEFAULT // set legacy sample rate calculation and
                                 //     configuration
or
#define DQ_AI217_SET_ADC_ENH     // set enhanced performance at lower sample rates
```

### *4.9.4 DqAdv217SetCjcAvg*

**Syntax:**

```
int DqAdv217SetCjcAvg(int hd, int devn, int factor)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int factor` | Value to select the number of averaged CJC readings. |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-217 |
| `DQ_BAD_PARAMETER` | Factor is not in the range of -1..0x14 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function can be used to override or restore the default averaging factor used by the CJC channel.

| Factor value setting | # of CJC readings averaged |
|---|---|
| -1 | Use default factor value |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |
| 5 | 32 |
| 6 | 64 |
| 7 | 128 |
| 8 | 256 |
| 9 | 512 |
| 0xa | 1024 |
| 0xb | 2048 |
| 0xc | 4096 |

| | |
|---|---|
| 0xd | 8192 |
| 0xe | 16384 |
| 0xf | 32768 |
| 0x10 | 65536 |
| 0x11 | 131072 |
| 0x12 | 262144 |
| 0x13 | 524288 |
| 0x14 | 1048576 |

The default factor value for the AI-217 is automatically set when the factor value setting is -1. This automatically sets factor value varies depending on the sample rate that is currently set. The default averaging factors are shown in the following table.

| When Sample rate less than or equal to: | And sample rate is greater than: | | Default factor value | # of CJC readings averaged |
|---|---|---|---|---|
| 120,000 | 60,000 | | 0 | 1 |
| 60,000 | 30,000 | | 1 | 2 |
| 30,000 | 15,000 | | 2 | 4 |
| 15,000 | 7,500 | | 3 | 8 |
| 7,500 | 3,750 | | 4 | 16 |
| 3,750 | 1,875 | | 5 | 32 |
| 1,875 | 937.5 | | 6 | 64 |
| 937.5 | 468.75 | | 7 | 128 |
| 468.75 | 234.375 | | 8 | 256 |
| 234.375 | 0 | | 9 | 512 |

## *4.9.5 DqAdv217SetFIR*

**Syntax:**

```
int DqAdv217SetFIR(int hd, int devn, int bank, int action,
int decrat, int tapsize, int* filter_total, double* data)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int bank` | Bit field to select group of 4 channels, see below |
| `int action` | Bit field to select which FIR setting functions to perform, see below |
| `int decrat` | Desired decimation ratio (`NULL` if not required), see below |
| `int tapsize` | Number of taps in filter, length of the following `*data` (`NULL` if not required), or new FIR index if `AI217_FIR_SET_INDEX` is used as the `action` value |
| `double *data` | Pointer to filter taps data (`NULL` if not required), see below |

**Output:**

| | |
|---|---|
| `int *filter_total` | Total of filter coefficients after conversion to integer. Expected returned value is 8388608, see below |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-217, AI-218, AI-219 or AI-228 |
| `DQ_BAD_PARAMETER` | No channel specified, decimation ratio is illegal value, tapsize is illegal value or *data is NULL |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function can be used to perform FIR configuration and control functions for the AI-217 when the user desires to specify his own filter coefficients and decimation rates. In the normal case, these configuration settings are set automatically by the `DqAcbInitOps()` function. Use these settings to override standard behavior for special applications. The user may configure as many parameters as he chooses. Any parameter that is not enabled by a `<bank>` bit will retain its present value.

`<bank>` parameter indicates which channels will get their FIR configuration changed. The AI-217 series has 4 quad FIR units and the others have 4 dual FIR units. In all cases each ADC channel has its own FIR data storage. Each FIR unit shares the same coefficients. The following flags are defined for the AI-217:

```
#define AI217_SEL_QFIR_A    (0x01) //select fir for channels 0-3
#define AI217_SEL_QFIR_B    (0x02) //select fir for channels 4-7
#define AI217_SEL_QFIR_C    (0x04) //select fir for channels 8-11
#define AI217_SEL_QFIR_D    (0x08) //select fir for channels 12-15
```

The following are for the AI-218, AI-219 and AI-228:

```
#define AI218_SEL_DFIR_A    (0x01) //select fir for channels 0,1
#define AI218_SEL_DFIR_B    (0x02) //select fir for channels 2,3
#define AI218_SEL_DFIR_C    (0x04) //select fir for channels 4,5
#define AI218_SEL_DFIR_D    (0x08) //select fir for channels 6,7

#define AI217_SEL_QFIR_ALL  (0x0f) //select all channels
```

More than one bank may be specified by ORing multiple bank flags together.

`<action>` flag bits select which parameters will be written. The following flags are defined:

```
#define DQ_AI217_FIR_SET_INDEX      (0x80)  // override the automatic selection
                                            // of the FIR index
#define AI217_FIR_SET_DEFAULT         (0x8) //set and enable the default filter
#define AI217_FIR_COEFF_LOAD          (0x4) // load taps and coefficients
#define AI217_FIR_SET_DECIMATION_RATE (0x2) // set decimation rate
#define AI217_FIR_ENABLE              (0x1) // enable fir filter
#define AI217_FIR_DISABLE             (0x0) // disable fir filter
```

`AI217_FIR_SET_DEFAULT` flag bit is used to easily set all of the FIR configuration values to their default state. Before setting up a custom configuration, it is recommended to first set all channels to their default state by setting `<bank>` to `AI217_SEL_QFIR_ALL`, setting <`action`> to `AI217_FIR_SET_DEFAULT` and calling `DqAdv217SetFIR().` When the `AI217_FIR_SET_DEFAULT` flag is set, all the other `<action>` parameter bits are ignored.

`AI217_FIR_ENABLE` and `AI217_FIR_DISABLE` defines are used to enable or bypass the FIR filter when a custom FIR filter has been loaded or a user defined decimation rate is used.
When standard default settings are used, the system will automatically load and enable the FIR filter. No additional enabling is required. Loading a zero length set of FIR coefficients will also bypass the FIR filter.

`AI217_FIR_SET_INDEX` flag bit as the `<action>` parameter allows the user to override the automatic selection of the default FIR filter. The `AI217_FIR_SET_INDEX` bit is never used in conjunction with any other `action` bits. The present enable/disable status will be unchanged by setting the index.
This <`action`> parameter gives the user some control over the cutoff frequency of the FIR filter without having to go to the effort of designing and loading custom filter coefficients. Normally the index is calculated and set automatically by the driver when the sample rate is set.

| default FIR index | User sample rate |
|---|---|
| 0 | 120kHz – 60.01kHz |
| 1 | 60kHz – 30.01kHz |
| 2 | 30kHz – 15.01kHz |

| | |
|---|---|
| 3 | 15kHz – 7.51kHz |
| 4 | 7.5kHz – 3751Hz |
| 5 | 3750 - 1876 |
| 6 | 1875 – 937.6 |
| 7 | 937.5 - 469 |
| 8 | 468.75 – 234.4 |
| 9 | 234.375 - 1 |

For example: you are sampling the data at 10 kHz but you find the low-pass cutoff frequency of the default filter to be too high. From the chart we see that the default index for 10 kHz is *3*. As the indices increase the cutoff frequencies get lower which makes the next lower setting be an index of 4. The new index value is passed in by reusing the `<tapsize>` parameter. The normal function of the `tapsize` parameter is not available when the index is being set by using `AI217_FIR_SET_INDEX`. To restore the default filter selection either set an index value of -1 or power cycle the IOM.

`<decrat>` parameter sets the decimation rate. Zero = no decimation. Setting a value of 1 discards one reading for every one kept, three discards three readings for every one kept, etc. When the `AI217_FIR_SET_DECIMATION_RATE` flag bit is set in `action` parameter, The value in `decrat` is set.
The default value is automatically determined by the chosen sampling rate and should not need to be set by the user. If the decimation value is changed, it must be set to the same value in all 4 banks. If you call this function to only change the decimation rate, remember to include the `AI217_FIR_ENABLE` flag with the `AI217_FIR_SET_DECIMATION_RATE` flag. Changing the decimation rate will change the output data rate. Decimation is applied after the FIR filter just before it is sent to the input data FIFO.

`<tapsize>` and `<*data>` parameters are used to load user defined filter coefficients to the selected FIR filters. When the `DQ_AI217_FIR_COEFF_LOAD` flag bit is set in the `action` parameter, the `tapsize` and `*data` values are set. `tapsize` is the number of coefficients expected in the array of double precision coefficients pointed to by `*data.` In practice, for the AI-217-1, AI-218-1 and AI-228-300, the ADC sampling and the FIR filtering always run at a rate that is between 60kHz and 120kHz.
The AI-217-803 will sample at a rate of 30 kHz or less and the FIR will run at the same rate as the sampling. When configuring the coefficients, calculate their value based upon the appropriate run rate.

| Model | Max allowable taps |
|---|---|
| AI-217-1 | 128 |
| AI-217-803 | 512 |
| AI-218-1 | 256 |
| AI-219-1 less than or equal to 120K s/s | 256 |
| AI-219-1 greater than 120K s/s | 128 |
| AI-228-300 | 256 |

The coefficients should be doubles in the [-1.0…1.0] range. The sum of all the coefficients should equal 1.0 or very near to 1.0 depending on the value of the returned `<filter_total>`. In order to maintain accurate calibration, the floating point filter coefficients should be scaled so that the returned `<filter_total>` value is exactly 8388608. Divergence from this optimum number happens because internally to the logic of the layer the FIR is implemented with fixed point numbers. The conversion from the double precision coefficients to the internal fixed point numbers introduces small rounding errors that accumulate.

The coefficient array is assumed to be symmetrical. This means the first coefficient will be the same as the last coefficient, the second will be the same as the next to last, etc. Because of this symmetry and the size limitation of the Ethernet packet, only the first half of the coefficients is actually sent over the Ethernet to the IOM. The IOM reconstructs the missing upper half.

### *4.9.6 DqAdv217SetPll*

**Syntax:**

```
int DqAdv217SetPll(int hd, double samplerate, double* sr_actual,
int* dec_fact, int line)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `double samplerate` | Desired sampling rate |
| `int line` | Sync line to assign signal |

**Output:**

| | |
|---|---|
| `double* sr_actual` | Actual sample rate |
| `int *dec_fact` | Required decimation factor for AI-217/218/228 layer |

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER` | requested sample rate is too high or too low |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

The PLL clock circuit on the CPU layer is programmed and routed by this function.
The PLL may be routed by way of the sync[1] or sync[3] lines by using the `DQ_EXT_SYNC1` or `DQ_EXT_SYNC3` constants for the ‘line’ parameter. The `DQ_EXT_IMMEDIATE` flag should be bitwise ORed with the `line` parameter.
When using ACB mode, `DqAdv217SetPll()`should be called before the `DqAcbInitOps()`function so that the correct sync[x] line will be connected by `DqAcbInitOps()`.

The AI-217 uses clock rates that are higher than the sampling rate (8X) and also uses decimation to reduce the effective sampling rate. This function will calculate the correct clock and decimation factor for the AI-217. If the sampling rate that is setup by this function is the same as the sample rate that is given to `DqAcbInitOps()`, then the correct decimation factor will be applied automatically. Otherwise, the decimation factor must be applied using the `DqAdv217SetFIR()` function after the call to `DqAcbInitOps()`.

The `Config` word used by the second parameter of `DqAcbInitOps()` must enable the PLL clock by specifying the `DQ_LN_CVCKSRC0` constant. For example:

```
       #define CFG217         (DQ_LN_ENABLED \
                              |DQ_LN_ACTIVE \
                              |DQ_LN_GETRAW \
                              |DQ_LN_IRQEN \
                              |DQ_LN_CVCKSRC0 \
                              |DQ_LN_STREAMING \
                              |DQ_FIFO_MODEFIFO)
```

**Note:**

Using `DqCmdSetClock()` is the preferred way to set the AI-217/218/228 sample rates. The AI-217/218/228 layers have an on layer PLL that is automatically setup and used by `DqCmdSetClock()`.

This `DqAdv217SetPll()` function is intended to be used only when multiple AI-217/218/228 layers are used and it is desired to have all of them clocked by the same source.

For the AI-217-803 only, you must use the `DqAdvRoutePll()` function for multiple layer clocking and set the frequency parameter to the desired sample rate multiplied by `DQ_AI217_ADC_CLOCK_FACTOR(8)`.

For the AI-219, use the `DqAdvRoutePll()` function.

### *4.9.7 DqAdv218ConfigDio*

**Syntax:**

```
int DqAdv218ConfigDio(int hd, int devn, int dir_bits)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int dir_bits` | Bits 0..7 correspond to channels 0..7      0 = input, 1 = output |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-218, AI-219 or AI-228 |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets DIO direction of all channels.

### *4.9.8 DqAdv218ReadDioIn*

**Syntax:**

```
int DqAdv218ReadDioIn(int hd, int devn, uint32* din)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `uint32* din` | Received data |

**Return:**

| | |
|---|---|
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-218, AI-219 or AI-228 |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Reads all 8 digital inputs.

### *4.9.9 DqAdv218SetBITMux*

**Syntax:**

```
int DqAdv218SetBITMux(int hd, int devn, int chan_mask, int mux_pos)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chan_mask` | bits 0..7 correspond to channels 0..7 0 = no change to multiplexer, 1 = set multiplexer per value in `mux_pos` |
| `int mux_pos` | Multiplexer position, see below. |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-218, AI-219 or AI-228 |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error - usually caused by trying to run BIT on an older version of the layer that does not support BIT. |

**Description:**

Sets the built-in-test (BIT) multiplexers. BIT is supported on PCB assemblies starting with revision 3.

`<mux_pos> -` select the multiplexer position, use one of the following defined constants:

| | |
|---|---|
| `#define DQ_AI218_BIT_OFF:` | Select normal operation, BIT testing turned OFF. |
| `#define DQ_AI218_BIT_5V_IN:` | Select input #1 with self-test 5V applied. |
| `#define DQ_AI218_BIT_ALT_5V:` | Select input #2 with self-test 5V applied. |

The default power-up state is `DQ_AI218_BIT_OFF.`

The AI-218 and AI-219 will read approximately +5 V (using `DqAdv217Read()` for example) when either the `DQ_AI218_BIT_5V_IN` or `DQ_AI218_BIT_ALT_5V` setting is used.
The AI-228-300 will read approx. +150 V.

### 4.9.10 DqAdv218WriteDioOut

**Syntax:**

```
int DqAdv218WriteDioOut(int hd, int devn, int channel, uint32 dout,
uint32*last_dout)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | Channel number (0 - 7) |
| `int dout` | New value to dout pin |

**Output:**

| | |
|---|---|
| `uint32* last_dout` | Last dout value |

**Return:**

| | |
|---|---|
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-218, AI-219 or AI-228 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Writes a bit to Dout port addressed by `channel`, returns old value. Port must be configured for output using `DqAdv218ConfigDio()`.

Channel 8 may be used to write the same value to all channels. In this case only the last Dout value from channel 0 is returned.

## 4.10 DNA-AI-222 layer

### 4.10.1 DqAdv222Config

**Syntax:**

```
int DqAdv222SetConfig (int hd, int devn, uint32 chan_mask,
uint32 mode, double lead_res)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 chan_mask` | Bit mask: "1" in the channel bit position enables configuration modification for that channel. Any number of channels may be updated at a time. |
| `uint32 mode` | #define value starting with DQ_AI222_RTD_ see below. Sets wiring scheme and resistance range |
| `double lead_res` | User specified lead resistance (for 2-wire measurements only). This value is multiplied by 2 and subtracted from the measured value. |

**Output:**

`<none>`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-222 |
| `DQ_BAD_PARAMETER` | `mode` is not one of the #defined DQ_AI222_RTD_* constants. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

`<uint32 mode>` Sets up the AI-222 layer to measure resistance.

The following DQ_AI222_RTD_ constants specify the wiring scheme and resistance range. Internal to the driver software, these values are used to select the voltage reference, multiplexer settings and gain for the mode selected.

#define DQ_AI222_RTD_2_WIRE_5K
#define DQ_AI222_RTD_3_WIRE_5K
#define DQ_AI222_RTD_4_WIRE_5K
#define DQ_AI222_RTD_2_WIRE_1_25K
#define DQ_AI222_RTD_3_WIRE_1_25K
#define DQ_AI222_RTD_4_WIRE_1_25K
#define DQ_AI222_RTD_2_WIRE_312

#define DQ_AI222_RTD_3_WIRE_312
#define DQ_AI222_RTD_4_WIRE_312
#define DQ_AI222_RTD_2_WIRE_156
#define DQ_AI222_RTD_3_WIRE_156
#define DQ_AI222_RTD_4_WIRE_156
#define DQ_AI222_RTD_2_WIRE_40K
#define DQ_AI222_RTD_3_WIRE_40K
#define DQ_AI222_RTD_4_WIRE_40K

`<double lead_res>` This lead resistance is multiplied by 2 and subtracted from the measured resistance value in 2 wire modes only.

### 4.10.2 DqAdv222Read

**Syntax:**

```
int DqAdv222Read(int hd, int devn, int CLSize, uint32 *cl,
uint32 *bData, double *fData)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | number of entries in the channel list |
| `uint32 *cl` | pointer to channel list with optional timestamp request |
| `uint32 *bData` | pointer to raw data received from device (`NULL` if not required) |
| `double *fData` | pointer to store converted resistance data (`NULL` if not required) |

**Output:**

| | |
|---|---|
| `uint32 *bData` | raw data received from device |
| `double *fData` | converted resistance data |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-222 |
| `DQ_BAD_PARAMETER` | `CLSize` isn't between 1 and `DQ_AI222_CL_MAX_SIZE`, or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

The user cannot perform this function call when the layer is involved in any streaming or data mapping operations. When this function is called for the first time, the firmware stops any ongoing operation on the device specified and reprograms it according to the channel list supplied. Channel list entries contain only the channel number (or the optional timestamp command as the last channel in the list) per the following format:

```
        Bits 7 - 0
    <channel number>
```

If the user specifies a short timeout delay, this function can time out when called for the first time because it is executed as a pending command and layer programming takes up to 100ms.

Once this function is called, the layer continuously acquires data and every call to the function returns the latest acquired data.

If one would like to cancel ongoing sampling, call the same function with `0xffffffff` as the first channel number in the channel list.

The user must setup the layer for reading by calling the DqAdv222Config() function before calling DqAdv222Read()

## 4.11 DNA-AI-224 layer

### 4.11.1 DqAdv224Read

**Syntax:**

```
int DqAdv224Read(int hd, int devn, int CLSize, uint32 *cl,
uint32 *bData, double *fData)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Number of channels in channel list, 1 to 4 |
| `uint32 *cl` | Pointer to the channel list |
| `uint32 *bData` | Pointer to raw data received from the device |
| `double *fData` | Pointer to store converted voltage data (`NULL` if not required) |

**Output:**

| | |
|---|---|
| `uint32 *cl` | Channel list |
| `uint32 *bData` | Received binary data |
| `double *fData` | Received voltage data |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-224 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, `bData` is NULL, or a channel number in `cl` is invalid |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

When this function is called for the first time, the firmware stops any ongoing operation on the device specified and reprograms it in accordance with the channel list supplied. Thus, the user cannot call this function when the layer is involved in any streaming or data mapping operations.

If the user specifies a timeout delay that is too short, this function can time out when called for the first time because it is executed as a pending command and layer programming takes up to 10 ms.

Once this function is called, the layer continuously acquires data and every next call function returns the latest acquired data.

If you would like to cancel ongoing sampling, call the same function with 0xffffffff as the first channel element.

The channel list (`cl`) is an array of uint32 elements with a length (`CLSize`) from 1 to 4. Each element of the channel list contains a bit-packed word that defines the channel number, the front-end multiplexer configuration and the gain. The following examples show the string of helper macros that are used to assemble the mux setting, channel number and gain setting.

Example 1. First element (0) of the channel list specifies channel 2 with the front-end mux set to read the S+ and S- inputs with a gain of 10.

```
 cl[0] = DQ_LNCL_CHANGAIN (DQ_AI224_SET_CHAN(DQ_AI224_MUX_SS,2),
DQ_AI224_GAIN_10);
```

Example 2. Second element (1) of the channel list specifies channel 3 with the front-end mux set to read the difference between the bridge completion voltage and the S- input with a gain of 1.

```
cl[1] = DQ_LNCL_CHANGAIN (DQ_AI224_SET_CHAN(DQ_AI224_MUX_CS,3),
DQ_AI224_GAIN_1);
```

You cannot have a channel list that has more than one multiplexer setting per channel.

When using these macros, the valid channel numbers are numbered from 0 to 3. The following #defines are used to select the front-end multiplexers:

```
DQ_AI224_MUX_SS       // S+ to S-
DQ_AI224_MUX_CS       // Bridge comp - S-
DQ_AI224_MUX_EXCP     // P+ to S-
DQ_AI224_MUX_PPS      // P+ to PS+
DQ_AI224_MUX_NULL     // GND only for nulling
DQ_AI224_MUX_PS       // PS+ to PS-
DQ_AI224_MUX_EXCN     // P- to S-
DQ_AI224_MUX_5K       // Vdrop on 5k prec. resistor
```

The following #defines are used to select gain:

```
DQ_AI224_GAIN_1
DQ_AI224_GAIN_2
DQ_AI224_GAIN_4
DQ_AI224_GAIN_8
DQ_AI224_GAIN_16
DQ_AI224_GAIN_32
DQ_AI224_GAIN_64
DQ_AI224_GAIN_128
DQ_AI224_GAIN_256
```

**Note:**

Channels are numbered from 0 to 3.

### *4.11.2 DqAdv224SetAveraging*

**Syntax:**

```
int DqAdv224SetAveraging(int hd, int devn, uint32 channel,
uint32 factor)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number |
| `uint32 channel` | Bit field of which channels to set |
| `uint32 factor` | Desired averaging factor |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-224 |
| `DQ_BAD_PARAMETER` | channel or factor have values that are illegal |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets the data averaging factor for any channel. The averaging factor may be set to any value from 0 to 20. 0=1, 1=2, 2=4, 3=8, 4=16, 5=32, 6=64, 7=128 ... 20=1024576.

Please note that the rate at which the data changes decreases as the averaging goes up. With an average factor of 20, the data will change only once every two seconds (approximately).
Averaged data may be read by performing a `DqAdv224Read` using channel numbers 4 thru 7 instead of the usual 0 thru 3.

`<channel>` parameter indicates which channels will get their averaging factor changed.
The following flags are defined:

```
#define AI224_SEL_CHAN_0        (0x01)
#define AI224_SEL_CHAN_1        (0x02)
#define AI224_SEL_CHAN_2        (0x04)
#define AI224_SEL_CHAN_3        (0x08)
#define AI224_SEL_CHAN_ALL      (0x0f)
```

More than one channel may be specified by oring multiple channel flags together.

**Note:**

### *4.11.3 DqAdv224SetBridgeCompletion*

**Syntax:**

```
int DqAdv224SetBridgeCompletion (int hd, int devn, uint32 channel,
double ref_level)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number |
| `uint32 channel` | Which channel to set |
| `double ref_level` | Desired reference level in volts, +/- 10V max. |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-224 |
| `DQ_BAD_PARAMETER` | `channel` is not in range of 0-3, or `ref_level` is greater than 10.0 or less than -10.0. |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets the internal bridge completion voltage. At run-time use the following steps to set the bridge completion voltage.

1) Set bridge completion voltage to 0V
2) Measure difference between Bridge Completion voltage and S- by doing a `DqAdv224Read` using `DQ_AI224_MUX_CS` in the channel list at a gain setting of 1.
3) Measure the voltage. Averaging a large number of samples will increase accuracy and help to cancel any motion sensed by the strain gauge.
4) Set the bridge completion voltage to the voltage measured in step 3. Note that the API will only allow bridge completion voltage changes in 1 millivolt steps.

The bridge should be ready to use. If desired, the bridge completion voltage may be adjusted by measuring and averaging again, this time measuring to see any difference between the measured and desired value. For example, if the reading is now 3 mV higher than desired, decrease the bridge completion voltage by 3 mV.

**Note:**

### *4.11.4 DqAdv224SetExcitation*

**Syntax:**

```
int DqAdv224SetExcitation(int hd, int devn, uint32 channel,
uint32 config, double exc_rate, double exc_level_hi,
double  exc_level_lo, double* calc_rate)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number |
| `uint32 channel` | Channel to program, 0-3 |
| `uint32 config` | Configuration word |
| `double exc_rate` | Desired sinewave rate, Hz (AC excitation only) |
| `double exc_level_hi` | Desired high excitation level in Volts, +/-10V max.<br>AC peak-to-peak sinewave voltage is twice this value |
| `double exc_level_lo` | Desired low excitation level in Volts, +/-10V max.<br>AC peak-to-peak sinewave voltage is twice this value |

**Output:**

| | |
|---|---|
| `double *calc_rate` | Actual AC sinewave rate |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-224 |
| `DQ_BAD_PARAMETER` | illegal `exc_level, config` or `channel` value. |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets excitation voltage for any channel.

Maximum voltage swing on either excitation output is +/-10V. This applies to both AC and DC settings. When AC excitation is selected, all channels with AC excitation will be set to the same excitation frequency.

`<config>` must be set to either of the following 2 defines:

`DQ_AI224_SET_DC_EXC`

`DQ_AI224_SET_AC_EXC` (not implemented at this time)

**Note:**

This function must be called before starting acquisition or reading channels to set up a proper excitation voltage source.

### *4.11.5 DqAdv224SetFIR*

**Syntax:**

```
int DqAdv224SetFIR(int hd, int devn, int ch_mask , int stage,
int action, int decrat, int tapsize, int *filter_total, double* data)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int ch_mask` | Bit field to select single or multiple channels |
| `int stage` | Which filter to set: `DQ_AI224_SELECT_FIR0` or `DQ_AI224_SELECT_FIR1` |
| `int action` | What operation to perform |
| `int decrat` | Filter decimation ratio |
| `int tapsize` | Number of taps in FIR post filter, length of the following *data (`NULL` if not required), see below |
| `double *data` | Pointer to filter taps data (`NULL` if not required), see below |

**Output:**

| | |
|---|---|
| `int *filter_total` | Total of filter coefficients after conversion to integer. Expected returned value is `DQ_AI224_FIR_TOTAL` (32768), see below |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-224 |
| `DQ_BAD_PARAMETER` | No channel specified in `ch_mask`, decimation ratio is illegal value, tapsize is illegal value or *data is NULL |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

`<ch_mask>` parameter indicates which channels will get their FIR configuration changed. The following flags are defined:

```
#define DQ_AI224_SEL_CHAN_0        (0x01)
#define DQ_AI224_SEL_CHAN_1        (0x02)
#define DQ_AI224_SEL_CHAN_2        (0x04)
#define DQ_AI224_SEL_CHAN_3        (0x08)
#define DQ_AI224_SEL_CHAN_ALL      (0x0f)
```

More than one channel may be specified by bitwise ORing multiple channel flags together.

`<stage>` parameter selects which FIR filter will be acted upon. Use 0 for FIR0 and 1 for FIR1. ADC data passes through FIR0 then through FIR1. The `DqAdv224SetFIR` function must be called once for each stage to fully program the FIR post filters.

`<action>` parameter consists of flag bits that select which parameters will be written.
The following flags are defined:

```
  #define DQ_AI224_FIR_DISABLE        (0x0) // disable fir filter
  #define DQ_AI224_FIR_ENABLE         (0x1) // enable fir filter
  #define DQ_AI224_FIR_SET_DECRATE    (0x2) // set decimation rate
  #define DQ_AI224_FIR_COEFF_LOAD     (0x4) // load taps and coefficients
  #define DQ_AI224_FIR_SET_DEFAULT    (0x8) //set and enable default filter
  #define DQ_AI224_FIR_SET_SKIP_COUNT (0x20)// set skip count by user for
                                                special applications
```

The `DQ_AI224_FIR_SET_DEFAULT` flag bit is used to easily set all of the FIR configuration values to their default state. When the `DQ_AI224_FIR_SET_DEFAULT` flag is set, all of the other <`action`> parameter bits are ignored.

The `DQ_AI224_FIR_ENABLE` and `DQ_AI224_FIR_DISABLE` defines are used to enable or bypass the FIR filter when a custom FIR post filter has been loaded or the decimation rate has been changed. Remember to use both the `DQ_AI224_FIR_SET_DECRATE` and the `DQ_AI224_FIR_ENABLE` flags when changing decimation rates.

The `DQ_AI224_FIR_SET_SKIP_COUNT` define can be used to override the automatic group delay settings. The setting is sent in the `<tapsize>` parameter. Bits 31:16 carry the setting for FIR0 and bits 15:0 carry the setting for FIR1. The `DQ_AI224_FIR_COEFF_LOAD` flag may not be used at the same time as the `DQ_AI224_FIR_SET_SKIP_COUNT` flag.

`<decrat>` parameter sets the decimation rate. Zero = no decimation. Setting a value of 1 discards one reading for every one kept, two discards two readings for every one kept, etc. When the `DQ_AI224_FIR_SET_DECRATE` flag bit is set in the <`action`> parameter, the value in <`decrat`> is set.

`<tapsize> and <*data>` parameters are used to load user defined filter coefficients to the selected FIR post filters. When the `DQ_AI224_FIR_COEFF_LOAD` flag bit is set in the <`action`> parameter, the <`tapsize`> and <`*data`> values are set. `<tapsize>` is the number of coefficients expected in the array of double precision coefficients pointed to by `<*data>.` The coefficients should be in the [-1.0…1.0] range. The sum of all the coefficients should equal 1.0 or very close to 1.0 depending on the value of the returned `<filter_total>`. In order to maintain accurate calibration, the filter coefficients should be scaled so that the returned <`filter_total`> value is `DQ_AI224_FIR_TOTAL` or 32768.

**Note:**

### *4.11.6 DqAdv224SetNullLevel*

**Syntax:**

```
int DqAdv224SetNullLevel(int hd, int devn, uint32 channel,
double level)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number |
| `uint32 channel` | Which channel to set, values 0 thru 3 |
| `double level` | Desired null level in volts, +/-10 V max, use `DQ_AI224_SET_NULLING_OFF` to set nulling OFF |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-224 |
| `DQ_BAD_PARAMETER` | `channel` is not in range 0..3, `level` is not 500.0 or is greater than 10.0 or less than -10.0 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets the nulling voltage and will null at gains up to 40.

Send `DQ_AI224_SET_NULLING_OFF` as the nulling voltage to turn nulling OFF.

**Note:**

### 4.11.7 DqAdv224SetShunt

**Syntax:**

```
int DqAdv224SetShunt(int hd, int devn, uint32 chan_gain,
uint32 config, uint32 position, double* r_shunt)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number |
| `uint32 chan_gain` | Channel number and gain value, see below |
| `uint32 config` | Configuration word, see below |
| `uint32 position` | Position of the digital shunt, 0x0..0xff [5K...205K ±30%] |

**Output:**

| | |
|---|---|
| `double *r_shunt` | Calculated value of shunt potentiometer |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-224 |
| `DQ_BAD_PARAMETER` | `chan_gain` is not in range of 0..3 or gain is too high, `config` is not one of the specified constants, `position` is not in range of 0x0..0xff |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function is used to enable or disable and adjust the setting of the digital shunt potentiometer. The shunt may be inserted across either the P+ to S- or the P- to S- bridge resistance.

`<chan_gain>` contains the channel number [0..3] and is optionally bit-packed with the gain number when the `<r_shunt>` value is not NULL. The `DQ_LNCL_CHANGAIN` macro is provided to pack the channel and gain together. Use one of the following constants:

```
DQ_AI224_GAIN_1
DQ_AI224_GAIN_2
DQ_AI224_GAIN_4
DQ_AI224_GAIN_8
DQ_AI224_GAIN_16
DQ_AI224_GAIN_32
DQ_AI224_GAIN_64
DQ_AI224_GAIN_128
```

For example, to use a gain setting of 4 on channel 0 use:

```
chan_gain = DQ_LNCL_CHANGAIN(0, DQ_AI224_GAIN_4);
```

`<config>` should contain one of the following constants:

```
DQ_AI224_SHUNT_DISABLED
DQ_AI224_SHUNT_A       // across P+ and S-
DQ_AI224_SHUNT_B       // across P- and S
```

**Notes:**

None

## 4.11.8 DqAdv224ShuntCal

**Syntax:**

```
int DqAdv224ShuntCal(int hd, int devn, uint32 channel,
uint32 shunt_set, int cycles, double excitation, double r_shunt,
double* r_actual, double* v_exc, double* v_gage, double* v_shunt)
```

**Command:**

None

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number |
| `uint32 channel` | What channel to measure |
| `uint32 shunt_set` | Type of shunt to apply:<br>`DQ_AI224_SHUNT_A`<br>`DQ_AI224_SHUNT_B` |
| `int cycles` | Number of measurement cycles to average (1..100) |
| `double excitation` | Requested excitation voltage: 2.0 V..9.0 V |
| `double r_shunt` | Requested shunt value: 10k..170k |

**Output:**

| | |
|---|---|
| `double* r_actual` | Actual shunt resistance |
| `double* v_exc` | Actual excitation voltage |
| `double* v_gage` | Actual voltage between S+ and S- |
| `double* v_shunt` | Actual voltage between S+ and S- with shunt applied |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-224 |
| `DQ_BAD_PARAMETER` | one of parameters supplied are bad |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| `DQ_EXECUTION_ERROR` | error while working with shunt cal, most likely connections |
| Other negative values | low level IOM error |

**Description:**

The function performs shunt calibration measurements.

**Notes:**

1. Shunt A is applied between P+ and S-; shunt B between S- and P-.

2. Measurement range of `r_shunt` can be from 10k to 170k (shunt has 5 k 0.1% resistor and 200 k nominal digital potentiometer with good ppm/C but with 30% tolerance). Some layers may be capable of providing shunt calibration with a resistance exceeding 170 k.

3. The `cycles` parameter controls the setting of the averager. Cycles values of less than 30 use an average setting of 256. Cycles values greater than 89 use a setting of 1024. In-between values of cycles use an average setting of 512.

## 4.12 DNA-AI-225 layer

### 4.12.1 DqAdv225Read

**Syntax:**

```
int DqAdv225Read(int hd, int devn, int CLSize, uint32 *cl,
uint32 *RawData, uint32 *uData, double *fData)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | number of channels |
| `uint32 *cl` | pointer to channel list |
| `uint32 *RawData` | pointer to raw data received from device |
| `uint32 *uData` | pointer to store calibrated binary data (`NULL` if not required) |
| `double *fData` | pointer to store calibrated voltage data (`NULL` if not required) |

**Output:**

| | |
|---|---|
| `uint32 *RawData` | raw data received from device |
| `uint32 *uData` | calibrated binary data |
| `double *fData` | calibrated voltage data |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-225 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, `RawData` is NULL, or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function works using an underlying `DqReadAIChannel()`, but converts data using internal knowledge of the input range and calibrates every channel. It uses `DQCMD_IOCTL` with `DQIOCTL_CVTCHNL` under the hood.

When this function is called for the first time, the firmware stops any ongoing operation on the device specified and reprograms it according to the channel list supplied. This function uses a preprogrammed CL update frequency – 13.75Hz. One can reprogram the update frequency by calling `DqCmdSetClk()`after the first call to `DqAdv225Read()`.

Thus, the user cannot perform this function call when the layer is involved in any streaming or data mapping operations.

If the user specifies a short timeout delay, this function can time out when called for the first time because it is executed as a pending command and layer programming takes up to 10ms.

Once this function is called, the layer continuously acquires data and every call to the function returns the latest acquired data.

If one would like to cancel ongoing sampling, call the same function with `0xffffffff` as a channel number.

**Note:**

### 4.12.2 DqAdv225SetRate

**Syntax:**

```
int DqAdv225SetRate(int hd, int devn, float *fCLClk, uint32
*TrigMode)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `float *fCLClk` | desired acquisition rate in Hz |
| `uint32 *TrigMode` | clocking and triggering mode; can be NULL |

**Output:**

| | |
|---|---|
| `float *fCLClk` | actual acquisition rate in Hz |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-225 |
| `DQ_BAD_PARAMETER` | `fCLClk` is NULL or points to 0 value |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets the layer acquisition rate.

The AI-225 supports a fixed rate set: from 5 to 3000Hz.

Possible values for TrigMode are given by these defines:

```
#define DQ_LN_PTRIGEDGE1 (1L<<9) // stop trigger edge MSB
#define DQ_LN_PTRIGEDGE0 (1L<<8) // stop trigger edge:
                                 // 00 - software, 01 - rising, 02 - falling
#define DQ_LN_STRIGEDGE1 (1L<<7) // start trigger edge MSB
#define DQ_LN_STRIGEDGE0 (1L<<6) // start trigger edge:
                                 // 00 - software, 01 - rising, 02 - falling
```

**Note:**

Affects `DqAdv225Read()` function only.

## *4.13 DNA-AI-248 layer*

### *4.13.1 DqAdv248Read*

**Syntax:**

```
int DqAdv248Read(int hd, int devn, int CLSize, uint32 *cl,
uint32 *bData, double *fData)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | number of channels |
| `uint32 *cl` | pointer to channel list |
| `uint32 *bData` | pointer to store calibrated binary data (`NULL` if not required) |
| `double *fData` | pointer to store calibrated voltage data (`NULL` if not required) |

**Output:**

| | |
|---|---|
| `uint32 *bData` | binary data |
| `double *fData` | voltage data |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-248 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE` or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

When this function is called for the first time, the firmware stops any ongoing operation on the device specified and reprograms it according to the channel list supplied. The user cannot perform this function call when the layer is involved in any streaming or data mapping operations.

If the user specifies a short timeout delay, this function can time out when called for the first time because it is executed as a pending command and layer programming takes up to 10ms.

Once this function is called, the layer continuously acquires data and every call to the function returns the latest acquired data.

Available gain settings are:

| | |
|---|---|
| #define DQ_AI248_GAIN_1 | (0) |
| #define DQ_AI248_GAIN_10 | (1) |
| #define DQ_AI248_GAIN_100 | (2) |
| #define DQ_AI248_GAIN_1000 | (3) |

Gain settings are combined with the channel number and placed in the channel list. See macros DQ_LNCL_GAIN() and DQ_LNCL_CHANGAIN() in powerdna.h.

If one would like to cancel ongoing sampling, call this function with `0xffffffff` as the first channel number.

**Note:**

### *4.13.2 DqAdv248SetAutozero*

**Syntax:**

```
int DqAdv248SetAutozero(int hd, int devn, int* onoff)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int* onoff` | autozero on or off, 0=off, 1=on |

**Output:**

| | |
|---|---|
| `int* onoff` | previous state |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-248 |
| `DQ_BAD_PARAMETER` | `onoff` pointer is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets the Autozero function in point-by-point mode to be either ON or OFF.

By default the autozeroing is ON. It is normally never turned OFF.

**Note:**

## *4.14 DNA-AI-254 layer*

DNA-AI-254 layer is an all-digital LVDT/RVDT input and simulator. It has two A/Ds and two D/As per channel, each capable of sampling at the rate up to 330 kHz. Once sampled, all readings from secondary windings of an LVDT/RVDT sensor are processed in the digital domain. The same applies to simulation – all processing occurs in the digital domain and the D/As convert a digital representation of the simulated signal into an analog waveform at the last stage of the processing.

There are six modes of operation supported by this layer. Three major modes are:

1. Input with internal excitation: AI-254 provides the required excitation voltage to an LVDT/RVDT sensor and reads back the output of its secondary windings.
2. Input with external excitation (when the layer can monitor signals on the existing avionics interface). This mode is used when either external excitation is required to excite an LVDT (for example, when the required current exceeds the capabilities of the AI-254 layer), or when an AI-254 is used in the role of a wiretapping device in parallel with the existing LVDT input device.
3. Simulation mode (when the layer is connected directly into the avionics and simulates (or mimics) the LVDT sensor itself). This mode is used when an external device expects to receive output from an LVDT sensor in response to an applied excitation voltage. The AI-254 calculates parameters of the externally applied excitation and generates sinewaves on its outputs in accordance with the requested LVDT position.

These modes of operation have two variations:

1. 5/6 wire connection (in case of 5 wire circuitry, connect common negative to both S1- and S2- lines)
2. 4 wire connection. In this case, you have to make sure that the external excitation signal is properly attenuated and applied to the S2 input of the AI-254.

```
DQ_AI254_MODE_INT_5     0   // Mode0: internal excitation of 5-wire LVDT
DQ_AI254_MODE_INT_4     1   // Mode1: internal excitation of 4-wire LVDT
DQ_AI254_MODE_EXT_5     2   // Mode2: internal excitation of 5-wire LVDT
DQ_AI254_MODE_EXT_4     3   // Mode3: internal excitation of 4-wire LVDT
DQ_AI254_MODE_SIM_5     4   // Mode4: simulation of the outputs of 5-wire LVDT
DQ_AI254_MODE_SIM_4     5   // Mode5: simulation of the outputs of 6-wire LVDT
```

The following table summarizes the required connections between an AI-254 layer and an LVDT:

**Wiring**

| | 4-wires | | 5/6-wires | |
|---|---|---|---|---|
| | AOut | AIn | AOut | AIn |
| Input, internal excitation | P1+ and P1- connected to the primary | S1+ and S1- connected to Vout | P1+ and P1- connected to the primary | S1+ and S1- connected to Va;<br>S2+ and S2- connected to Vb |

| | | | | |
|---|---|---|---|---|
| Input, external excitation | N/C | S1+ and S1- connected to Vout;<br>S2+ and S2- connected to external excitation | N/C | S1+ and S1- connected to Va, S2+ and S2- connected to Vb |
| Simulator, external excitation | P1+ and P1- connected to InA+ and InA- of the device | S1+ and S1- connected to Exc+ and Exc- of the device | P1+ and P1- connected to InA+ and InA- of the device;<br>P2+ and P2- connected to InB+ and InB- of the device; | S1+ and S1- connected to Exc+ and Exc- of the device |

**Mode: Internal excitation, 4/5/6 wire LVDR/RVDT**

```
DQ_AI254_MODE_INT_5: <- 5/6 wires
DQ_AI254_MODE_INT_4: <- 4 wires

exc_rate = 2600.0; <- specified excitation frequency
exc_level = 22.0; <- specified excitation voltage (from the datasheet)
adc_rate = 0; -> returns actual sampling rate
ret = DqAdv254SetExcitation(hd0, DEVN, CHANNEL,
                            DQ_AI254_ENABLE_EXC_A, // A channel only
                            exc_rate, exc_level, &adc_rate);

exc_level = 11.8; <- expected level on the secondary coils (datasheet). This is
required to select proper Se for four-wire configurations.
usr_offset = 0.0; <- additional offset (for in-system calibration, keep 0 if not
needed)
usr_gain = 1.0; <- additional gain (for in-system calibration, keep 1.0 if not
needed)
ret = DqAdv254SetMode(hd0, DEVN, CHANNEL, mode, flags, usr_offset, usr_gain,
(float)exc_rate, (float)exc_level);
```

**Mode: External excitation, 4/5/6 wire LVDR/RVDT**

```
DQ_AI254_MODE_EXT_5: <- 5/6 wires
DQ_AI254_MODE_EXT_4: <- 4 wires

// Measure frequency and level on input B
ret = DqAdv254MeasureWF(hd0, DEVN, CHANNEL_COARSE,
                        &exc_rate, &exc_level, &exc_offs);

exc_level = 11.8; <- expected level on the secondary coils (datasheet). This is
required to select proper Se for four-wire configurations.
usr_offset = 0.0; <- additional offset (for in-system calibration, keep 0 if not
needed)
```

```
usr_gain = 1.0; <- additional gain (for in-system calibration, keep 1.0 if not
needed)
ret = DqAdv254SetMode(hd0, DEVN, CHANNEL, mode, flags, usr_offset, usr_gain,
(float)exc_rate, (float)exc_level);
```

**Mode: Simulation with external excitation, 4/5/6 wire LVDR/RVDT**

```
DQ_AI254_MODE_SIM_5: <- 5/6 wires
DQ_AI254_MODE_SIM_4: <- 4 wires

// Measure frequency and level on input B
ret = DqAdv254MeasureWF(hd0, DEVN, CHANNEL_COARSE,
                        &exc_rate, &exc_level, &exc_offs);

exc_level = 11.8; <- expected level on the secondary coils (datasheet). This is
required to select proper Se for four-wire configurations
usr_offset = 0; <- set to 0
usr_gain = 1.0; <- set to 1
ret = DqAdv254SetMode(hd0, DEVN, CHANNEL_SIM, mode, flags, usr_offset, usr_gain,
exc_rate, exc_level);
```

### 4.14.1 DqAdv254SetMode

**Syntax:**

```
      int DAQLIB DqAdv254SetMode(int hd, int devn, uint32 channel, uint32 mode,
uint32 flags, uint32* meas_pts, double usr_offset, double usr_gain, double
exc_freq, double Se_level)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 channel` | Channel to apply mode to |
| `uint32 mode` | Mode of operation |
| `uint32 flags` | Additional flags, reserved |
| `uint32* meas_pts` | Number of points per period of the generated sine wave (equal to the number of sampling points per period) |
| `double usr_offset` | User-defined offset from -1.0 to +1.0 in (1/0x7fffff) increments to fine-calibrate output data (to trim LVDT/RVDT position) |
| `double usr_gain` | User-defined gain from 0 to 1.0 (to trim LVDT/RVDT gain). Set to 1.0 |
| `double exc_freq` | Expected excitation frequency for external excitation and simulation modes (need not be exact, ignored for internal excitation modes) |
| `double Se_level` | Expected excitation level (Vpp) to use for Sa/Se division in 4-wire modes. For simulation mode, this is an initial excitation level in Volts (2..22V) |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-254 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, or a channel number in `cl` is incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up one of six operating modes supported by an AI-254 layer.

The AI-254 layer has two hardware registers that are required to scale the output of calculations of $(S_a-S_b)/(S_a+S_b)$ for 5/6 wire configuration or $(S_{(a-b)})/S_e$ for 4-wire configuration.
The result of this division is scaled from 0x0 to 0xffffff (24-bit representation) into -1.0 to +1.0 by the `DqCmd254Read()` function.

`<usr_offset>` defines the fine-tuning offset. This offset is set in the range of -1.0 (all Sb, no Sa) to +1.0 (opposite side of LVDT) and can be used to adjust the middle point of the LVDT device. This offset is converted into a binary representation and applied in the hardware to the raw values of conversion.

`<usr_gain>` defines fine-tuning gain. This gain is set in the range from 0 to +1.0 and can be used to adjust the response characteristics of an LVDT device. This gain is converted into a binary representation and applied in the hardware to raw values of calculated position of the LVDT.

`<exc_freq>` specifies the expected frequency to be received on S2 input (four-wire case) or S1 input (simulation case). This parameter can be obtained using `DqAdv254GetWFMeasurements()` and is required to set internal parameters such as the number of points per period and the A/D rate (to be in the acceptable range for simulation mode)

`<Se_level>` defines different things in different modes of operation:

- in 4-wire mode, it defines a constant, Se, which is used to divide the input voltage.
- in simulation mode, it defines initial levels of the simulated output.

**Notes:**

Function returns `<meas_pts>` which is the number of points per period of the generated sinewave. This comes handy when you need to calculate true Vrms of the sinewave.
Vrms = (Sa+Sb)/`<meas_pts>`

### 4.14.2 DqAdv254SetExt

**Syntax:**

```
int DAQLIB DqAdv254SetExt(int hd, int devn, uint32 channel, uint32
mode, uint32 flags, pDQ254SetExt params)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 channel` | Channel to apply mode to |
| `uint32 mode` | Mode of operation |
| `uint32 flags` | Flags defining valid parameters |
| `pDQ254SetExt params` | Parameters |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-254 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function allows setting up additional parameters for AI-254 layer. It is not necessarily to call this function if you would like to rely on default settings.
Three parameters can be changed as defined in the following structure:

```
typedef struct {
    uint32 PositionAvg;       // Set moving average window size: 0=1,...,8=256
    uint32 MinMaxAvg;         // Set min/max window size: 0=1,...,8=256
    uint32 ZeroCrossing;      // Set zero crossing window: 0=1,...,6=64
} DQ254SetExt, *pDQ254SetExt;
```

`<flags>` define valid parameters:
If a flag is set then parameter is valid and used, if not set it is ignored:

```
DQ_AI254_MODE_SETAVG - Set position calculation moving average 0=1, 1=2, ..., 8=256
DQ_AI254_MODE_SETMMAVG - Set min/max moving average 0=1, 1=2, ..., 8=256
DQ_AI254_MODE_SETZEROC - Set zero crossing window 0=1,...,6=64 samples
DQ_AI254_MODE_USE_SX4 - Use Sx averaging instead of Min/Max average for 4-wire scheme
DQ_AI254_MODE_SET_SQ_ALL    -    Set excitation of all channels to squarewave
DQ_AI254_MODE_SET_SINE_ALL  -    Set excitation of all channels to sine wave
```

```
DQ_AI254_MODE_SET_SINE_SQR -    Enable changing of the following sine/square flags:
DQ_AI254_MODE_SET_SQ_CH0   -    Set Ch0 excitation to squarewave (0=sine)
DQ_AI254_MODE_SET_SQ_CH1   -    Set Ch1 excitation to squarewave (0=sine)
DQ_AI254_MODE_SET_SQ_CH2   -    Set Ch2 excitation to squarewave (0=sine)
DQ_AI254_MODE_SET_SQ_CH3   -    Set Ch3 excitation to squarewave (0=sine)
```

Notes:

`DQ_AI254_MODE_USE_SX4` flag does not require any parameters. Instead, it switches averaging for 4-wire scheme from calculating signal amplitude to calculating position.

### *4.14.3 DqAdv254SetExcitation*

**Syntax:**

```
int  DqAdv254SetExcitation(int hd, int devn, uint32 channel, uint32 config,
double exc_rate, double exc_level, double* calc_rate);
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 channel` | Channel to apply mode to |
| `uint32 config` | Configuration: DQ_AI254_ENABLE_EXC_A to enable output P1 and DQ_AI254_ENABLE_EXC_B to enable output P2 |
| `double exc_rate` | Desired excitation rate in Hz |
| `double exc_level` | Desired excitation level, in V |

**Output:**

| | |
|---|---|
| `double* calc_rate` | Actual D/A rate, limited by the abilities of D/A converter, timebase error, and number of points per period. For informational purposes only |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-254 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function is used to set up internal excitation frequency and amplitude in internal excitation modes of operation.

The following constants are used in <config>:

```
#define DQ_AI254_ENABLE_EXC_A   (1L<<0) // enable excitation channel A
#define DQ_AI254_ENABLE_EXC_B   (1L<<1) // ditto B
```

**Notes:**

### 4.14.4 DqAdv254GetWFMeasurements

**Syntax:**

```
      int  DqAdv254GetWFMeasurements(int hd, int devn, uint32 channel,
pWFMEASURE_254 prms, pWFPRM_254 chan_a, pWFPRM_254 chan_b)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 channel` | Channel to use |
| `pWFMEASURE_254 prms` | Measurement parameters |

**Output:**

| | |
|---|---|
| `pWFPRM_254 chan_a` | Results of measurements for channel S1 |
| `pWFPRM_254 chan_b` | Results of measurements for channel S2 |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-254 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function measures parameters of the waveform presented on the inputs S1 and S2 of the <channel>. The waveform parameters measurements are required for setting up proper excitation frequency in DqAdv254SetMode() for modes with external excitation and simulation.

The function requires input that specifies various parameters of measurement:

```
typedef struct {
       uint32 changain;        // channel and gain
       uint32 zero;            // zero level, 0 == default (0x8000)
       uint32 clock;           // A/D rate, 0 == maximum rate
       uint32 uzb;             // TRUE == use ZC on channel S2
       uint32 uaz;             // TRUE == auto-zero
       uint32 s_e;             // if divider is set (not-zero), use Sa/Se
       uint32 period_ms;       // maximum expected waveform period
} WFMEASURE_254, *pWFMEASURE_254;
```

Not all parameters are required to be supplied. For 5/6-wire devices, you may set up the <changain> parameter only (to the same value as in <channel>). If the excitation voltage is applied to channel S2, or channel S1 has a low amplitude waveform, set <uzb> to TRUE to use channel S2 for detecting zero-crossing events. Use <s_e> to set up a constant for $S_{a-b}/S_e$ calculation, keeping it above $S_a$ on S1. The rest of the parameters can be set to zero.

Auto-zero is a function of the hardware, which is useful when an input waveform can show some offset. Auto-zero resets the zero level automatically, by subtracting the minimum level from the maximum level and dividing the result by two. Alternatively, a user can specify zero-level implicitly in <zero>.
A user can also set up the A/D sampling rate. The proper range of sampling rates is between 64 and 256 times the expected waveform frequency.

The function returns basic parameters of the waveform presented on S1 and S2 in the following structure (it is user's responsibility to allocate those structures).

```
typedef struct {
      uint32 min_lvl;     // waveform minimum
      uint32 max_lvl;     // waveform maximum
      uint32 sum;         // average accumulated sum (Sa or Sb)
      uint32 cal;         // calibrated position
      uint32 raw;         // raw position
      uint32 zc0;         // zero-crossing 0
      uint32 zc1;         // zero-crossing 1
      uint32 zc0_var;     // variation of zc0
      uint32 zc1_var;     // variation of zc1
      uint32 raw_var;     // variation of the raw position
      uint32 clk_frq;     // selected AIn/AOut clock
      float ampl;         // waveform amplitude
      float freq;         // waveform frequency
      float offs;         // waveform offset
} WFPRM_254, *pWFPRM_254;
```

Information is returned for each S1 and S2 separately, in the hardware representation.

`<min_lvl>` and `<max_lvl>` are readings from minimum and maximum level detectors in int16 format. You can calculate the span of the waveform by subtracting the minimum from the maximum. The maximum span of the waveform is $22V_{pp}$.

`<sum>` is a current sum (integration) of all measurements performed during a half-period of the waveform from one zero-crossing event to another.

`<cal>` is a calibrated position (the one returned in `DqAdv254Read()` a 24-bit 2s complement number.

`<raw>` is the same position before applying the <usr_offset> and <usr_gain> values specified in `DqAdv254SetMode().`

`<zc0>` is a number of 33 MHz pulses counted for the positive segment of the waveform (counted when waveform level is above `<zero>` level). To calculate the waveform frequency, use (zc0+zc1)*30.3ns

`<zc1>` is a number of 33 MHz pulses for the negative part of the waveform.

`<zc0_var>, <zc1_var>` is a variation in counts between last 16 measurements of `<zc0>` and `<zc1>` respectively, in 33 MHz counts.

`<raw_var>` is a variation of the position data, in counts over last 16 measurements
`<clk_frq>` is the sampling clock divider
`<ampl>` is the calculated waveform amplitude
`<freq>` is the calculated waveform frequency
`<offs>` is the calculated waveform offset

**Note:**

Use of this function requires a good knowledge of AI-254 layer design. It is very useful for debugging and selecting parameters when programming for a new LVDT sensor or simulating an LVDT output. For the most purposes, use default input settings and use the calculated amplitude and frequency as parameters for `DqAdv254SetMode`().

### *4.14.5 DqAdv254MeasureWF*

**Syntax:**

```
int  DqAdv254MeasureWF(int hd, int devn, uint32 changain, double*
frequency_b, double* amplitude_b, double* offset_b)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 changain` | Channel and gain to use ((gain << 8)\| channel) |

**Output:**

| | |
|---|---|
| `double* frequency_b` | signal frequency of the signal on channel B |
| `double* amplitude_b` | amplitude of the signal on channel B |
| `double* offset_b` | offset of the signal on channel B |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-254 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function is a simplified version of `DqAdv254GetWFMeasurements()` that allows measurements on channel B only. This is sufficient for most applications.

### *4.14.6 DqAdv254Enable*

**Syntax:**

```
int  DqAdv254Enable(int hd, int devn, uint32 config, int enable, int CLSize,
uint32* cl);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 config` | Reserved |
| `int enable` | TRUE to enable channels |
| `int CLSize` | Size of the supplied channel list |
| `uint32* cl` | Channel list |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-254 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function enables/disables operations for the channels specified in the channel list. This function should be called after DqAdv254SetMode() in point-to-point mode.

**Notes:**

This function should not be used in DMap mode.

### 4.14.7 DqAdv254GetExcitation

**Syntax:**

```
int  DqAdv254GetExcitation(int hd, int devn, uint32 channel, uint32 config,
double* exc_rate, double* exc_level, double* ADC_rate, uint32* sine_points);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 config` | Reserved |
| `uint32 channel` | Channel of interest |
| `double* exc_rate` | Actual excitation rate |
| `double* exc_level` | Actual excitation level |
| `double* ADC_rate` | Current ADCs and DACs rate |
| `uint32* sine_points` | Number of waveform points per period |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-254 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function retrieves actual parameters of excitation waveform. They may be slightly different compared to parameters specified in DqAdv254SetExcitation() because of hardware limitations and mode of operation

**Notes:**

### *4.14.8 DqAdv254Read*

**Syntax:**

```
int  DqAdv254Read(int hd, int devn, int CLSize, uint32* cl, uint32* bdata,
double* fdata);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Length of `uint32* cl` |
| `uint32* cl` | Channels of interest |

**Output:**

| | |
|---|---|
| `uint32* bdata` | Pointer to store binary 24-bit 2s complement format or NULL |
| `double* fdata` | Pointer to store converted position data from -1.0 to +1.0 or NULL |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-254 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads calculated position or special data for selected channels

Special channels are defined as follows:

```
// Special channels - AIn
Special channels - AIn
#define DQ_AI254_DIV_CAL (0x00) // calibrated result of division
#define DQ_AI254_DIV_RAW (0x10) // raw result of (Sa-Sb)/(Sa+Sb)
#define DQ_AI254_AVG     (0x20) // average position
#define DQ_AI254_ZC      (0x30) // number of counts for both half periods
#define DQ_AI254_LAST_A  (0x40) // last value from A/D A
#define DQ_AI254_MAX_A   (0x48) // maximum value A
#define DQ_AI254_LAST_B  (0x50) // last value from A/D B
#define DQ_AI254_MAX_B   (0x58) // maximum value B
#define DQ_AI254_LAST_Sa (0x60) // last calc value from Sa adder
#define DQ_AI254_MIN_A   (0x68) // minimum value A
#define DQ_AI254_LAST_Sb (0x70) // last calc value from Sb adder
#define DQ_AI254_MIN_B   (0x78) // minimum value B
```

```
#define DQ_LNCL_TIMESTAMP(0xFF) // get timestamp

#define DQ_AI254_STATUS  (0x18) // retrieve status information

#define DQ_AI254_CHTYPE  (0xf8) // channel type mask (two lower bits are channel)
#define DQ_AI254_CHNUM   (0x3)  // channel number only
```

A user has to add the AI-254 channel number to the constant defined above to form a full channel number.
Please note that only `DQ_AI254_DIV_CAL` information makes sense in floating-point format.
Zero-crossing results are returned as two uint16 values packed into one uint32 field, where ZC0 occupies the lower part of the word and ZC1 occupies the upper one.
`DQ_AI254_LAST_A and DQ_AI254_LAST_B` returns the last immediate value converted by A/D converters for channels S1 and S2, respectively.
`DQ_AI254_LAST_Sa` and `DQ_AI254_LAST_Sb` return the last immediate values from the integrator, accumulated over the last period of the input sinewave.

**Notes:**

Use the same channel identifications to specify channels for DMap operation.

### *4.14.9 DqAdv254ReadVrms*

**Syntax:**

```
int DqAdv254ReadVrms(int hd, int devn, int CLSize, uint32* cl, double* pos,
double* VArms, double* VBrms);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | length of channel list |
| `uint32* cl` | Channel list (only channel numbers with gains are valid entries) |

**Output:**

| | |
|---|---|
| `double* pos` | Calculated position [-1..+1] |
| `double* VArms` | Pointer to double array to store RMS voltage on channel A of specified channels or NULL |
| `double* VBrms` | Pointer to double array to store RMS voltage on channel A of specified channels or NULL |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-254 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |

| | |
|---|---|
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function calculates RMS voltage on channels A and B (S1 and S2) of the channels specified in the channel list.

**Notes:**

This function uses point-by-point mode function DqAdv254Read() to retrieve data

### *4.14.10 DqAdv254Write*

**Syntax:**

```
int DAQLIB DqAdv254Write(int hd, int devn, int CLSize, uint32* cl, double*
amplitude, double* fdata);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Channel list size |
| `uint32* cl` | Channels of interest |
| `double* amplitude` | Signal amplitude for each channel |
| `double* fdata` | Simulated position data from -1.0 to +1.0 |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-254 |
| `DQ_BAD_PARAMETER` | configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function writes a simulated position to simulate 4 and 5/6 wire LVDTs.

The function converts a position, depending on the mode of operation. <amplitude> defines signal amplitude, peak-to-peak (0..11V)

For 5/6-wire simulation mode, the output is always in-phase with the excitation signal; the amplitude varies from 0 to 100% depending on the simulated position. For a 4-wire LVDT, the phase is switched from 0 to 180 degrees when the simulated position crosses a zero point.

### *4.14.11 DqAdv254ConvertSim*

**Syntax:**

```
int DAQLIB DqAdv254ConvertSim(int hd, int devn, int CLSize, uint32* cl,
double* amplitude, double* fdata, uint32* act_cl, uint32* bdata, uint32*
act_size)
```

**Command:**

None

**Input:**

| | |
|---|---|
| `int mode` | Mode of operation as set in DqAdv254SetMode() |
| `int CLSize` | number of channels in `uint32* cl` |
| `uint32* cl` | list of channel numbers to convert |
| `double* amplitude` | Amplitude for each channel |
| `double* fdata` | Simulated position data from -1.0 to +1.0 |

**Output:**

| | |
|---|---|
| `uint32* act_cl` | Actual channel list required for DMap operations |
| `uint32* bdata` | Binary data |
| `uint32* act_size` | Actual channel list size for use with DMap |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function is intended to simplify programming of LVDT simulation in DMap mode. It converts a +/-1.0 position into raw data representation for gain and phase control taking into consideration whether 4 or 5/6 wiring is in use.

An AI-254 requires controlling the gain for both P1 and P2 outputs for a 5/6 wire LVDT simulation and the P1 output and phase for 4-wire one. Because of that, each position function produces a pair of pre-packaged channel lists and binary data ready to be written into DMap fields or for use with the DqAdv254WriteBin() function.

**Notes:**

Don't forget to allocate arrays for <act_cl> and <bdata> twice as large as CLSize.

### *4.14.12 DqAdv254WriteBin*

**Syntax:**

```
int  DqAdv254WriteBin(int hd, int devn, int CLSize, uint32* cl, uint32*
bdata);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | number of channels in `uint32* cl` |
| `uint32* cl` | Channels of interest |
| `uint32* bdata` | Binary gain or phase data |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-254 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

The function writes a simulated position of an LVDT or special data for selected channels. It writes data separately into four special channels defined as:

```
#define DQ_AI254_GAIN_A  (0x00)  // gain of P1 excitation channel
#define DQ_AI254_GAIN_B  (0x10)  // gain of P2 excitation channel
#define DQ_AI254_PHASE_A (0x20)  // phase of P1 excitation channel
#define DQ_AI254_PHASE_B (0x30)  // phase of P2 excitation channel
```

Gain should be in the range of 0x0 0x8000.

Phase should be in the range of 0 thru the number of points minus one in the generated sinewave (which can be obtained using DqAdv254GetExcitation() call).

You will need to use the following macro to pack parameters for phase setting

```
#define DQ_AO254_PHASE_SET(PHASE, DELAY) (((PHASE)<<18)|((DELAY)&0x3ffff))
```

PHASE is an offset from which sinewave is output and DELAY specifies delay in uS (for fine-tune of phase control, normally set to zero)

**Notes:**

To write a position from -1 to +1, use the DqAdv254Write() function.

### 4.14.13 DqAdv254SetWForm

**Syntax:**

```
int  DqAdv254SetWForm(int hd, int devn, uint32 channel, uint32 mask, double
upd_rate, int size, uint16* data);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 channel` | Channel to use |
| `uint32 mask` | Which D/A to write waveform to: 0x1 to P1, 0x3 to both |
| `double upd_rate` | Desired update rate |
| `int size` | Number of points in the waveform |
| `uint16* data` | Waveform data |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-254 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up a custom waveform for excitation channels P1 and/or P2. It may be useful in certain circumstances, such as when a standard sinewave needs to be replaced with a custom waveform. In such a case, the function DqAdv254SetWForm () should be used instead of DqAdv254SetExcitation().

**Notes:**

This function is only supported in the DNx-AI-254.

## 4.15 DNA-AI-255 layer

The DNA-AI-255 layer is a two-channel all-digital synchro and resolver input-output module.

Modes of operation supported by an AI-255:

1. Synchro Input
2. Resolver Input
3. Synchro Output/Simulator
4. Resolver Output/Simulator

Modes of operation are defined as follows:

```
DQ_AI255_MODE_SI_INT    0 // Synchro input, int. exc.
DQ_AI255_MODE_RI_INT    1 // Resolver input, int.exc.
DQ_AI255_MODE_SI_EXT    2 // Synchro input, ext. exc. - readback exc. on D
DQ_AI255_MODE_RI_EXT    3 // Resolver input, ext. exc. - readback exc. on D

DQ_AI255_MODE_SS_INT    4 // Synchro drive/simulation, int. exc.
                          //  - fully sourced
DQ_AI255_MODE_RS_INT    5 // Resolver drive/simulation, int.exc.
                          //  - fully sourced
DQ_AI255_MODE_SS_EXT    6 // Synchro drive/simulation, ext. exc.
                          //  - readback exc. on D
DQ_AI255_MODE_RS_EXT    7 // Resolver drive/simulation, ext. exc.
                          //  - readback exc. on D
```

A synchro has three stator coils S1, S2, S3 connected in a star fashion to the Common. The rotor primary coil (exciter) has wires R1 and R2
Resolver stator coils are S1-S3 and S2-S4. Rotor coil is R1-R3 (and R2-R4 in the case of two rotor windings)

**Wiring**

| | Synchro (three 120º coils) | | Resolver (two 90º coils) | |
|---|---|---|---|---|
| | AOut | AIn | AOut | AIn |
| Input, internal excitation, 28Vrms 400Hz-4kHz | R1 and R2 connected to D+ and D- (optionally use A, B, or C in parallel) | S1 to A+, S2 to B+, Common connected to A- and B-. Optionally S3 to C+ | R1 and R3 connected to A+ and A-, optionally R2 and R4 to B+ and B- (two windings per rotor) | S1 to A+, S3 to A-, S2 to B+, S4 to B- |
| Input, external excitation 28Vrms (from A/C bus) | N/C | S1 to A+, S2 to B+, Common connected to A- and B-, Optionally S3 to C+, Excitation readback to D+/D- | N/C | S1 to A+, S3 to A-, S2 to B+, S4 to B-, Excitation readback to D+/D- |

| | Synchro (three 120º coils) | | Resolver (two 90º coils) | |
|---|---|---|---|---|
| | AOut | AIn | AOut | AIn |
| Output, internal excitation, 28Vrms | S1 to A+, S2 to B+, S3 to C+, R1 to D+, R2 to D- and Common to AGND | N/C | S1 to A+, S3 to A-, S2 to B+, S4 to B-<br>R1 to D+, R3 to D- and Common to AGND<br>Optionally R2 to C+ and R4 to C- | N/C |
| Output, external excitation 28Vrms (from A/C bus), internal drive (resolver only) | S1 to A+, S2 to B+, S3 to C+ and Common to AGND | Excitation readback to D+/D- | S1 to A+, S3 to A-, S2 to B+ and S4 to B- | Excitation readback to D+/D- |

## Mode: Input, internal excitation

```
DQ_AI255_MODE_SI_INT:
DQ_AI255_MODE_RI_INT:

     exc_rate = 400.0; <- excitation frequency is the same for both calls
     exc_level = 26.0; <- level for the rotor coil
     adc_rate = 0; -> returns actual sampling rate
     ret = DqAdv255SetExcitation(hd0, DEVN, CHANNEL,
                                 DQ_AI255_ENABLE_EXC_A, // A channel only
                                 exc_rate, exc_level, &adc_rate);

     exc_level = 26.0; <- level for the rotor coil (from the datasheet)
     ret = DqAdv255SetMode(hd0, DEVN, CHANNEL, mode, flags,
                          (float)exc_rate, (float)exc_level);
```

## Mode: Input, external excitation

```
DQ_AI255_MODE_SI_EXT:
DQ_AI255_MODE_RI_EXT:

// Measure frequency and level on input D
ret = DqAdv255MeasureWF(hd0, DEVN, CHANNEL_COARSE,
                        &exc_rate, &exc_level, &exc_offs);


// use excitation frequency measured on the rotor winding
// use excitation voltage measured on the rotor winding
// this information is required to estimate A/D settings
ret = DqAdv255SetMode(hd0, DEVN, CHANNEL, mode, flags, exc_rate, exc_level);
```

**Mode: Output, internal excitation**

```
DQ_AI255_MODE_SS_INT:
DQ_AI255_MODE_RS_INT:

     exc_rate = 400.0; <- excitation frequency is the same for both calls
     exc_level = 26.0; <- level for the rotor coil
     adc_rate = 0; -> returns actual sampling rate
     ret = DqAdv255SetExcitation(hd0, DEVN, CHANNEL_SIM,
                                DQ_AI255_ENABLE_EXC_D, // D channel only
                                exc_rate, exc_level, &adc_rate);

    exc_level = 11.8; <- level for the stator coils (from the datasheet)
    ret = DqAdv255SetMode(hd0, DEVN, CHANNEL_SIM, mode, flags,
                         (float)exc_rate, (float)exc_level);
```

**Mode: Output, external excitation**

```
DQ_AI255_MODE_SS_EXT:
DQ_AI255_MODE_RS_EXT:

// Measure frequency and level on input D
ret = DqAdv255MeasureWF(hd0, DEVN, CHANNEL_COARSE,
                        &exc_rate, &exc_level, &exc_offs);

exc_level = 11.8; <- level for the stator coils (from the datasheet)
ret = DqAdv255SetMode(hd0, DEVN, CHANNEL, mode, flags, exc_rate, exc_level);
```

### *4.15.1 DqAdv255SetMode*

**Syntax:**

```
int DAQLIB DqAdv255SetMode(int hd, int devn, uint32 channel,
uint32 mode, uint32 flags, uint32* meas_pts, double exc_freq,
double Se_level)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 channel` | Channel to apply mode to |
| `uint32 mode` | Mode of operation |
| `uint32 flags` | Additional flags, reserved |
| `double exc_freq` | Expected excitation frequency for external excitation and simulation modes (need not to be exact, ignored for internal excitation modes) |
| `double Se_level` | Expected excitation level (Vpp) to use for gain selection. For simulation mode this is an initial excitation level in Volts (0..80 Vpp) |

**Output:**

| | |
|---|---|
| `uint32* meas_pts` | Number of data points per period in the sampled or generated sinewave. |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-255 or an AI-256 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, or a channel number in `cl` is incorrect |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up one of nine operating modes supported by AI-255 layer.
The supported modes are:

```
DQ_AI255_MODE_SI_INT    0 // Synchro input, int. exc.
DQ_AI255_MODE_RI_INT    1 // Resolver input, int.exc.
DQ_AI255_MODE_SI_EXT    2 // Synchro input, ext. exc. - readback exc. on D
DQ_AI255_MODE_RI_EXT    3 // Resolver input, ext. exc. - readback exc. on D

DQ_AI255_MODE_SS_INT    4 // Synchro drive/simulation, int. exc.
                          //  - fully sourced
DQ_AI255_MODE_RS_INT    5 // Resolver drive/simulation, int.exc.
                          //  - fully sourced
DQ_AI255_MODE_SS_EXT    6 // Synchro drive/simulation, ext. exc.
                          //  - readback exc. on D
DQ_AI255_MODE_RS_EXT    7 // Resolver drive/simulation, ext. exc.
                          //  - readback exc. on D
DQ_AI255_MODE_EN_AIN    9 // Allow use of inputs on unused channels
                          // as generic analog inputs, see DqAdv255Read()
```

**Notes:**
See wiring description for the proper wiring in various modes.

*Version Compatibility Note:*
The following note affects **only** resolver simulation with external excitation when set with `DqAdv255SetMode`:
In release versions 4.4.0.0 through 4.10.0.36, programming a gain of 1 would be bumped to gain of 2. If your application was developed using a software version 4.4.0.0 through 4.10.0.36 and you are updating your software version, you can add `DQ_AI255_GAIN_2` to the channel parameter with `DQ_LNCL_CHANGAIN` macro for compatibility with newer versions

### *4.15.1.1 DqAdv255SetExt*

**Syntax:**

```
int DAQLIB DqAdv255SetExt(int hd, int devn, uint32 channel,
uint32 mode, uint32 flags, pDQ255SetExt params)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 channel` | Channel to apply mode to |
| `uint32 mode` | Mode of operation |
| `uint32 flags` | Flags defining valid parameters |
| `pDQ255SetExt params` | Parameters |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-255 or an AI-256 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function allows setting up additional parameters for AI-255 layer. It is not necessary to call this function if you would like to rely on default settings.

Three parameters can be changed as defined in the following structure:

```
typedef struct {
    uint32 PositionAvg;    // Set moving average window size:
                           //   0=1,...,8=256
    uint32 ZeroCrossing;   // Set zero crossing window: 0=1,...,6=64
} DQ255SetExt, *pDQ255SetExt;
```

`<flags>` define valid parameters:

If a flag is set then parameter is valid and used, if not set it is ignored:

```
DQ_AI254_MODE_SETAVG   - Set position calculation moving average 0=1,
                         1=2, ..., 8=256
DQ_AI254_MODE_SETZEROC - Set zero crossing window 0=1,...,
                         6=64 samples
```

Notes:

### *4.15.2 DqAdv255SetExcitation*

**Syntax:**

```
    int  DqAdv255SetExcitation(int hd, int devn, uint32 channel, uint32 config,
double exc_rate, double exc_level, double* calc_rate);
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 channel` | Channel to apply mode to |
| `uint32 config` | Configuration: DQ_AI255_ENABLE_EXC_A to enable output A and DQ_AI255_ENABLE_EXC_B to enable output B, etc. |
| `double exc_rate` | Desired excitation rate in Hz |
| `double exc_level` | Desired excitation level, in V (0..80Vpp) |

**Output:**

| | |
|---|---|
| `double* calc_rate` | Actual D/A rate, limited by the abilities of D/A converter, timebase error and number of points per period. For informational purposes only |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-255 or an AI-256 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function is used to set up the internal excitation frequency and amplitude in internal excitation modes of operation.

Following constants are used in <config>:

```
DQ_AI255_ENABLE_EXC_A   (1L<<0) // enable excitation channel A
DQ_AI255_ENABLE_EXC_B   (1L<<1) // ditto B
DQ_AI255_ENABLE_EXC_C   (1L<<2) // ditto C
DQ_AI255_ENABLE_EXC_D   (1L<<3) // ditto D
```

**Notes:**

### *4.15.3 DqAdv255GetWFMeasurements*

**Syntax:**

```
      int DqAdv255GetWFMeasurements(int hd, int devn, uint32 config,
pWFMEASURE_255 prms, pWFPRM_255 chan_m);
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 config` | Reserved. Set to 0. |
| `pWFMEASURE_255 prms` | Measurement parameters |

**Output:**

| | |
|---|---|
| `pWFPRM_255 chan_m` | Results of measurements for all channel |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-255 or an AI-256 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function measures parameters of the waveform presented on the inputs A thru D of the channel. The waveform parameter measurements are required for setting up proper excitation frequency in DqAdv255SetMode() for modes with external excitation and simulation.

The function requires input that specifies the parameters of measurement:

```
typedef struct {
       uint32 changain;         // channel and gain
       uint32 zero;             // zero level, 0 == default (0x8000)
       uint32 clock;            // A/D rate, 0 == maximum rate
       uint32 uzb;              // TRUE == use ZC on channel S2
       uint32 uaz;              // TRUE == auto-zero
       uint32 s_e;              // if divider is set (not-zero), use Sa/Se
       uint32 period_ms;        // maximum expected waveform period
       uint32 phase_meas;    // channel to use as reference (exc.input)
} WFMEASURE_255, *pWFMEASURE_255;
```

The <`changain`> parameter is required. The rest of parameters are not required and can be set to zero.

Auto-zero is a function of the hardware, which is useful when the input waveform can show some offset. Auto-zero resets zero level automatically by subtracting the minimum level from the maximum level and dividing the result by two. Alternatively, a user can specify zero-level implicitly in <`zero`>. User can also set up A/D sampling rate. The proper range of sampling rates is between 64 and 256 times the expected waveform frequency.

The function returns basic parameters of the waveform presented on all four channels in an array of four following structures (it is user responsibility to allocate this array).

```
typedef struct {
      uint32 min_lvl;     // waveform minimum
      uint32 max_lvl;     // waveform maximum
      uint32 sum;         // average accumulated sum (Sa...Sd)
      uint32 cal;         // calibrated angle
      uint32 raw;         // raw angle
      uint32 zc0;         // zero-crossing 0
      uint32 zc1;         // zero-crossing 1
      uint32 zc0_var;     // variation of zc0
      uint32 zc1_var;     // variation of zc1
      uint32 raw_var;     // variation of the raw position
      uint32 clk_frq;     // selected AIn/AOut clock
      uint32 wf_phase;    // phase in 15.15ns clocks
      float ampl;         // waveform amplitude
      float freq;         // waveform frequency
      float offs;         // waveform offset
} WFPRM_255, *pWFPRM_255;
```

Information is returned for each channel separately, in the hardware representation.
`<min_lvl>` and `<max_lvl>` are readings from minimum and maximum level detectors in int16 format. You can calculate the span of the waveform by subtracting the minimum from the maximum. The maximum span of the waveform is 80$V_{pp}$ measured from AGND to Sx.
`<sum>` is a current sum (integration) of all measurements performed during the half-period of the waveform from one zero-crossing event to another.
`<cal>` is a calibrated position of a synchro or resolver (the one returned in DqAdv255Read(); this is a 24-bit 2s complement number.
`<raw>` is the position of a synchro or resolver before applying the <usr_offset> and <usr_gain> values specified in DqAdv255SetMode().
`<zc0>` is a number of 33MHz pulses counted for the positive segment of the waveform (counted when waveform level is above `<zero>` level). To calculate waveform frequency, use (`zc0`+`zc1`)*30.3ns
`<zc1>` is a number of 33-MHz pulses for the negative part of the waveform.
`<zc0_var>, <zc1_var>` is a variation in counts between last 16 measurements of <`zc0`> and <`zc1`> respectively, in 33MHz counts.
<`raw_var`> is a variation of the position data in counts over last 16 measurements.
<`clk_frq`> is the sampling clock divider.
<`ampl`> is the calculated waveform amplitude.
<`freq`> is the calculated waveform frequency.
<`offs`> is the calculated waveform offset.

**Note:**
Use of this function requires a good knowledge of AI-255 layer design. It is very useful for debugging and selecting parameters when programming for a new synchro or resolver input or output. For most purposes, use the default input settings and use the calculated amplitude and frequency as parameters for DqAdv255SetMode().

### 4.15.4 DqAdv255MeasureWF

**Syntax:**

```
    int DqAdv255MeasureWF(int hd, int devn, uint32 changain, double*
frequency_d, double* amplitude_d, double* offset_d);
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 changain` | Channel and gain to use ((gain << 8)\| channel) |

**Output:**

| | |
|---|---|
| double* frequency_d | signal frequency of the signal on channel D |
| double* amplitude_d | amplitude of the signal on channel D |
| double* offset_d | offset of the signal on channel D |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-255 or an AI-256 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function is a simplified version of `DqAdv255GetWFMeasurements()` that allows measurements on channel D only. This is sufficient for most applications.

### *4.15.5 DqAdv255Enable*

**Syntax:**

```
int  DqAdv255Enable(int hd, int devn, uint32 config, int enable, int CLSize,
uint32* cl);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 config` | Reserved |
| `int enable` | TRUE to enable channels |
| `int CLSize` | Size of the supplied channel list |
| `uint32* cl` | Channel list |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-255 or an AI-256 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function enables/disables operations for the channels specified in the channel list. This function should be called after DqAdv255SetMode() in point-to-point mode. Channel list should be NULL if enable = FALSE.

**Notes:**

This function should not be used in DMap mode.

### *4.15.6 DqAdv255GetExcitation*

**Syntax:**

```
int  DqAdv255GetExcitation(int hd, int devn, uint32 channel, uint32 config,
double* exc_rate, double* exc_level, double* ADC_rate, uint32* sine_points)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 config` | Reserved |
| `uint32 channel` | Channel of interest |
| `double* exc_rate` | Actual excitation rate |
| `double* exc_level` | Actual excitation level |
| `double* ADC_rate` | Current ADCs and DACs rate |
| `uint32* sine_points` | Number of waveform points per period |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-255 or an AI-256 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Gets layer excitation voltage parameters calculated by Ain A thru D channels while acquiring an externally or internally generated waveform. They may be slightly different compared to parameters specified in DqAdv255SetExcitation() because of hardware limitations and mode of operation.

**Notes:**

### 4.15.7 DqAdv255Read

**Syntax:**

```
int DAQLIB DqAdv255Read(int hd, int devn, int CLSize, uint32* cl, uint32* bdata,
double* fdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | number of channels in array cl |
| `uint32* cl` | Array of Channels of interest |

**Output:**

| | |
|---|---|
| `uint32* bdata` | Pointer to store binary 24-bit 2s complement format or NULL |
| `double* fdata` | Converted data (if applicable; in Pi Rad = 0..6.2830...) or NULL |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-255 or an AI-256 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Read the calculated angle or special data for selected channels

Special channels are defined as follows (more in powerdna.h):

```
DQ_AI255_ANGLE_CAL (0x00) // calibrated angle
DQ_AI255_ACCEL_CAL (0x10) // calibrate acceleration
DQ_AI255_STATUS    (0x18) // current status information
DQ_AI255_RAW_DATA  (0x20) // raw position
DQ_AI255_ZC        (0x30) // number of counts for both half periods
DQ_AI255_LAST_X    (0x40) // results of the last conversion A thru C
DQ_AI255_LAST_SxX  (0x50) // last accumulated value from A/D A thru C
DQ_AI255_MIN_X     (0x60) // minimum reading A thru C
DQ_AI255_MAX_X     (0x70) // maximum A thru C
DQ_AI255_PHASE_DET (0x80) // phase detector for channels 0 and 1
```

```
#define DQ_AI255_CHTYPE            (0xfc)      // channel type mask
```

Bits for A/D A-C occupy bits `[3,2]` in the channel entry
for example:
`entry = channel | (ADC << 2) | DQ_AI255_LAST_Sx`

Gain factor for reading occupy bits [9,8] in the channel list entry, see DQ_AI255_GAIN_1 ... DQ_AI255_GAIN_10 in powerdna.h
User has to add the AI-255 channel number to the constant defined above to form a full channel number.

Zero-crossing results are returned as two uint16 values packed into one uint32 field, where ZC0 occupies the lower part of the word and ZC1 occupies the upper one. Note that this channel is useful only when the excitation frequency is greater than 252Hz

`DQ_AI255_LAST_X` returns the last immediate value converted by A/D converters for channels A thru C, respectively. These channels may be used to retrieve the raw ADC inputs when DQ_AI255_MODE_EN_AIN is used.

`DQ_AI255_MIN_`*`X and`* `DQ_AI255_MAX_`*`X`* return the minimum and maximum value of the input signal over the last period of the input waveform. They are useful to read the peak-to-peak value of the input signal. (Or RMS value, multiply the P-P value by 0.3535)

`DQ_AI255_LAST_Sx` returns the last immediate value from the integrator, accumulated over the last period of the input sinewave.
`DQ_AI255_PHASE_DET` returns the time shift between input D and the highest amplitude signal on inputs A-C. The phase is angle is calculated: (bdata[x] / (66000000/(input D frequency)) ) * 360.

**Notes:**

Use the same channel identifications to specify channels for DMap operation

### 4.15.8 DqAdv255Write

**Syntax:**

```
int  DqAdv255Write(int hd, int devn, int CLSize, uint32* cl, double* amplitude,
double* fdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `double* amplitude` | Maximum voltage of the simulated stator signals (0..80Vpp) for the AI-255, (0..57.8Vpp) for the AI-256 |
| `int CLSize` | Channel list |
| `uint32* cl` | Channels of interest |
| `double* fdata` | Output position in radians (from 0 to 2 Pi Rad - 0..6.2830...) |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-255 or an AI-256 |
| `DQ_BAD_PARAMETER` | configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Writes a simulated position of a synchro or resolver or special data for selected channels.

For a synchro, the simulated position is calculated as:
A - S1: $\sin(-(\varphi) - 30°)) * \text{max\_amplitude}$
B - S2: $\sin(-(\varphi) - 150°)) * \text{max\_amplitude}$
C - S3: $\sin(-(\varphi) - 270°)) * \text{max\_amplitude}$
where $\varphi$ is in degrees.

The reason for this transformation is that an AI-255 controls voltages on S1, S2 and S3-relative-to-AGND for the purpose of creating the following voltages between synchro lines:
$V_{S1S2} = \sin(\varphi) * \text{max\_amplitude}$
$V_{S2S3} = \sin(\varphi+120°) * \text{max\_amplitude}$
$V_{S3S1} = \sin(\varphi+240°) * \text{max\_amplitude}$

If sin(φ) is greater than zero, the output is in the same phase as the excitation voltage; otherwise, it is rotated by 180°.

For a resolver, the simulated position is calculated as:

A - S1: sin ( φ ) * max_amplitude
B - S2: cos ( φ ) * max_amplitude

If sin(φ) or cos(φ) is greater than zero, the output is the same phase as the excitation voltage; otherwise, it is rotated by 180°.

**Notes:**
The proper device type should be selected in DqAdv255SetMode()
Amplitude is the maximum momentary value of the signal. The sinewave signal is relative to zero.

### *4.15.9 DqAdv255ConvertSim*

**Syntax:**

```
int DAQLIB DqAdv255ConvertSim(int hd, int devn, int CLSize, uint32* cl, double*
amplitude, double* fdata, uint32* act_cl, uint32* bdata, uint32* act_size)
```

**Command:**

None

**Input:**

| | |
|---|---|
| `int mode` | Mode of operation as set in DqAdv255SetMode() |
| `int CLSize` | Length of channel list |
| `uint32* cl` | Channel list |
| `double* amplitude` | Amplitude of the output signal (0..80Vpp) |
| `double* fdata` | Simulated position data in radians (from 0 to 2*pi) |

**Output:**

| | |
|---|---|
| `uint32* act_cl` | Actual channel list required for DMap operations |
| `uint32* bdata` | Binary data |
| `uint32* act_size` | Actual channel list size |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SUCCESS` | successful completion |

**Description:**

This function is intended to simplify programming of synchro/resolver output in DMap mode. It converts 0…2pi*rad angle into raw data representation, taking gain and phase control into consideration.

Because of that, for each position this function produces a pair of pre-packaged channel list and binary data ready to be written into DMap fields or for use DqAdv255WriteBin() function.

A synchro requires control of three amplitudes and three phases. Resolver requires control of two amplitude and two phases.

Data is calculated as follows:

A - S1: sin ( -( φ) - 30°)) * max_amplitude
B - S2: sin ( -( φ) - 150°)) * max_amplitude
C - S3: sin ( -( φ) - 270°)) * max_amplitude

The reason for this transformation is that an AI-255 controls voltages on S1, S2 and S3 for the purpose of creating the following functions
$V_{S1S2} = \sin(\varphi)$
$V_{S2S3} = \sin(\varphi+120°)$
$V_{S3S1} = \sin(\varphi+240°)$

If sin(φ) is greater than zero, the output is in the same phase as excitation voltage; otherwise, it is rotated by 180°.

For a resolver, a simulated position is calculated as:
A - S1: sin ( φ ) * max_amplitude
B - S2: cos ( φ ) * max_amplitude

If sin(φ) or cos(φ) is greater than zero, the output is in the same phase as excitation voltage; otherwise, it is in rotated by 180°.

**Notes:**

Don't forget to allocate arrays for <act_cl> and <bdata> four times as large as CLSize for a resolver and six times as large as CLSize for a synchro.

Pass NULL as <bdata> for creating a channel list for DqRtDmapAddChannel().

### 4.15.10 DqAdv255WriteBin

**Syntax:**

```
int  DqAdv255WriteBin(int hd, int devn, int CLSize, uint32* cl, uint32* bdata);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Channel list size |
| `uint32* cl` | Channel list |
| `uint32* bdata` | Binary gain or phase data |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-255 or an AI-256 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

The function writes an angle or special data for selected channels. It writes data separately into four special channels defined as:

```
DQ_AI255_GAIN_A     (0x00)  // gain of A excitation channel
DQ_AI255_GAIN_B     (0x04)  // gain of B excitation channel
DQ_AI255_GAIN_C     (0x08)  // gain of C excitation channel
DQ_AI255_GAIN_D     (0x0C)  // gain of D excitation channel
DQ_AI255_PHASE_A    (0x10)  // phase of A excitation channel
DQ_AI255_PHASE_B    (0x14)  // phase of B excitation channel
DQ_AI255_PHASE_C    (0x18)  // phase of C excitation channel
DQ_AI255_PHASE_D    (0x1C)  // phase of D excitation channel
```

Gain should be in the range of 0x0 0x8000.

Phase should be in the range of 0 thru 0xff or in general, the number of points minus one in the generated sinewave (which can be obtained using DqAdv255GetExcitation() call).

You will need to use the following macro to pack parameters for the phase setting:

```
#define DQ_AO255_PHASE_SET(PHASE, DELAY) (((PHASE)<<18)|((DELAY)&0x3ffff))
```

PHASE is an offset from which sinewave is output and DELAY specifies a delay in uS (for fine-tune of phase control, normally set to zero)

**Notes:**

To write an angle from 0 to 2 Pi Rad, use the DqAdv255Write() function.

### 4.15.11 DqAdv255ReadDIn

**Syntax:**

```
int  DqAdv255ReadDIn(int hd, int devn, uint32* din);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `uint32* din` | Digital input value |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-255 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

The function reads available auxiliary digital inputs.

**Notes:**

### 4.15.12 DqAdv255WriteDOut

**Syntax:**

```
int  DqAdv255WriteDOut(int hd, int devn, uint32 dout, uint32* din);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 dout` | Digital output value |

**Output:**

| | |
|---|---|
| `uint32* din` | Digital input value |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-255 or an AI-256 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**
The function writes available digital outputs and reads back auxiliary digital inputs.

**Notes:**

## 4.16 DNA-AI-256 layer

The DNx-AI-256 layer is a two-channel all-digital synchro and resolver input-output module. The DNx-AI-256 has a superset of the functionality of the DNA-AI-255. All of the DqAdv255*() functions can be used with the DNx-AI-256.

### 4.16.1 DqAdv256ConvertSimLvdt

**Syntax:**

```
int DAQLIB DqAdv256ConvertSimLvdt(int hd, int devn, int CLSize, uint32* cl,
double* amplitude, double* fdata, uint32* act_cl, uint32* bdata, uint32* act_size)
```

**Command:**

None

**Input:**

| | |
|---|---|
| `int mode` | Mode of operation as set in DqAdv256SetMode() |
| `int CLSize` | number of elements in `uint32* cl` |
| `uint32* cl` | List of Channels of interest |
| `double* amplitude` | Amplitude for each channel |
| `double* fdata` | Simulated position data from -1.0 to +1.0 |

**Output:**

| | |
|---|---|
| `uint32* act_cl` | Actual channel list required for DMap operations |
| `uint32* bdata` | Binary data |
| `uint32* act_size` | Actual channel list size for use with DMap |

**Return:**

| | |
|---|---|
| DQ_NO_MEMORY | error allocating buffer |
| DQ_BAD_PARAMETER | Configuration parameters are incorrect |
| DQ_SUCCESS | successful completion |

**Description:**

This function is intended to simplify programming of LVDT simulation in DMap mode. It converts a +/-1.0 position into raw data representation for gain and phase control taking into consideration whether 4 or 5/6 wiring is in use.
An AI-256 requires controlling the gain for both P1 and P2 outputs for a 5/6 wire LVDT simulation and the P1 output and phase for 4-wire one. Because of that, each position function produces a pair of pre-packaged channel lists and binary data ready to be written into DMap fields.

**Notes:**

Don't forget to allocate arrays for <act_cl> and <bdata> twice as large as CLSize.

### *4.16.2 DqAdv256Enable*

**Syntax:**

```
int  DqAdv256Enable(int hd, int devn, uint32 config, int enable, int CLSize,
uint32* cl);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 config` | Reserved |
| `int enable` | TRUE to enable channels |
| `int CLSize` | Size of the supplied channel list |
| `uint32* cl` | Channel list |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-256 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function enables/disables operations for the channels specified in the channel list. This function should be called after DqAdv256SetMode() in point-to-point mode.

**Notes:**

This function should not be used in DMap mode.

### 4.16.3 DqAdv256ReadLvdt

**Syntax:**

```
int DqAdv256ReadLvdt(int hd, int devn, int CLSize, uint32* cl,
double* sensitivity, double* position)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Length of `uint32* cl` |
| `uint32* cl` | Channels of interest |
| `double *sensitivity` | list of sensitivities corresponding to entries in cl.<br>Sensitivity unit is mV/V/mm. |

**Output:**

| | |
|---|---|
| `double *position` | position output |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-256 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function can be used to read an LVDT device.

The channel list (`uint32* cl`) is very simple, it will be of length 1 or 2 and will contain the channel number only, i.e. 0, 1, or both.

**Note:**

Requires logic version 02.11.1E or greater to be installed on the AI-256

### *4.16.4 DqAdv256ReadPADC*

**Syntax:**

```
int DqAdv256ReadPADC(int hd, int devn, uint32* bdata, double*
fdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `uint32 *bdata` | Raw binary data, an array of 8 values, (`NULL` if not required) |
| `double *fdata` | Converted data, an array of 8 values, (`NULL` if not required) |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-256 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function returns the power monitor ADC readings from an AI-256 layer.

AI-256 Power ADC channel assignments, indices for `bdata` [] and `fdata` []:

| | | |
|---|---|---|
| DQ_AI256_SUBCH_NEGV_0 | (0) | // ch0 -VA-CH |
| DQ_AI256_SUBCH_POSV_0 | (1) | // ch0 +VCCH |
| DQ_AI256_SUBCH_I_DC_0 | (2) | // ch0 DC current on common |
| DQ_AI256_SUBCH_THERM_0 | (3) | // ch0 temperature of the ADC IC in degrees C |
| DQ_AI256_SUBCH_NEGV_1 | (4) | // ch1 -VA-CH |
| DQ_AI256_SUBCH_POSV_1 | (5) | // ch1 +VCCH |
| DQ_AI256_SUBCH_I_DC_1 | (6) | // ch1 DC current on common |
| DQ_AI256_SUBCH_THERM_1 | (7) | // ch1 temperature of the ADC IC in degrees C |

For troubleshooting or device calibration purposes, you may logical OR the DQ_AI256_UNCALFLAG constant with the device number to have *bdata return uncalibrated binary data. *fdata returns all zeros if DQ_AI256_UNCALFLAG is used.

**Note:**

None.

### *4.16.5 DqAdv256SetAll*

**Syntax:**

```
int DqAdv256SetAll(int hd, int devn, pDQAI256DATAOUT data)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `pDQAI256DATAOUT data` | pointer to store output data to AI-256 |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-256 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up most of AI-256 configuration parameters.

To accomplish what the user must allocate, initialize, and pass a pointer to a `DQAI256DATAOUT` structure is defined as:

```
/* structure for SETPARAM data (WRITE) */
typedef struct {
    int32 cfgmask;              // What parameters to set
    int32 cfg;                  // Reserved for future use
    int32 portocm;             // Set over-current interrupt mask
    int32 portocv0;            // Set over-current limit value, chan0
    int32 portocv1;            // Set over-current limit value, chan1
    int32 rdcnt;               // Set number of "failed" samples prior to breaker engagement
    int32 adccfg[DQ_AI256_PADC_CHAN];  // "User" ADC conversion control words for ADC inputs
    int32 reserved;            // Reserved for future use
} DQDIO432DATAOUT, *pDQDIO432DATAOUT;
```

<cfgmask> flags select what parameter should be written. Following flags are defined:

```
#define DQAI256_CFGSET         (1L<<0)    // =1 to set "cfg" parameter
#define DQAI256_PORTOCMSET     (1L<<1)    // =1 if "portocm" contains valid data
#define DQAI256_PORTOCV0SET    (1L<<2)   // =1 if "portocv0" contains valid data
#define DQAI256_PORTOCV1SET    (1L<<3)   // =1 if "portocv1" contains valid data
#define DQAI256_RDCNTSET       (1L<<4)   // =1 if "rdcnt" contains valid data
#define DQAI256_ADCCFG0SET     (1L<<5)   // =1 if "adccfg[0]" contains valid data
#define DQAI256_ADCCFG1SET     (1L<<6)   // =1 if "adccfg[1]" contains valid data
#define DQAI256_ADCCFG2SET     (1L<<7)   // =1 if "adccfg[2]" contains valid data
#define DQAI256_ADCCFG3SET     (1L<<8)   // =1 if "adccfg[3]" contains valid data
```

<cfg> reserved for future use.

<portocm> sets the interrupt mask for the overcurrent breaker function

<portocv0> sets the overcurrent limit for channel 0

<portocv1> sets the overcurrent limit for channel 1

<rdcnt> specifies the number of consecutive samples above the selected overcurrent limit to be acquired before disconnecting the line. Valid values are 1.31; the default value is four.

<adccfg[0]> thru <adccfg[3]> select configuration values for the PADC A/D converter. This allows the user to change the configuration values if they are required for special applications,

Note:

### *4.16.6 DqAdv256SetModeLvdt*

**Syntax:**

```
      int DAQLIB DqAdv256SetModeLvdt(int hd, int devn, uint32 channel, uint32
mode, uint32 flags, uint32* meas_pts, double usr_offset, double usr_gain, double
exc_freq, double Se_level)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 channel` | Channel to apply mode to |
| `uint32 mode` | Mode of operation |
| `uint32 flags` | Additional flags, reserved |
| `uint32* meas_pts` | Number of data points per period of the generated sine wave (equal to the number of sampling points per period) |
| `double usr_offset` | User-defined offset from -1.0 to +1.0 in (1/0x7fffff) increments to fine-calibrate output data (to trim LVDT/RVDT position) |
| `double usr_gain` | User-defined gain from 0 to 1.0 (to trim LVDT/RVDT gain) |
| `double exc_freq` | Expected excitation frequency for external excitation and simulation modes (need not be exact for external excitation modes) |
| `double Se_level` | Expected excitation level in Volts (Vpp) (2 .. 56V) |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-256 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, or a channel number in `cl` is incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up LVDT operating modes supported by the AI-256 layer.

`<usr_offset>` defines the fine-tuning offset. This offset is set in the range of -1.0 (all Sb, no Sa) to +1.0 (opposite side of LVDT) and can be used to adjust the middle point of the LVDT device. This offset is converted into a binary representation and applied in the hardware to the raw values of conversion.
`<usr_gain>` defines fine-tuning gain. This gain is set in the range from 0 to +1.0 and can be used to adjust the response characteristics of an LVDT device. This gain is converted into a binary representation and applied in the hardware to raw values of calculated position of the LVDT.
`<exc_freq>` specifies the expected excitation frequency to be received on the S4 input. This parameter can be obtained using DqAdv254GetWFMeasurements() and is required to set internal parameters such as the number of points per period and the A/D rate (to be in the acceptable range for simulation mode).
`<Se_level>` in simulation mode, it defines initial levels of the simulated output.

**Notes:**
Function returns `<meas_pts>` which is the number of points per period of the generated sinewave. This comes in handy when you need to calculate true Vrms of the sinewave.

Vrms = (Sa+Sb)/<meas_pts>

### 4.16.7 DqAdv256SetModeSynchroResolver

**Syntax:**

```
int DAQLIB DqAdv256SetModeSynchroResolver(int hd, int devn, uint32 channel, uint32
mode, uint32 flags, uint32* meas_pts, double exc_freq, double Se_level)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 channel` | Channel to apply mode to |
| `uint32 mode` | Mode of operation |
| `uint32 flags` | Additional flags, reserved |
| `double exc_freq` | Expected excitation frequency for external excitation and simulation modes (need not to be exact, ignored for internal excitation modes) |
| `double Se_level` | Expected excitation level (Vpp) to use for gain selection. For simulation mode this is an initial excitation level in Volts (0..56Vpp) |

**Output:**

| | |
|---|---|
| `uint32* meas_pts` | Number of points per period in the generated sinewave |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-255 or an AI-256 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, or a channel number in `cl` is incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up one of six Synchro/Resolver operating modes supported by AI-256 layer. It is identical to the `DqAdv255SetMode()`function. Please refer to the `DqAdv255SetMode()`function above.

### *4.16.8 DqAdv256WriteLvdt*

**Syntax:**

```
int DAQLIB DqAdv256WriteLvdt(int hd, int devn, int CLSize, uint32* cl, double*
amplitude, double* fdata);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Channel list size |
| `uint32* cl` | List of Channels of interest |
| `double* amplitude` | Signal amplitude for each channel |
| `double* fdata` | Simulated position data from -1.0 to +1.0 |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-256 |
| `DQ_BAD_PARAMETER` | configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function writes a simulated position to simulate 4 and 5/6 wire LVDTs.
The function converts a position, depending on the mode of operation. <amplitude> defines signal amplitude, peak-to-peak (0..57.8V)

For 5/6-wire simulation mode, the output is always in-phase with the excitation signal; the amplitude varies from 0 to 100% depending on the simulated position. For a 4-wire LVDT, the phase is switched from 0 to 180 degrees when the simulated position crosses a zero point.

### *4.16.9 DqAdv256WriteSynchroResolver*

**Syntax:**

```
int  DqAdv256WriteSynchroResolver(int hd, int devn, int CLSize, uint32* cl,
double* amplitude, double* fdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Channel list size |
| `uint32* cl` | Channel list |
| `double* amplitude` | Voltage of the simulated stator signals relatively to rotor excitation (0..56Vpp) |
| `double* fdata` | Output position in radians (from 0 to 2 Pi Rad - 0..6.2830...) |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AI-256 |
| `DQ_BAD_PARAMETER` | configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function writes a simulated position of a synchro or resolver or special data for selected channels. It is identical to the `DqAdv255Write()`function. Please refer to the `DqAdv255Write()`function above.

## *4.17 DNx-DMM-261 layer*

### *4.17.1 DqAdv261Read*

**Syntax:**

```
int DqAdv261Read(int hd, int devn, int CLSize, uint32* clist,
uint32* bData, double* fData, uint32* status)
```

**Command:**

Read data from the specified DMM channels.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Number of channels |
| `uint32* clist` | Pointer to channel list |

**Output:**

| | |
|---|---|
| `uint32* bData` | Calibrated binary data; (pass `NULL` if not required) |
| `double* fData` | Data converted to SI units (pass `NULL` if not required) |
| `uint32* status` | Status of input protection (pass `NULL` if not required) |

**Return:**

| | |
|---|---|
| `DQ_DATA_NOTREADY` | Data is not yet settled |
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not a DMM-261 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function can read all available data from the DMM-261 layer. When this function is called for the first time, the firmware stops any ongoing operation on the device specified and reprograms it according to the channel list supplied. The layer continuously acquires data and every subsequent call to the function returns the latest acquired data. To ensure that you are reading new data, you can confirm that the return value of the call is not `DQ_DATA_NOTREADY`.

`<clist>`:

The channel list is an array of measurement channels selected from the following table of `DQ_DMM261_x_CH` constants. Each channel list element is a uint32 word which defines the measurement mode and range (if applicable). You can build the channel list element with the help of predefined macros:

- DCV, DCI, ACV, or ACI channels: OR the channel with the `DQ_LNCL_RANGE()`macro, e.g.
  `c[2]= DQ_DMM261_VDC_SAVG_CH | DQ_LNCL_RANGE(DQ_DMM261_10VDC);`
- Resistance channels: set the channel using the `DQ_DMM261_RES_CH_RANGE()`macro, e.g.
  `c[2]= DQ_DMM261_RES_CH_RANGE(DQ_DMM261_R_2W_1K);`
  Resistance channels require 1 additional bit that `DQ_LNCL_RANGE()` would mask out.

| Channel Name | Description | Ranges |
|---|---|---|
| **DC Voltage** | | |
| `DQ_DMM261_VDC_CAL_CH` | Calibrated DC voltage with or without FIR filtering (no averaging) | `DQ_DMM261_AUTORANGE_VDC`<br>`DQ_DMM261_300VDC`<br>`DQ_DMM261_100VDC`<br>`DQ_DMM261_10VDC`<br>`DQ_DMM261_1VDC`<br>`DQ_DMM261_0_1VDC`<br>`DQ_DMM261_0_030VDC` |
| `DQ_DMM261_VDC_SAVG_CH` | DCV averaged over all samples since the previous call to `DqAdv261Read()`. | |
| `DQ_DMM261_VDC_AVG_PER_CH` | DCV averaged over `DQ_DMM261_CONFIG_RD_CYCLES` number of power line cycles<br>**Note:** This mode is for UEI internal use only. | |
| `DQ_DMM261_VDC_AVG_DCS_CH` | DCV averaged over N samples (`DQ_DMM261_CONFIG_V_ZC`).<br>**Note:** This mode is for UEI internal use only. | |
| **AC Voltage** | | |
| `DQ_DMM261_VAC_RMS_CH` | True RMS voltage (AC-coupled) computed over a time limit based on `DQ_DMM261_CONFIG_LINE_HZ`.<br>**Note:** If `VAC_SAVG_CH` is added to the channel list, `VAC_RMS_CH` will be overwritten by `VAC_SAVG_CH` and return the same N-period data. | `DQ_DMM261_AUTORANGE_VAC`<br>`DQ_DMM261_300VAC`<br>`DQ_DMM261_100VAC`<br>`DQ_DMM261_10VAC`<br>`DQ_DMM261_1_5VAC`<br>`DQ_DMM261_0_5VAC` |
| `DQ_DMM261_VAC_SAVG_CH` | True RMS voltage computed over N periods, where N is `DQ_DMM261_CONFIG_RD_CYCLES`. | |
| `DQ_DMM261_VAC_MIN_CH` | Minimum peak AC voltage; uses a configurable moving average | |
| `DQ_DMM261_VAC_MAX_CH` | Maximum peak AC voltage; uses a configurable moving average | |
| `DQ_DMM261_VAC_FREQ_CH` | Last detected AC voltage frequency | n/a |

| | | |
|---|---|---|
| **DC Current** | | |
| `DQ_DMM261_IDC_CAL_CH` | Calibrated DC current with or without FIR filtering (no averaging) | `DQ_DMM261_AUTORANGE_ADC`<br>`DQ_DMM261_RANGE_3_ADC`<br>`DQ_DMM261_RANGE_1_5_ADC`<br>`DQ_DMM261_RANGE_0_150_ADC`<br>`DQ_DMM261_RANGE_0_015_ADC`<br>`DQ_DMM261_RANGE_0_0015_ADC` |
| `DQ_DMM261_IDC_SAVG_CH` | DCI averaged over all samples since the previous call to `DqAdv261Read()` | |
| `DQ_DMM261_IDC_AVG_PER_CH` | DCI averaged over `DQ_DMM261_CONFIG_RD_CYCLES` number of power line cycles.<br>**Note:** This mode is for UEI internal use only. | |
| `DQ_DMM261_IDC_AVG_DCS_CH` | DCI averaged over N samples (`DQ_DMM261_CONFIG_I_ZC`).<br>**Note:** This mode is for UEI internal use only. | |
| **AC Current** | | |
| `DQ_DMM261_IAC_RMS_CH` | True RMS current (DC-coupled), computed over a time limit based on `DQ_DMM261_CONFIG_LINE_HZ`.<br>**Note:** If `IAC_SAVG_CH` is added to the channel list, `IAC_RMS_CH` will be overwritten by `IAC_SAVG_CH` and return the same N-period data. | `DQ_DMM261_AUTORANGE_AAC`<br>`DQ_DMM261_RANGE_2_AAC`<br>`DQ_DMM261_RANGE_1_AAC`<br>`DQ_DMM261_RANGE_0_150_AAC`<br>`DQ_DMM261_RANGE_0_015_AAC`<br>`DQ_DMM261_RANGE_0_0015_AAC` |
| `DQ_DMM261_IAC_SAVG_CH` | True RMS current computed over N periods, where N is `DQ_DMM261_CONFIG_RD_CYCLES.` | |
| `DQ_DMM261_IAC_MIN_CH` | Minimum peak AC current; uses a configurable moving average | |
| `DQ_DMM261_IAC_MAX_CH` | Maximum peak AC current; uses a configurable moving average | |
| `DQ_DMM261_IAC_FREQ_CH` | Last detected AC current frequency | n/a |

| **Resistance** | | |
|---|---|---|
| `DQ_DMM261_RES_CH` | Resistance measurement<br>Range can be programmed using the `DQ_DMM261_RES_CH_RANGE()` macro, e.g. `c[2]= DQ_DMM261_RES_CH_RANGE( DQ_DMM261_R_2W_1K);`<br>**Note:** In addition to setting the range in the channel list, resistance measurements also require a separate call to `DqAdv261SetResRange()`. | 2-wire mode:<br>`DQ_DMM261_R_2W_AUTORANGE`<br>`DQ_DMM261_R_2W_10`<br>`DQ_DMM261_R_2W_100`<br>`DQ_DMM261_R_2W_1K`<br>`DQ_DMM261_R_2W_10K`<br>`DQ_DMM261_R_2W_100K`<br>`DQ_DMM261_R_2W_1M`<br>`DQ_DMM261_R_2W_10M`<br>`DQ_DMM261_R_2W_100M`<br>4-wire mode:<br>`DQ_DMM261_R_4W_AUTORANGE`<br>`DQ_DMM261_R_4W_10`<br>`DQ_DMM261_R_4W_100`<br>`DQ_DMM261_R_4W_1K`<br>`DQ_DMM261_R_4W_10K`<br>`DQ_DMM261_R_4W_100K`<br>`DQ_DMM261_R_4W_1M`<br>`DQ_DMM261_R_4W_10M` |
| **Guardian Diagnostics** | | |
| `DQ_DMM261_GV0_CH` | -17.5 Volt supply | n/a |
| `DQ_DMM261_GV1_CH` | +17.5 Volt supply | n/a |
| `DQ_DMM261_GV2_CH` | -5.5 Volt supply | n/a |
| `DQ_DMM261_GV3_CH` | +5.5 Volt supply | n/a |
| `DQ_DMM261_TEMP_CH` | Temperature in the ADC IC (°C) | n/a |
| `DQ_DMM261_GIDC_CH` | Guardian DC current | n/a |
| `DQ_DMM261_GIAC_CH` | Guardian AC RMS current | n/a |
| `DQ_DMM261_GIAC_MIN_CH` | Minimum value of `GIAC_CH` current (peak-) | n/a |
| `DQ_DMM261_GIAC_MAX_CH` | Maximum value of `GIAC_CH` current (peak+) | n/a |
| **Timestamp** | | |
| `DQ_LNCL_TIMESTAMP` | Timestamp is reset by the firmware on the first read of a new channel list. There is some delay between the reset and the first read of the timestamp, so the first data point may not correspond to t=0. The timestamp channel's `fdata` contains the elapsed time in seconds. | n/a |

`<bData>`:

- If reading from the timestamp channel, all 32 bits of `bData` are used for the timestamp.

- For all other channels, ADC data is returned in the lower 24 bits. Bit 31 indicates when the ADC data has been updated since the last read. The remaining bits contain status information depending on the type of data being measured. Status bits in `bData` are returned as:

| Bit | DC Voltage | AC Voltage |
|---|---|---|
| 31 | `DQ_DMM261_NEW_BDATA`<br>1 = new ADC data available | `DQ_DMM261_NEW_BDATA`<br>1 = new ADC data available |
| 30 | n/a | `DQ_DMM261_RMS_VALID`<br>1 = valid AC zero crossings |
| 29 | `DQ_DMM261_OVER_RANGE`<br>1 = input is over-range | `DQ_DMM261_AC_CLIP`<br>1 = AC input is clipping |
| 28 | `DQ_DMM261_V_X100_RANGE`<br>1 = using voltage divider (100 V or 300 V range) | n/a |
| 27 | `DQ_DMM261_BDATA_AUTORANGE`<br>1 = autorange on | `DQ_DMM261_BDATA_AUTORANGE`<br>1 = autorange on |
| 26 | `DQ_DMM261_AUTOIDX`<br>Range used if autorange is on:<br>0 = 300 V<br>1 = 100 V<br>2 = 10 V<br>3 = 1 V<br>4 = 0.1 V<br>5 = 0.030 V | `DQ_DMM261_AUTOIDX`<br>Range used if autorange is on:<br>0 = 300 Vrms<br>1 = 100 Vrms<br>2 = 10 Vrms<br>3 = 1.5 Vrms<br>4 = 0.5 Vrms |
| 25 | | |
| 24 | | |

| Bit | DC Current | AC Current |
|---|---|---|
| 31 | `DQ_DMM261_NEW_BDATA`<br>1 = new ADC data available | `DQ_DMM261_NEW_BDATA`<br>1 = new ADC data available |
| 30 | n/a | `DQ_DMM261_RMS_VALID`<br>1 = valid AC zero crossings |
| 29 | `DQ_DMM261_OVER_RANGE`<br>1 = input is over-range | `DQ_DMM261_AC_CLIP`<br>1 = AC input is clipping |
| 28 | `DQ_DMM261_I_2A_RANGE`<br>1 = using 0.1 Ω shunt (0.15A or higher range) | `DQ_DMM261_I_2A_RANGE`<br>1 = using 0.1 Ω shunt (0.15A or higher range) |
| 27 | `DQ_DMM261_BDATA_AUTORANGE`<br>1 = autorange on | `DQ_DMM261_BDATA_AUTORANGE`<br>1 = autorange on |
| 26 | `DQ_DMM261_AUTOIDX`<br>Range used if autorange is on:<br>0 = 3 A<br>1 = 1.5 A<br>2 = 0.15 A<br>3 = 0.015 A<br>4 = 0.0015 A | `DQ_DMM261_AUTOIDX`<br>Range used if autorange is on:<br>0 = 2 Arms<br>1 = 1 Arms<br>2 = 0.15 Arms<br>3 = 0.015 Arms<br>4 = 0.0015 Arms |
| 25 | | |
| 24 | | |

| Bit | Resistance |
|---|---|

| | |
|---|---|
| 31 | `DQ_DMM261_NEW_BDATA`<br>1 = new ADC data available |
| 30 | `DQ_DMM261_RES_NOPEN`<br>reserved |
| 29 | `DQ_DMM261_OVER_RANGE`<br>1 = input is over-range |
| 28 | `DQ_DMM261_I_2A_RANGE`<br>1 = resistance measurement is on |
| 27 | `DQ_DMM261_BDATA_AUTORANGE`<br>1=autorange on |
| 26 | `DQ_DMM261_AUTOIDX`<br>Range used if autorange is on:<br>0 = 10 Ω<br>1 = 100 Ω<br>2 = 1 kΩ<br>3 = 10 kΩ<br>4 = 100 kΩ<br>5 = 1 MΩ<br>6 = 10 MΩ<br>7 = 100 MΩ |
| 25 | |
| 24 | |

`<status>`:

This uint32 word contains the status of the input protection system, which automatically disconnects DCV, ACV, and Resistance inputs in case of over-voltage. The following bits are defined:

- `Bit 3, DQ_DMM261_READ_STS_DCV_SETTLE_WARN:` 1= DC voltage and resistance data may still be settling. It may take up to 15 seconds for the data to completely settle.
- `Bit 2, DQ_DMM261_READ_STS_PROT_RANGE:` 1 = Protection tripped and caused the layer to jump up to a safe range. The layer is now waiting for the data to re-enter the user range. When that happens, the layer will automatically reconfigure to the user range and clear this bit.
- `Bit 1, DQ_DMM261_READ_STS_PROT_TRIP:` 1 = Protection tripped and disconnected the inputs. The layer will reconfigure the range and reconnect unless it’s already in the maximum range. This bit is cleared when the layer returns to the user range.
- `Bit 0, DQ_DMM261_READ_STS_PROT_TRIP_MAX`: 1 = Protection tripped in the max range. Inputs are disconnected until the layer is manually reconfigured by the user. To reset the protection, we recommend users read using a channel list that contains only -1:

  ```
  DqAdv261Read(hd, DEVN, 1, -1, NULL, NULL, NULL);
  ```

  and then reconnect by reading their existing channel list.

**Notes:**

- If you specify a short timeout delay, this function can time out when called for the first time. This is because the function is executed as a pending command, and layer programming takes up to 10ms.

- If your DMM-261 is connected to MUX layers, use `DqAdv261MuxGroupRead()` to safely read data. `DqAdv261Read()` will return a `DQ_FUNC_NOT_AVAILABLE` error if the DMM is part of a MuxGroup.

### *4.17.2 DqAdv261ReadStatus*

**Syntax:**

```
int DqAdv261ReadStatus(int hd, int devn, int SLSize,
uint32* slist, uint32* sData)
```

**Command:**

Read status data.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int SLSize` | Number of status channels to read |
| `uint32* slist` | Pointer to list of status channels |

**Output:**

| | |
|---|---|
| `uint32* sData` | Binary status data; (pass `NULL` if not required) |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not a DMM-261 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function is meant for UEI internal testing purposes. It can read all available hardware status data from the DMM-261 layer. Please contact technical support if your application requires low-level status data beyond what is returned by `DqAdv261Read()`.

`<slist>`:

The channel list is an array of status channels selected from the following table of `DQ_DMM261_GET_ST_x` constants.

| Channel Name | Description |
|---|---|
| `DQ_DMM261_GET_ST_GSTS` | ADC and relay status (`DQ_DMM261_GSTS_x`) |
| `DQ_DMM261_GET_ST_INT` | Current status of enabled interrupts |
| `DQ_DMM261_GET_ST_HEAT` | Reference heater status (remaining ON time) |
| `DQ_DMM261_GET_ST_ADCR` | Status of resistance measurement ADC (ADS1247) |
| `DQ_DMM261_GET_ST_GIMON` | Status of Guardian current monitor (AD7476) |

| | |
|---|---|
| `DQ_DMM261_GET_ST_G_ADC` | Status of Guardian voltage/temp monitor (LTC2486) |
| `DQ_DMM261_GET_ST_PGA` | Current status of the PGA access module including “data ready” and error status bits for each PGA (`DQ_DMM261_PGAERR_x`) |
| `DQ_DMM261_GET_ST_EXP_ST` | ADC and relay status; same as GSTS channel except sticky bits are not cleared. |

### *4.17.3 DqAdv261AutoZero*

**Syntax:**

```
int DqAdv261AutoZero(int hd, int devn, int CLSize,
uint32* clist, uint32 readings)
```

**Command:**

Auto-zero DCV and DCI channels.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Number of channels |
| `uint32* clist` | Pointer to channel list |
| `uint32 readings` | Number of seconds for zero value measurement (max 20) |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not a DMM-261 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function auto-zeros the DC channels in the channel list. It disconnects the inputs, takes an average zero value reading over the specified duration, and reconnects the inputs. These zero values will be subtracted from future measurements until `DqAdv261AutoZero()` is called again.

Channels are auto-zeroed in series, each for `readings` number of seconds. You can pass in the same channel list used by `DqAdv261Read()`, but auto-zero only affects DC voltage and DC current channels.

### *4.17.4 DqAdv261SetResRange*

**Syntax:**

```
int DqAdv261SetResRange(int hd, int devn, uint32 range,
uint32* limit)
```

**Command:**

Set the resistance mode and range.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 range` | Choose one of the following defined constants: |

```
DQ_DMM261_R_2W_10         //2-wire mode, 10 ohm range
DQ_DMM261_R_2W_100        //2-wire mode, 100 ohm range
DQ_DMM261_R_2W_1K         //2-wire mode, 1 kohm range
DQ_DMM261_R_2W_10K        //2-wire mode, 10 kohm range
DQ_DMM261_R_2W_100K       //2-wire mode, 100 kohm range
DQ_DMM261_R_2W_1M         //2-wire mode, 1 Mohm range
DQ_DMM261_R_2W_10M        //2-wire mode, 10 Mohm range
DQ_DMM261_R_2W_100M       //2-wire mode, 100 Mohm range
DQ_DMM261_R_2W_AUTORANGE  //2-wire mode, autorange

DQ_DMM261_R_4W_10         //4-wire mode, 10 ohm range
DQ_DMM261_R_4W_100        //4-wire mode, 100 ohm range
DQ_DMM261_R_4W_1K         //4-wire mode, 1 kohm range
DQ_DMM261_R_4W_10K        //4-wire mode, 10 kohm range
DQ_DMM261_R_4W_100K       //4-wire mode, 100 kohm range
DQ_DMM261_R_4W_1M         //4-wire mode, 1 Mohm range
DQ_DMM261_R_4W_10M        //4-wire mode, 10 Mohm range
DQ_DMM261_R_4W_AUTORANGE  //4-wire mode, autorange
```

| | |
|---|---|
| `uint32* limit` | Maximum resistance before over-range indicator is set |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not a DMM-261 |
| `DQ_NOT_IMPLEMENTED` | Firmware and PDNA library version are mismatched |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function configures the DMM-261 for resistance measurements. Range still needs to be programmed into the channel list in `DqAdv261Read()`for some checks the firmware makes.

**Notes:**

- The 100 MOhm range is only available in 2-wire mode.

### *4.17.5 DqAdv261SetConfig*

**Syntax:**

```
int DqAdv261SetConfig(int hd, int devn, uint32 item,
uint32 value, uint32* flags)
```

**Command:**

Configure the DMM-261.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 item` | Name of the parameter to be configured |
| `uint32 value` | Configuration value for `item` (pass `NULL` if not required) |
| `uint32* flags` | Reserved (pass `NULL`) |

**Output:**

| | |
|---|---|
| `uint32* flags` | Returned values for `item` |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not a DMM-261 |
| `DQ_BAD_PARAMETER` | `item` is invalid. |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function writes advanced configuration settings to the DMM-261 layer. The following table lists the user-configurable Items and the valid Values associated with each item.

| Item | Description | Value |
|---|---|---|
| **DC Averaging** | | |
| `DQ_DMM261_CONFIG_ V_ZC` | Set number of Voltage samples to average.<br>DC mode: Average is returned in `DQ_DMM261_VDC_AVG_DCS_CH`.<br>AC mode: The average is the default ACV zero crossing threshold when no valid AC waveform is detected. | `DQ_DMM261_ZCC_1_SAMPL`<br>`DQ_DMM261_ZCC_2_SAMPL`<br>`DQ_DMM261_ZCC_4_SAMPL`<br>`DQ_DMM261_ZCC_8_SAMPL`<br>`DQ_DMM261_ZCC_16_SAMPL`<br>`DQ_DMM261_ZCC_32_SAMPL`<br>`DQ_DMM261_ZCC_64_SAMPL`<br>`DQ_DMM261_ZCC_128_SAMPL`<br>`DQ_DMM261_ZCC_256_SAMPL` |
| `DQ_DMM261_CONFIG_ I_ZC` | Set number of Current samples to average.<br>DC mode: Average is returned in `DQ_DMM261_IDC_AVG_DCS_CH`.<br>AC mode: The average is the default ACI zero crossing threshold when no valid AC waveform is detected. | |
| **AC Zero Crossing Detection** | | |
| `DQ_DMM261_CONFIG_ V_ZC_MODE` | Configure automatic zero crossing detection for ACV. | `DMM261_ADCVI_ZCC_ZMD_ZMM //(Max+Min)/2`<br>`DMM261_ADCVI_ZCC_ZMD_ZRMS //DC average over the last period`<br>`DMM261_ADCVI_ZCC_ZMD_ZDC //DC average from AVG_DCS_CH`<br>To set a fixed zero crossing level, use `DqAdv261SetFixedZeroCrossing()`. |
| `DQ_DMM261_CONFIG_ I_ZC_MODE` | Configure automatic zero crossing detection for ACI. | |
| **NPLC Configuration** | | |
| `DQ_DMM261_CONFIG_ LINE_HZ` | Set power line frequency.<br>DC mode: Used with `RD_CYCLES` to reject AC noise.<br>AC mode: In special averaging mode, this value is unused.<br>In all other AC modes, this is used with `RD_CYCLES` to set a time limit for the RMS measurement. | Frequency in Hz<br>(default = 60, min = 25, max = 450) |
| `DQ_DMM261_CONFIG_ RD_CYCLES` | DC mode: Number of `LINE_HZ` cycles used per read (NPLC).<br>AC mode: In special averaging mode, this is the number of input signal periods used for the RMS calculation.<br>In all other AC modes, this is the number of `LINE_HZ` cycles used for the RMS measurement. | DC mode: maximum is 900 cycles @ 60Hz or 750 cycles @50Hz (15 seconds per read).<br>AC mode: maximum is 29 cycles @60Hz or 24 cycles @50Hz (0.5 seconds per read) |

| | | |
|---|---|---|
| **AC Peak Averaging** | | |
| DQ_DMM261_CONFIG_ V_PK_AVG | Set moving average used when reading AC voltage peak values. | Number of min/max readings to average:<br>`DQ_DMM261_AVG_1_SAMPL`<br>`DQ_DMM261_AVG_2_SAMPL`<br>`DQ_DMM261_AVG_4_SAMPL`<br>`DQ_DMM261_AVG_8_SAMPL`<br>`DQ_DMM261_AVG_16_SAMPL`<br>`DQ_DMM261_AVG_32_SAMPL`<br>`DQ_DMM261_AVG_64_SAMPL`<br>`DQ_DMM261_AVG_128_SAMPL`<br>`DQ_DMM261_PKAVG_DEFAULT     //64` |
| DQ_DMM261_CONFIG_ I_PK_AVG | Set moving average used when reading AC current peak values. | |
| **FIR Filters** | | |
| DQ_DMM261_CONFIG_ V_FIR | Configure digital FIR filter for voltage measurements. | Turn FIR filter off or select 1 of 4 cutoff frequencies:<br>`DMM261_CFG_VI_FIR_OFF   //FIR filter is bypassed`<br>`DMM261_CFG_VI_FIR_24HZ  //24Hz cutoff`<br>`DMM261_CFG_VI_FIR_50HZ  //50Hz cutoff`<br>`DMM261_CFG_VI_FIR_100HZ //100Hz cutoff`<br>`DMM261_CFG_VI_FIR_1KHZ  //1kHz cutoff` |
| DQ_DMM261_CONFIG_ I_FIR | Configure digital FIR filter for current measurements. | |
| **Input Protection** | | |
| DQ_DMM261_CONFIG_ DISCON_TIME | Sets the disconnect timeout. Inputs are disconnected if `DqAdv261Read()` has not been called in this amount of time. The next `DqAdv261Read()` call reconnects the inputs. | Time in seconds<br>(default = 10, min=1, max=250) |
| DQ_DMM261_CONFIG_ I_RESET | Resets the current range after an over-current range change. | n/a |

### *4.17.6 DqAdv261SetFixedZeroCrossingLevel*

**Syntax:**

```
int DqAdv261SetFixedZeroCrossingLevel (int hd, int devn,
uint32 mode, uint32 range, double zero_level)
```

**Command:**

Set a fixed zero-crossing level for AC measurements.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 mode` | The measurement mode of the channel being configured:<br>`DQ_DMM261_MODE_ACV_MEAS   //Set level for AC voltage`<br>`DQ_DMM261_MODE_ACI_MEAS   //Set level for AC current` |
| `uint32 range` | The range of the channel being configured:<br>`// AC voltage`<br>`DQ_DMM261_300VAC`<br>`DQ_DMM261_100VAC`<br>`DQ_DMM261_10VAC`<br>`DQ_DMM261_1_5VAC` |

```
DQ_DMM261_0_5VAC

// AC current
DQ_DMM261_RANGE_3_ADC
DQ_DMM261_RANGE_1_5_ADC
DQ_DMM261_RANGE_0_150_ADC
DQ_DMM261_RANGE_0_015_ADC
DQ_DMM261_RANGE_0_0015_ADC
```

| | |
|---|---|
| `double zero_level` | Desired zero crossing level (must be within the limits of the selected range) |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not a DMM-261 |
| `DQ_BAD_PARAMETER` | Invalid `mode`, `range`, or `zero_level` |
| `DQ_NOT_IMPLEMENTED` | Firmware and PDNA library version are mismatched |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function configures a user-defined zero crossing level for AC voltage or current measurements. Voltages for the RMS calculation are measured relative to this level.

**Notes:**

- When you call this function, the DMM’s zero crossing detection mode is automatically configured to use a fixed level. The setting in `DQ_DMM261_CONFIG_V_ZC_MODE` will be ignored.

### *4.17.7 DqAdv261MuxGroupCreate*

**Syntax:**

```
int DqAdv261MuxGroupCreate (int hd, int dmm_devn, int
mux_dev_count, int* reserved)
```

**Command:**

Link DMM-261 and MUX layers into a group.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int dmm_devn` | DMM-261 device number inside the IOM |
| `int mux_dev_count` | Number of MUX layers connected to the DMM (max 5) |
| `int* reserved` | reserved (pass in `NULL`) |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `dmm_devn` does not exist or is not a DMM-261 |
| `DQ_BAD_PARAMETER` | Invalid `mux_dev_count` |
| `DQ_FUNC_NOT_AVAILABLE` | DMM-261 is already part of a MuxGroup or group includes incompatible Mux layer(s) |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function creates a "MuxGroup" consisting of one DMM-261 and up to 5 MUX layers. The driver assumes that MUX layers are installed directly after the DMM layer in the chassis. A layer can only be in one MuxGroup. You can use `DqAdv261MuxGroupDestroy()` to release layers from an existing group.

**Notes:**

- When this function is called, all MUX layers in the group are automatically set to a disconnected state.

### *4.17.8 DqAdv261MuxGroupDestroy*

**Syntax:**

```
int DqAdv261MuxGroupDestroy (int hd, int dmm_devn)
```

**Command:**

Dissolve a MuxGroup.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int dmm_devn` | DMM-261 device number inside the IOM |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `dmm_devn` does not exist or is not a DMM-261 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
This function releases all DMM and MUX layers from the specified MuxGroup and resets MUX layers to a disconnected state.

**Notes:**

- If an application closes before calling `DqAdv261MuxGroupDestroy()`, you will have to call `DqAdv261MuxGroupDestroy()` at the start of the next application to release layers from the group.

### *4.17.9 DqAdv261MuxGroupDisconnect*

**Syntax:**

```
int DqAdv261MuxGroupDisconnect (int hd, int dmm_devn)
```

**Command:**

Disconnect Mux channel from the DMM-261.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int dmm_devn` | DMM-261 device number inside the IOM |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | Device indicated by `dmm_devn` does not exist, is not a DMM-261, or is not in a MuxGroup. |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
This function disconnects all input channels from the DMM. This is useful if you want to open/disable all MUX relays without connecting a new input channel (`DqAdv261MuxGroupSet`) or destroying the MuxGroup.

**Notes:**

### *4.17.10 DqAdv261MuxGroupSet*

**Syntax:**

```
int DqAdv261MuxGroupSet (int hd, int dmm_devn, int mux_devn, int
mux_channel_num, uint32 mux_relay_select, int dmm_cl_size,
uint32* dmm_cl, uint32* flags)
```

**Command:**

Select a Mux input channel and DMM mode.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int dmm_devn` | DMM-261 device number inside the IOM |
| `int mux_devn` | MUX device number inside the IOM |
| `int mux_channel_num` | MUX channel number to connect to DMM-261 |
| `uint32 mux_relay_select` | MUX relay to connect to DMM-261 |
| `int dmm_cl_size` | Number of entries in `dmm_cl` (max 64) |
| `uint32* dmm_cl` | DMM-261 channel list |
| `uint32* flags` | Reserved (pass in `NULL`) |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `dmm_devn` does not exist or is not a DMM-261, or `dmm_devn` and `mux_devn` are not in the same MuxGroup. |
| `DQ_BAD_PARAMETER_4` | Invalid `mux_relay_select`; 4-wire relay select is only allowed in 4-wire resistance mode. |
| `DQ_BAD_PARAMETER_5` | Invalid `dmm_cl_size` |
| `DQ_BAD_PARAMETER_6` | Invalid `dmm_cl`; all channels in the list should fall under the same DMM measurement mode. |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function safely configures MuxGroup relays for the specified MUX input channel and DMM measurement mode (VDC, VAC, IDC, IAC, R, or Guardian). Whenever you call this function with a new DMM mode or input channel, the API enforces break-before-make to ensure that only one input is connected to the DMM at a time.

`<mux_channel_num>` and `<mux_relay_select>`:

These two parameters together determine which MUX input channel is connected to the DMM. For the MUX-461, refer to `DqAdv461SetChannel() <channel_num>` and `<relay_select>` for valid options.

`<dmm_cl>`:

Refer to `DqAdv261Read() <clist>` for valid channel list options. `dmm_cl` can contain different channels from the same DMM mode (e.g., `DQ_DMM261_VDC_CAL_CH` and `DQ_DMM261_VDC_SAVG_CH`). However, you cannot mix and match channels from different DMM modes (e.g. `DQ_DMM261_VDC_CAL_CH` and `DQ_DMM261_VAC_RMS_CH`). If you want to make a different type of measurement, call `DqAdv261MuxGroupSet()`again with a new channel list.

**Notes:**

### *4.17.11 DqAdv261MuxGroupRead*

**Syntax:**

```
int DqAdv261MuxGroupRead(int hd, int dmm_devn, int* size,
uint32* bData, double* fData, uint32* status)
```

**Command:**

Read data from a DMM that is part of a MuxGroup.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int dmm_devn` | DMM-261 device number inside the IOM |

**Output:**

| | |
|---|---|
| `int size` | Number of entries in bData and fData (pass `NULL` if not required) |
| `uint32* bData` | Calibrated binary data (pass `NULL` if not required) |
| `double* fData` | Data converted to SI units (pass `NULL` if not required) |
| `uint32* status` | Status of input protection (pass `NULL` if not required) |

**Return:**

| | |
|---|---|
| `DQ_DATA_NOTREADY` | Data is not yet settled |
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `dmm_devn` does not exist, is not a DMM-261, or is not part of a MuxGroup. |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function reads all available data from a DMM-261 that is part of a MuxGroup. Before calling this function, select the desired input channel and DMM mode using `DqAdv261MuxGroupSet`(). To ensure that you are reading new data, you can confirm that the return value of the call is not `DQ_DATA_NOTREADY`.

Refer to `DqAdv261Read()` for a description of `bData`, `fData`, and `status` outputs.

**Notes:**

- The MUX input channel remains connected to the DMM until you call `DqAdv261MuxGroupSet()`, `DqAdv261MuxGroupDisconnect()`, or `DqAdv261MuxGroupDestroy()`.

## 4.18 DNx-AO-308/318/332/333/364 layers

### 4.18.1 DqAdv3xxWrite

**Syntax:**

```
int DqAdv3xxWrite(int hd, int devn, uint32 CLSize, uint32 *cl,
int takeraw, uint16 *data, double *fdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Channel list size |
| `uint32 *cl` | Channel list (channel numbers) |
| `int takeraw` | Use raw data instead of float |
| `uint16 *data` | Array of data (should be of CL size) or `NULL` if `takeraw` is `FALSE` |
| `double *fdata` | Array of float point voltages or `NULL` if `takeraw` is `TRUE` |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device number indicated by `devn` does not exist or is not AO-308 series, AO-318, AO-318-02x, AO-319-420, AO-332, AO-333, AO-364, TC-378 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, `takeraw` is `TRUE` and `data` is NULL, or `takeraw` is `FALSE` and `fdata` is NULL, or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

When this function is called for the first time, the firmware stops any ongoing operation on the device.

The firmware then parses the channel list and writes the passed values one by one accordingly.

The following output ranges are used for each product:

| | |
|---|---|
| • +/- 10 V output range: | AO-308, AO-308-350, AO-318, AO-332, AO-333 |
| • +/- 13.5 V output range: | AO-308-352 |
| • +/- 40 V output range: | AO-308-353 |
| • +/-0.125 V range: | TC-378 |
| • 0-20 mA output range: | AO-308-020, AO-318-020 |
| • 4-20 mA output range: | AO-319-420 |
| • 0-24 mA output range: | AO-318-024 |

Every write to the channel takes approximately 3.3µs. Thus, the execution time for this function depends on the number of channels in the channel list.

**Note:**

The user cannot perform this function call when the layer is involved in any streaming or data mapping operations.

This function works using an underlying `DqCmdIoctl()`. It uses a `DQCMD_IOCTL` command with a `DQIOCTL_CVTCHNL` function under the hood.

### 4.18.2 DqAdv3xxSetWForm

**Syntax:**

```
int DqAdv3xxSetWForm(int hd, int devn, uint32 Cfg, int CLSize,
uint32* cl, int samples, double UpdRate)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 Cfg` | Range, clocking and trigger settings for `DqCmdSetConfig()` call |
| `int CLSize` | Channel list size |
| `uint32 *cl` | Channel list array |
| `int samples` | Number of samples per channel |
| `double UpdRate` | Update rate , Hz |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device number indicated by `devn` does not exist or is not AO-308 series, AO-332 or AO-333 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, or a channel |

| | |
|---|---|
| | number in `cl` is too high |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets configuration for continuous waveform output
Use `DqAdvRoute...()` to configure synchronization interface to use with this function.

**Note:**

None.

### *4.18.3 DqAdv3xxWriteWFormCL*

**Syntax:**

```
int DqAdv3xxWriteWFormCL(int hd, int devn, int size, int from,
int takeraw, uint16* data, double* fdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int size` | Channel list size |
| `int from` | Write from a certain sample in the buffer |
| `int takeraw` | Use raw data instead of float |
| `uint16 *data` | Array of data (should be of CL size) or `NULL` if `takeraw` is `TRUE` |
| `double *fdata` | Array of float point voltages or `NULL` if `takeraw` is `FALSE` |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device number indicated by `devn` does not exist or is not AO-308 series, AO-332 or AO-333 |
| `DQ_BAD_PARAMETER` | `takeraw` is `TRUE` and `data` is NULL, or `takeraw` is `FALSE` and `fdata` is NULL |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |

| | |
|---|---|
| Other negative values | low level IOM error |

**Description:**

This function loads a repetitive waveform into the analog output layer memory. <`from`> parameters points to the location in the FIFO where the first of <`size`> samples should be written. Thus, this function allows replacement of only a segment of a waveform at a time. All channels has to be updated at the same time.
Use this function before calling `DqAdv3xxEnableWForm()` and then every time when waveform needs to be changed.

**Note:**

The sequence of the data points in the data should reflect the sequence of the channels in the channel list.
For example, if your channel list size `sl_size = 3` and `cl=[1,2,3],` the data should be represented like an array `data[i][cl_size]:` [1,2,3][1,2,3]..[1,2,3]

Maximum packet size limits the amount of data we can transfer.
This function must be called multiple times using the <`from`> parameter to append more samples to the waveform buffer.

### *4.18.4 DqAdv3xxEnableWForm*

**Syntax:**

```
int DqAdv3xxEnableWForm(int hd, int devn, int enable)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int enable` | Configure and start, pause, continue and stop operations |

```
#define DQ_AO3xx_STOP_WF     0
#define DQ_AO3xx_START_WF    1
#define DQ_AO3xx_PAUSE_WF    2
#define DQ_AO3xx_CONT_WF     3
```

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device number indicated by `devn` does not exist or is not AO-308 series, AO-332 or AO-333 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |

| | |
|---|---|
| Other negative values | low level IOM error |

**Description:**

This function configures, enables, pauses, resume or disables waveform output.

**Note:**

None.

### *4.18.5 DqAdv333ReadADC*

**Syntax:**

```
int DqAdv333ReadADC(int hd, int devn, int CLSize, uint32* cl,
uint32* bdata, double* fdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Channel list size |
| `uint32* cl` | Channel list (channel numbers) optionally with a timestamp request |

**Output:**

| | |
|---|---|
| `uint32* bdata` | Array of raw binary ADC data |
| `double* fdata` | Array of float point voltages |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-333 |
| `DQ_BAD_PARAMETER` | channel number in channel list is not a valid AO-333 channel number |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function requests conversion results for each channel specified in the channel list and stores them alongside converted data.

It will take a little under 3 seconds for the AO-333 to make all 32 readings. If this function is called at a rate that does not allow enough time for conversion to occur, then old or invalid data may be returned. New data for any ADC channel is indicated by the presence of a ‘1’ in the least significant bit of the `bdata` for that channel.

Because the ADC readings are started when a changed channel list is presented, the data from a first read with a new channel list must be discarded.

## 4.19 DNx-AO-318/AO-318-02x/AO-319-420 layer

See `DqAdv3xxWrite()` for writing outputs. The following output ranges are used for each product:

| | |
|---|---|
| - +/- 10 V output range: | AO-318 |
| - 0-20 mA output range: | AO-318-020 |
| - 4-20 mA output range: | AO-319-420 |
| - 0-24 mA output range: | AO-318-024 |

This document will refer to the AO-318-020 and AO-318-024 collectively as the AO-318-02x.

### 4.19.1 DqAdv318CBStatus

**Syntax:**

```
int DqAdv318CBStatus(int hd, int devn, int chan_mask, uint32 flags,
uint32 reset, uint32* status)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chan_mask` | Bitmask of channels to apply parameters to |
| `uint32 flags` | Additional behavior modifiers <reserved> |
| `uint32 reset` | Bitmask what CBs to reset |

**Output:**

| | |
|---|---|
| `uint32* status` | Status of tripped CBs. `status[0]` = main status register, `status[1]` and greater contain status of channels enabled by `chan_mask`, reported in ascending order |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-318, AO-318-02x, AO-319-420 or TC-378 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function resets circuit breakers defined by the bitmask parameter `uint32 reset` and returns circuit breaker and ADC status.

- Bit definitions for `status[0]` may be found in powerdna.h, search key `DQ_AO318_STS_CB7_CS`

- Bit definitions for `status[1]`and greater may also be found in powerdna.h, search key `DQ_AO318_CSTS_DACBSY`

**Note:**

One of the main uses of this function is to retrieve the status of the circuit breakers. If you are not interested in any of the other status information, you have the option of retrieving status from data returned with the `DqAdv318ReadADC()` API instead.

For AO-318, AO-318-02x, AO-319-420 and TC-378 boards with layer logic version 02.14.15 or higher, bit position 30 of `*bdata` returned by the `DqAdv318ReadADC()` API indicates the circuit breaker status (1 is tripped / 0 is not tripped).

### *4.19.2 DqAdv318ReadADC*

**Syntax:**

```
int DqAdv318ReadADC(int hd, int devn, int CLSize, uint32* cl,
uint32* bdata, double* fdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Channel list size |
| `uint32* cl` | Channel list |

**Output:**

| | |
|---|---|
| `uint32* bdata` | 32-bit raw binary data (NULL if not required) |
| `double* fdata` | Converted data (NULL if not required) |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-318, AO-318-02x or AO-319-420 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function requests conversion results for each channel specified in the channel list and stores them alongside converted data.

`<cl>`: Use the `DQ_AO318_MAKE_CL(CH,SUBCH)` helper macro to combine the channel and ADC diagnostic subchannel to make the entries in the `cl` channel list.

- The AO-318 has 8 analog output channels with 5 diagnostic ADC measurements (or subchannels) available for each output channel. Subchannels are 0: Iin; 1: VAint; 2: VBint; 3: VABext; 4: ADC on-die temperature.
- The AO-319 has 8 current output channels with 3 diagnostic ADC measurements (subchannels) available for each output channel. Subchannels are 0: Iin; 1: Vext; 2: ADC on-die temperature.

There is a maximum of 60 entries in the channel list. (not including the optional timestamp request).

The rate that the diagnostic ADC reads diagnostic subchannels is slow.

- The default CV clock for the AO-318 is 5 Hz, resulting in all 5 monitor subchannels for all 8 analog output channels updating every second.
- The default CV clock for the AO-319 is 7.5 Hz, resulting in all 3 monitor subchannels for all 8 analog output channels updating approximately every half second.

The sample rate can be reset with the `DqAdv318SetConfig()` API (UEI recommends sampling at the default rate; performance accuracy degrades at higher (> 10 Hz) sample rates).

`<bdata>`: Valid data flag/circuit breaker state/diagnostic channel ID are encoded in `bdata`:

- **Bit 31**: The most significant bit of the raw data (`bdata`) will be a 1 if that reading has been updated since the last read. If the data has not been updated, the previous reading will be returned. The ADC starts running automatically upon first call to `DqAdv3xxWrite()` or entering the board into an operating mode (ACB, DMAP, VMAP, AVMAP) .
- **Bit 30:** For AO-318, AO-319 and TC-378 boards with layer logic version 02.14.15 or higher, bit position 30 indicates state of circuit breaker (1 if currently tripped).
- **Bits 27..25** of the `bdata` indicate which of the ADC channels produced the data. The macro `DQ_AO318_CH_ID(S)` can be used to extract the channel numbers ($I_{in}$, $V_a$, $V_b$, $V_{out}$, T). Because custom settings for the circuit breaker function may sometimes change the order in which the ADC channels are reported, it is always a good idea to pull the channel numbers out of the `bdata` values using the `DQ_AO318_CH_ID` macro instead of relying on the default order of the channels.
- The `DqAdv318CBStatus()` API is used to read current CB status and/or re-enable it.

**Note:**

If custom circuit breaker settings cause an ADC channel to be unused it will be reported as `DQ_AO318_ADC_CH_UNUSED` in bits 27..25 of `bdata`.

You can use the following #defines to determine the ADC value returned from the `DQ_AO318_CH_ID(S)`:

Defines for AO-318:

```
DQ_AO318_ADC_CH_I      // output current (over 0.1 Ohm shunt)
DQ_AO318_ADC_CH_A_int  // voltage on the output A at the buffer
DQ_AO318_ADC_CH_B_int  // voltage on the output B at the buffer
DQ_AO318_ADC_CH_AB_ext // voltage on output after the relays
DQ_AO318_ADC_CH_TEMP   // temperature channel, degrees C
DQ_AO318_ADC_CH_UNUSED // channel is unused
```

Defines for AO-318- 02x:

```
DQ_AO318_020_ADC_CH_I     // output current (over 10 Ohm shunt)
DQ_AO318_020_ADC_CH_INT_1 // internal buffer voltage 1
DQ_AO318_020_ADC_CH_INT_2 // internal buffer voltage 2
DQ_AO318_020_ADC_CH_EXT   // voltage on output after the relay
DQ_AO318_ADC_CH_TEMP      // temperature channel, degrees C
DQ_AO318_ADC_CH_UNUSED    // channel is unused
```

The #defined constants are listed in powerdna.h.

This function works using an underlying `DqCmdIoctl().` It uses a `DQCMD_IOCTL` command with a `DQIOCTL_CVTCHNL` function called in the API.

### 4.19.3 DqAdv318Reengage

**Syntax:**

```
int DqAdv318Reengage(int hd, int devn, uint32 reset)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 reset` | Bits 7:0, reset the breakers for the corresponding channels |

**Output:**

`-none-`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-318, AO-318-02x, AO-319-420, or TC-378 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function resets the circuit breakers for the channels where a '1' is found in the bit position corresponding to the channel number of the AO-318, AO-318-02x, and AO-319-420. If the ADC readings are still beyond the circuit breaker trip limits (set by `DqAdv318SetCBLevels()`), the breaker will trip again.

If reset is zero, the function will do nothing and return `DQ_SUCCESS`.

**Note:**

### 4.19.4 DqAdv318SetCBLevels

**Syntax:**

```
int DqAdv318SetCBLevels(int hd, int devn, int chan_mask,
pDQAO318BRK_CFG cblev)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chan_mask` | Bitmask of channels to apply parameters to |
| `pDQAO318BRK_CFG cblev` | Parameters |

**Output:**

`- none -`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-318, AO-318-02x, AO-319-420 or TC-378 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets the circuit breaker trip levels.

As an example, the following code sets `MINMAX_0` to +/- 11 mA, `MINMAX_1` to +/- 12 V and `MINMAX_2` to +/- 11V on AO channels 0 and 1:

```
DQAO318BRK_CFG my_cfg;
chan_mask = 0x03;       // channels 0 and 1

my_cfg.CB_val_min_f[0] = -11.0;   // mA
my_cfg.CB_val_max_f[0] = 11.0;
my_cfg.units[0] = DQAO318_CB_UNIT_I;  // current

my_cfg.CB_val_min_f[1] = -12.0;   // Volts
my_cfg.CB_val_max_f[1] = 12.0;
my_cfg.units[1] = DQAO318_CB_UNIT_V;  // volts

my_cfg.CB_val_min_f[2] = -11.0;   // Volts
my_cfg.CB_val_max_f[2] =  11.0;
my_cfg.units[2] = DQAO318_CB_UNIT_V;  // volts

DqAdv318SetCBLevels(hd, devn, chan_mask,&my_cfg);
```

The `DqAdv318SetCBLevels()` function calls the `DqAdv318SetConfig()` function to set the levels and is a simplified method for setting limits.

Please refer to the powerdna.h header file for details of the `DQAO318BRK_CFG` structure.

**Note:**

### *4.19.5 DqAdv318SetConfig*

**Syntax:**

```
int DqAdv318SetConfig(int hd, int devn, int chan_mask, int flags,
pDQAO318CFG cfg)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chan_mask` | Bitmask of channels to apply parameters to |
| `int flags` | Additional behavior modifiers <reserved> |
| `pDQAO318CFG cfg` | Structure of configuration parameters |

**Output:**

`- none -`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-318, AO-318-02x or AO-319-420 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets various parameters, configuring circuit breakers and ADC functionality.

The `chan_mask` parameter determines the channels to which the parameters are applied. A ‘1’ in the corresponding bit position enables the parameters for that channel, (e.g. 0x03 applies parameters to channels 0 and 1 only, 0xFF applies parameters to all 8 channels).

Parameters are set using the `DQAO318CFG structure` as follows:

```
typedef struct {
    uint32 prmmask;                  // bitmask: which parameters are valid
                                     //     and need to be set
    uint32 en_DAC;                   // select outputs A (1) or B (2) or
                                     //     both (3) or neither(0)
```

```
        uint32 ADC_CL[DQ_AO318_ADC_CH];    // ADC channel list
        float  ADC_rate;                   // rate of ADC converter if external
                                           //    clock is used
        uint32 CB_val_min[DQ_AO318_CB_CH]; // CB minimum limit value,
                                           //    code 0..ffff
        uint32 CB_val_max[DQ_AO318_CB_CH]; // CB maximum limit value,
                                           //    code 0..ffff
        uint32 CB_mode;                    // CB source (channel) and
                                           //    reenabling mode
        uint32 rdcnt;                      // Number of consecutive ‘past limit’
                                           //     ADC reads that need to occur
                                           //     before breaker trips
        float  CB_rate;                    // CB reenabling rate
    } DQAO318CFG, *pDQAO318CFG;
```

The `prmmask` member of the `DQAO318CFG` structure is a bitfield that indicates which of the other members of the struct are valid and need to be set.

Four `prmmask` bits are defined:

```
#define DQ_AO318_CBCFG_DAC          (1L<<0)
#define DQ_AO318_CBCFG_SETADCCL     (1L<<1)
#define DQ_AO318_CBCFG_SETCB        (1L<<2)
#define DQ_AO318_CBCFG_SETCBSRC     (1L<<3)
```

- `DQ_AO318_CBCFG_DAC` indicates that `en_DAC` is valid.
- `DQ_AO318_CBCFG_SETADCCL` indicates that `ADC_CL` and `ADC_rate` valid. Please note that the channel list for the ADC’s will be the same for all channels, `chan_mask` does not apply.
- `DQ_AO318_CBCFG_SETCB` indicates that `CB_val_min` and `CB_val_max` are valid. Note that this setting is normally only used internally by the `DqAdv318SetCBLevels()` function.
- `DQ_AO318_CBCFG_SETCBSRC` indicates that `CB_mode`, `rdcnt` and `CB_rate` are valid.

**Note:**

Please refer to the powerdna.h file for more detail of the `DQAO318CFG` struct.

### *4.19.6 DqAdv319WriteDiag*

**Syntax:**

```
int DqAdv319WriteDiag(int hd, int devn, uint32 ch_open, uint32*
reserved)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 ch_open` | ‘1’ in bit position to open channel, ‘0’ for normal operation |
| `uint32* reserved` | Reserved for future use, set to NULL |

**Output:**

- none -

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-319-420 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function allows the user to turn off a channel output to simulate an open device. To open a channel, set its corresponding bit in <ch_open> to “1’ (i.e CH0 corresponds to bit 0, CH1 corresponds to bit 1, etc.). Set the bit value to ‘0’ for normal operation.

## *4.20 DNA-AO-358 layer*

### *4.20.1 DqAdv358ExCalAccess*

**Syntax:**

```
int DqAdv358ExCalAccess(int hd, int devn, uint32 cmd, uint32
len, uint32 addr, uint8 *data)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 cmd` | access command constant, see below |
| `uint32 len` | length of read or write, 1Kb max read, 256 max write |
| `uint32 addr` | Address of data to read or write, used by the DQ_AO358_EE_RD_ID_ADDR and DQ_AO358_EE_WR_OPEN commands, ignored by all other commands |
| `uint8 *data` | array of data in or out |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-358 |
| `DQ_BAD_PARAMETER` | `cmd` is not one of the defined constants below or `len` is less than or equal to zero or greater than 1024 for read or greater than 256 for write. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function is used to retrieve the extended calibration data for the AO-358.

This function works using an underlying `DqCmdIoctl()`. It uses a `DQCMD_IOCTL` command with a `DQIOCTL_CVTCHNL` function under the hood.

**Command defines**:

DQ_AO358_EE_CHK_STS - get the status, see below.

DQ_AO358_EE_RD_ID_ADDR - sets the address for the subsequent read commands, gets 8-bit silicon ID to `data` [0]

DQ_AO358_EE_RD - read from the device to *`data` 1Kb max. Sequential reads will return contiguous data from the device.

DQ_AO358_EE_WR_OPEN - open the device for writing, sets the write address, removes write protect.

DQ_AO358_EE_ERASE - erase the EPCS device. Device must be open for writing for erase to succeed. The erase cycle takes approx 10 seconds. Busy flag will be nonzero during erase cycle and must be polled to determine when command is complete.

DQ_AO358_EE_WR - write *`data` to device, 256 bytes max. Sequential write commands will write contiguous data into the device.

DQ_AO358_EE_WR_CLOSE - close device to writing, sets write protection.

**Status return:**

The status from the DQ_AO358_EE_CHK_STS command is returned as 5 32-bit values using *data cast to a uint32*.

data[0] is the busy flag.

data[1] is the AO358_ESTS value, see logic document for details

data[2] is the RDSTS value from the EPCS device. This value will only be returned when the busy flag is zero. The msbit will be set when the value is invalid.

value[3] is the read address that the next DQ_AO358_EE_RD command will use.

value[4] is the write address that the next DQ_AO358_EE_WR command will use.

**Note:**

Internally, the busy flag does toggle to the busy state briefly after every RD and WR command, but it is not necessary for the user to test the state of the busy flag for the RD and WR commands. The firmware will automatically test for busy and delay the command for the appropriate time.

The DQ_AO358_EE_RD_ID_ADDR command sets the read address and DQ_AO358_EE_WR_OPEN sets the write address. As long as sequential addresses are being read or written, then multiple RD or WR commands may be sent without re-sending the addresses. Read and write addresses are stored separately, so read and write may be interleaved if desired.

### *4.20.2 DqAdv358Write*

**Syntax:**

```
int DqAdv358Write(int hd, int devn, uint32 CLSize, uint32* cl,
int takeraw, uint32* data, double* fdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 CLSize` | channel list size |
| `uint32 cl` | channel list (channel numbers) |
| `int takeraw` | use raw data instead of float |
| `uint32* data` | array of raw binary data (should be of CL size) or `NULL` if `takeraw` is `FALSE` |
| `double* fdata` | array of float point resistances or `NULL` if `takeraw` is `TRUE` |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-358 |
| `DQ_BAD_PARAMETER` | channel number in channel list is not a valid AO-385 channel number. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function is used to write either floating point or raw values to the AO-358.

This function works using an underlying `DqCmdIoctl()`. It uses a `DQCMD_IOCTL` command with a `DQIOCTL_CVTCHNL` function under the hood.

When this function is called for the first time, the firmware stops any ongoing operation on the device.

Then, the firmware parses the channel list and writes the passed values one by one accordingly.

**Note:**

None.

### *4.20.3 DqAdv358ReadADC*

**Syntax:**

```
int DqAdv358ReadADC(int hd, int devn, int CLSize, uint32* cl,
uint32* bdata, double* fdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |

| | |
|---|---|
| `int devn` | Layer inside the IOM |
| `int CLSize` | channel list size |
| `uint32* cl[0]` | channel list (channel numbers) |

**Output:**

| | |
|---|---|
| `uint32* bdata` | array of raw binary ADC data |
| `double* fdata` | array of float point voltages or currents |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-358 |
| `DQ_BAD_PARAMETER` | channel number in channel list is not a valid AO-385 channel number. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function requests conversion results for each channel specified in the channel list and stores them alongside converted data.

At each channel, each ADC reading takes approx 0.16 seconds. If all 5 channels are to be read, then it will take 0.8 seconds to make all 5 readings. If this function is called at a rate that does not allow enough time for conversion to occur, then old or invalid data may be returned. New data is indicated by the presence of a '1' in the MSBit of the `bdata` .

The DQ_AO358_MAKE_CL(CH,SUBCH) macro should be used to make the contents of the channel list. The AO-358 has 8 channels with each channel having 5 ADC channels, also known as subchannels. The range of the channels is 0 thru 7 and the range of the ADC subchannels is 0 thru 4 as shown below.

```
DQ_AO358_SUBCH_I_SENSE      (0)    // current in adjustable bridge arm
DQ_AO358_SUBCH_EX1          (1)    // excitation voltage (normally positive)
DQ_AO358_SUBCH_EX2          (2)    // excitation voltage (normally negative)
DQ_AO358_SUBCH_VS_N         (3)    // voltage at the S- bridge point
DQ_AO358_SUBCH_THERM        (4)    // temperature of the ADC in degrees C
```

The entire ADC subsystem is normally OFF at power-up. An ADC converter is started when a valid channel list entry for that ADC is sent by this function. The ADC will continue to run until a channel list is presented by this function that does not contain any entries for that channel. To shut off all ADCs for a given device number, send a read command with this function with `CLSize` set to 1 and `cl[0]` set to 0xFFFFFFFF .

Because the ADCs are started when a changed channel list is presented, the data from a first read with a new channel list must be discarded.

## 4.21 DNA-AO-364 layer

### 4.21.1 DqAdv364Enable

**Syntax:**

```
int DqAdv364Enable(int hd, int devn, int enable_mask, int
out_enable_mask)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int enable_mask` | Bit mask: "1" in the channel bit position enables channel |
| `int out_enable_mask` | Bit mask: "1" means the channel is active driver,<br>"0" puts it into high impedance state |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-358 |
| `DQ_BAD_PARAMETER` | `cmd` is not one of the defined constants below or `len` is less than or equal to zero or greater than 1024 for read or greater than 256 for write. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This enables or disables selected channels and enable/disables outputs.

**Note:**

This function can be used to control outputs. If a channel is enabled and operational, user can turn channel from being a driver into high-impedance state and back without stopping operations.

If `DqAdv364Write()` was used to output a DC offset, then the channel must be disabled and re-enabled to output a programmed waveform again.

### 4.21.2 DqAdv364SelectAWF

*Not Implemented.*

**Syntax:**

```
int DqAdv364SelectAWF(int hd, int devn, int channel, int mode,
int buffer, pDQAO364WFPRM wf_prm, uint32* status)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | channel to config |
| `int mode` | modifiers <reserved> |
| `int buffer` | new waveform buffer (from allocated in DqAdv364SetAWF |
| `pDQAO364WFPRM wf_prm` | a set of new parameters to apply |
| `uint32* status` | WF generation status and current buffer [0..63] |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-358 |
| `DQ_BAD_PARAMETER` | `cmd` is not one of the defined constants below or `len` is less than or equal to zero or greater than 1024 for read or greater than 256 for write. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function switches to the new buffer with loaded AWF and applies new parameters. If applied to a current buffer it changes parameters.

**Note:**

This function can be used to control a pre-loaded waveform by switching buffers and applying changes to the waveform shape.

### 4.21.3 DqAdv364SetAWF

*Not Implemented.*

**Syntax:**

```
int DqAdv364SetAWF(int hd, int devn, int channel, int mode, int
n_buffers, pDQAO364BUFLST* buf_lst)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | channel to config |
| `int mode` | modifiers <reserved> |
| `int n_buffers` | number of buffers per channel |
| `int* buf_list` | list of the buffer sizes |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-358 |
| `DQ_BAD_PARAMETER` | `cmd` is not one of the defined constants below or `len` is less than or equal to zero or greater than 1024 for read or greater than 256 for write. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function removes any previous allocation (if any) and creates a new list of buffers for use with AWF function.

**Notes:**

Buffer list has encoded information about the size of each buffer (must be in multiple of 128-sample blocks)

```
typedef struct {
 uint32 buf_size;        // the size of the buffer, bytes (multiple of 128)
 uint32 d_size;          // the size of the data in the buffer
 uint32 buf_next;        // AO364_AUTONEXT flag and next buffer ID for linked list ops.
 uint32 flags;           // buffer control flags
} DQAO364BUFLST, *pDQAO364BUFLST;
```

The maximum number of buffers is 64.

If buffer is used in single-shot mode the size `<d_size>` should reflect the actual size of the data. Output will stop when `<d_size>` is reached.

Flags can include a countdown value. Each time the buffer is processed the value is decremented by 1 and output stops when the counter is exhausted. If the countdown flag is not set then the output will continue through the chain of buffers indefinitely. Flags can also inlclude an event bit to send an asynchronous event notification when this buffer is completed.

### 4.21.4 DqAdv364SetBaseClocks

*Not Implemented.*

**Syntax:**

```
int DqAdv364SetBaseClocks(int hd, int devn, int mode, int chan,
int source1, double f_PLL1, double* true_PLL1, int source2,
double f_PLL2, double* true_PLL2, int source_o, double*
true_offs_f)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int mode` | behavior flags |
| `int chan` | channel |
| `int source1` | select 1st clock source |
| `double f_PLL1` | select 1st PLL frequency |
| `double* true_PLL1` | actual PLL1 frequency |
| `int source2` | select 2nd clock source |
| `double f_PLL2` | select 2nd PLL frequency |
| `double* true_PLL2` | actual PLL2 frequency |
| `int source_o` | select offset DAC clock |
| `double* true_offs_f` | actual offset DAC frequency |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-358 |
| `DQ_BAD_PARAMETER` | `cmd` is not one of the defined constants below or `len` is less than or equal to zero or greater than 1024 for read or greater than 256 for write. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function allows user to overwrite automatically selected PLL frequencies for output waveforms and/or retrieve true frequencies to check for tolerances.

**Note:**

This is extended function which is used for more precise control of the sampling clocks used to output waveform.

### *4.21.5 DqAdv364SetChannelPll*

*Not Implemented.*

**Syntax:**

```
int DqAdv364SetChannelPll(int hd, int devn, int channel, int
pll, double samplerate, double* act_rate)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | channel to config |
| `int pll` | PLL number (1 thru 4) |
| `double samplerate` | desired sample rate |
| `double* act_rate` | returned actual rate |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-358 |
| `DQ_BAD_PARAMETER` | `cmd` is not one of the defined constants below or `len` is less than or equal to zero or greater than 1024 for read or greater than 256 for write. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Calculates and sends setup values for use by on-layer PLL.

**Note:**

This function has no implementation in this release.

### *4.21.6 DqAdv364SetConfig*

**Syntax:**

```
int DqAdv364SetConfig(int hd, int devn, int channel, uint32
clk_src, uint32 trig_src, uint32 sync_mode)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | channel to config |
| `uint32 clk_src` | bitfield: clock source for the channel |
| `uint32 trig_src` | bitfield: trigger source for the channel |
| `uint32 sync_mode` | synchronization mode |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-358 |
| `DQ_BAD_PARAMETER` | `cmd` is not one of the defined constants below or `len` is less than or equal to zero or greater than 1024 for read or greater than 256 for write. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets configuration for a single channel of AO-364 layer.

**Notes:**

A few modes of clocking and triggering are available.
A channel can work as individual or synchronized with one channel located on the same board/cube/rack or on another cube/rack connected with a cable.

If channels are to be used individually any clock can be used.
All start triggers are triggered on the rising edge.
There is no stop trigger but a trigger can be used in gate mode (high gates DAC clock).

If the phase of the signals needs to be syncrhonized channel 0 on one of the board should become a clock source for all other channels including itself.
For a single board use `DQ_AO364CFG_CLK_CH0` for all channels `<mode>` and `DQ_AO364CFG_CLKOUT_CH0` for `<syncmode>` of channel 0. For multile boards use the `..SYNC0/..SYNC2` option. Set trigger for channel 0 only since this channel will gate to clock to all other channels.

Use `DQ_AO364CFG_CLK_...` constants for clock
Use `DQ_AO364CFG_TRG_...` constants for trigger
Use `DQ_AO364CFG_CLKOUT_...` constants for setting synchornization mode

### *4.21.7 DqAdv364SetDIO*

**Syntax:**

```
int DqAdv364SetDIO(int hd, int devn, int channel, uint32
dio_cfg, uint32* status)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | channel to config |
| `uint32 dio_cfg` | DIO lines configuration |

**Output:**

| | |
|---|---|
| `uint32* dio_state` | current DIO levels among other status bits |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-358 |
| `DQ_BAD_PARAMETER` | `cmd` is not one of the defined constants below or `len` is less than or equal to zero or greater than 1024 for read or greater than 256 for write. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function set direction and source for DIO on the channel.

**Notes**:

Use `AO364IS_SYNC_DIO()` macro and `AO364IS_DIO_SW_...` constants to set up DIO lines
Use `AO364IS_STS_...` bit definitions to decode status.

### *4.21.8 DqAdv364SetOffsWF*

*Not Implemented.*

**Syntax:**

```
int DqAdv364SetOffsWF(int hd, int devn, int channel, int config,
pDQAO364WFPRM param)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | channel to configure |
| `int config` | configuration flags |
| `double freq` | sampling frequency |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-358 |
| `DQ_BAD_PARAMETER` | `cmd` is not one of the defined constants below or `len` is less than or equal to zero or greater than 1024 for read or greater than 256 for write. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up configuration for Offset DAC.

```
typedef struct {
    uint32 flags;      // configuration flags (what to set)
    double freq;       // frequency
    double span;       // waveform span
    double offset;     // waveform offset
    double phase;      // waveform phase
    double set_time;   // time to apply parameters
} DQAO364WFPRM, *pDQAO364WFPRM;
```

**Note:**

Each channel has two DACs - main DAC which generates WF and offset DAC which generates a DC offset or a waveform to add to the main waveform. Offset DAC is an 18-bit DAC with an ability to extend to 20 bits.

In normal operation the offset DAC is used either to select an offset of the waveform (in `DqAdv364SetWFParameters`) or generate a wave function to add to the main DAC.

Offset DAC is slower (1MS/s) than the main DAC (16.5MS/s).
This function applies new parameters immediately.
If also can stop and start the Offset DAC clock.

### *4.21.9 DqAdv364SetWF*

**Syntax:**

```
int DqAdv364SetWF(int hd, int devn, int channel, int mode,
pDQAO364WFPRM wf_prm, pDQAO364STDWF wf_shape)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | channel number |
| `int mode` | mode of operation |
| `pDQAO364WFPRM wf_prm` | waveform parameters |
| `pDQAO364STDWF wf_shape` | additional waveform shape information or NULL |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-358 |
| `DQ_BAD_PARAMETER` | `cmd` is not one of the defined constants below or `len` is less than or equal to zero or greater than 1024 for read or greater than 256 for write. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This call applies waveform parameters for the channel
`<mode>` defines which of the standard waveforms is to output and how it is synthesized

| | |
|---|---|
| `DQ_AO364CFG_WF_PLL` | use PLL mode of operation |
| `DQ_AO364CFG_WF_DSS` | use DSS (digiral signal synthesis mode) |
| `DQ_AO364CFG_WF_INVERT` | apply inversion function to the waveform |
| `DQ_AO364CFG_WF_MIRROR` | apply mirror function to the waveform |
| `DQ_AO364CFG_WF_PULSE` | standard WF is a pulse traingle |
| `DQ_AO364CFG_WF_RAMP` | standard WF is a ramp |
| `DQ_AO364CFG_WF_SAW` | standard WF is a saw tooth |
| `DQ_AO364CFG_WF_SINE` | standard WF is a sinewave |

```
typedef struct {
    uint32 flags;     // configuration flags (what to set)
    double freq;      // frequency
    double span;      // waveform span
    double offset;    // waveform offset
    double phase;     // waveform phase
    double set_time;  // time to apply parameters
} DQAO364WFPRM, *pDQAO364WFPRM;
```

**Notes:**

This function can apply parameters immediately (`set_time==0`) or gradually within the programmed `<set_time>`.
For example, this function can be used to gradually vary the phase of the waveform or perform a gradual attenulation of it. The waveform parameters always change in linear fashion during specified time.
Flags can provide additional information whether to change parameters continuously (ramp or triangle) or apply them only once. The firmware takes care of generating and feeding parameter channel list to DACs.

Standard waveforms can be synthesised in two ways:

PLL mode: AO-364 has two available digital PLLs for each channel. PLL can be programmed to generate quantization clock within 1Hz accuracy. When the user changes the frequency a second PLL is programmed to the new quantization frequency and switchover occurs at the beginning of the next waveform period. This mode gives the lowest possible THD (-84dB) but requires 500ms to switch the frequency. The sample waveform has 4096 points and should the quantization frequency fall below 4MHz additiona points for the waveform sample are generated using linear interpolation. Should the required frequency exceed 32MHz the points are dropped from the waveform.

DSS mode: Quantization frequency is generated from the fixed PLL clock and NCO (numerical control oscillator) is used in conjunction with FCW (frequency code word) to select a proper point in the sample waveform. This technique allows immediate change in the waveform frequency at the expence of the lower THD (-72dB) and the fact that selected waveform points are different from one period to another.

### *4.21.10 DqAdv364SetWFParametersCL*

**Syntax:**

`int DqAdv364SetWFParametersCL(int hd, int devn, int mode, int clsize, int* CL, pDQAO364WFPRM wf_prm)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int mode` | additional flags <reserved> |
| `int clsize` | size of the following CL array |
| `int* CL` | a list of channels to apply parameters to |
| `pDQAO364WFPRM* wf_prm` | array of waveform parameters |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-358 |
| `DQ_BAD_PARAMETER` | `cmd` is not one of the defined constants below or `len` is less than or equal to zero or greater than 1024 for read or greater than 256 for write. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This call applies waveform parameters for the channel specified in the channel list. Unlike `DqAdv364SetWFParameters()`, this function applies all parameters to channels specified in the channel list simutaneously.

**Notes:**

One exception applies: this function doesn't reload waveform shape but rather controls amplitude, phase, etc. on all listed channels. Use `DqAdv364SetWF` to select waveform type.

### *4.21.11 DqAdv364SetWFSweep*

**Syntax:**

```
int DqAdv364SetWFSweep(int hd, int devn, int channel, int mode,
pDQAO364WFSWEEP wf_prm)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | channel number |
| `int mode` | select waveform |
| `pDQAO364WFSWEEP wf_prm` | sweep parameters |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-358 |
| `DQ_BAD_PARAMETER` | `cmd` is not one of the defined constants below or `len` is less than or equal to zero or greater than 1024 for read or greater than 256 for write. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This call switches the waveform into sweep mode.

```
typedef struct {
    uint32 flags;      //configuration flags (form of the sweep, once or continuous)
    double start_freq;      // frequency
    double end_freq;        // final frequency
    double sweep_time;      // the length of sweel (>= 1ms)
} DQAO364WFSWEEP, *pDQAO364WFSWEEP;
```

**Note:**

None.

### *4.21.12 DqAdv364Write*

*See DqAdv3xxWrite.*

### *4.21.13 DqAdv364WriteAWF*

**Syntax:**

```
int DqAdv364WriteAWF(int hd, int devn, int channel, int mode,
int buffer, int from, int size, uint16* data, double* fdata,
uint32* status)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | channel to write data to |
| `int mode` | modifiers |
| `int buffer` | buffer ID (created in DqAdv364SetAWF()) |
| `int from` | write from a certain sample in the buffer |
| `int size` | data size in samples |
| `uint16* data` | array of data (should be of CL size) or NULL if takeraw is TRUE |
| `double* fdata` | array of float point voltages or NULL if takeraw is FALSE |
| `uint32* status` | WF generation status and current buffer [0..63] |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-358 |
| `DQ_BAD_PARAMETER` | `cmd` is not one of the defined constants below or `len` is less than or equal to zero or greater than 1024 for read or greater than 256 for write. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function writes waveform to AO-364 waveform buffer
The waveform size can be anything for up to 2MS of data (it will take some time to load since we can process 1k of data each millisecond).

<mode> accepts two flags:

`DQL_AO364_WRITE_USEDBL` - to use double values from `<f_data>` and convert into uint16

`DQL_AO364_WRITE_STDWF` - write to SWF area instead of AWF (fixed 4096 size, a waveform of a different length must be stretched/compressed to this size for the phase and frequency control to work correctly)

**Notes:**

This function can be used for continuous AWF generation. User needs to allocate multiple buffers and write next buffer on the timer. Status reports the current active buffer to avoid overwriting data on the fly
To make span control to acjieve full range `<f_data>` should be within +/-14.3V and `<data>` from 0 to 0xFFFF (0x8000 is zero volts).

### *4.21.14 DqAdv364WriteChannel*

**Syntax:**

```
int DqAdv364Function(int hd, int devn)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | channel (use DQ_AO364_OFS_CH for offset DAC - 20 bits) |
| `uint32 data` | data for the port |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-358 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Write to DAC's channel.

**Note:**

This is a single channel function.
It will stop any waveform on that channel and output a DC value.

### 4.21.15 DqAdv364WriteOffsWF

*Not Implemented.*

**Syntax:**

```
int DqAdv364WriteOffsWF(int hd, int devn, int channel, int mode,
int buffer, int from, int size, uint16* data, double* fdata,
uint32* status)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | channel to config |
| `int mode` | modifiers <reserved> |
| `int buffer` | buffer ID (created in DqAdv364SetAWF()) |
| `int from` | write from a certain sample in the buffer |
| `int size` | data size in samples |
| `uint16* data` | array of data (should be of CL size) or NULL if takeraw is TRUE |
| `double* fdata` | array of float point voltages or NULL if takeraw is FALSE |
| `uint32* status` | WF generation status |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an AO-358 |
| `DQ_BAD_PARAMETER` | `cmd` is not one of the defined constants below or `len` is less than or equal to zero or greater than 1024 for read or greater than 256 for write. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function writes waveform to AO-364 waveform buffer for offset DAC.

**Note:**

Offset DAC buffer allows 4096 points to be written.

## 4.22 DNx-TC-378 layer

See `DqAdv3xxWrite()` for writing outputs.
See the following for configuration/circuit breaker programming:

- `DqAdv318CBStatus()`
- `DqAdv318Reengage()`
- `DqAdv318SetCBLevels()`
- `DqAdv318SetConfig()`

### 4.22.1 DqAdv378ADCEnable

**Syntax:**

```
int DqAdv378ADCEnable(int hd, int devn, int enable, uint32* mask)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int enable` | TRUE to enable; FALSE to disable |
| `uint32* mask` | Pointer to a bit mask of channels to apply ADC enable or disable. Setting `*mask` to zero results in only retrieving the current enable state |

**Output:**

| | |
|---|---|
| `uint32* mask` | Returned bits 7:0 show which channels are enabled or disabled. Note that this value is returned *after* applying changes. |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a TC-378 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Enables or disables the Guardian ADC on a per channel basis and retrieves present ON/OFF state of each ADC.

**Notes:**

The circuit breaker functionality will also be disabled when the ADCs are disabled. The ADCs and circuit breaker function will be restarted when CONFIG MODE is entered (CONFIG is entered on power-up or set via the `DqCmdSetMode`() API).

### *4.22.2 DqAdv378ReadADC*

**Syntax:**

```
int DqAdv378ReadADC(int hd, int devn, int CLSize, uint32* cl,
uint32* bdata, double* fdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Channel list size |
| `uint32* cl` | Channel list |

**Output:**

| | |
|---|---|
| `uint32* bdata` | 32-bit raw binary data (NULL if not required) |
| `double* fdata` | Converted data (NULL if not required) |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a TC-378 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function requests conversion results for each channel specified in the diagnostic channel list and stores them alongside converted data.

The TC-378 has 8 thermocouple output channels with 3 diagnostic measurements (or subchannels) available for each output channel.

- Diagnostic measurements:
  The default rate that the diagnostic ADC reads diagnostic subchannels is 7.5 Hz, resulting in all 3 monitor subchannels for all 8 analog output channels updating about twice a second (every 400 ms). The sample rate can be reset with the `DqAdv318SetConfig()` API (Note that UEI recommends not exceeding sampling at 10 Hz; up to 50 Hz is supported, though performance degrades at higher sample rates). The ADC starts running automatically upon first call to `DqAdv3xxWrite()`.
  Also note, the `DqAdv318CBStatus`() is used to read the current circuit breaker status and/or re-enable circuit breakers.

- Valid data flag/circuit breaker state/diagnostic channel ID encoded in `bdata`:
  Bit 31: The most significant bit of the raw data (`bdata`) will be a 1 if that reading has been updated since the last read. If the data has not been updated, the previous reading will be returned.
  Bit 30 indicates state of circuit breaker (1 if currently tripped).

Bits 27..25 of the `bdata` indicate which of the 3 ADC channels produced the data. The macro `DQ_AO318_CH_ID(S)` can be used to extract the channel numbers, also known as conversion type. The channels are defined by the `DQ_TC378_ADC_CH_*` constants. Because custom settings for the circuit breaker function may sometimes change the order in which the ADC channels are reported, it is always a good idea to pull the channel numbers out of the `bdata` values using the `DQ_AO318_CH_ID` macro instead of relying on the default order of the channels.

If custom circuit breaker settings cause an ADC channel to be unused it will be reported as `DQ_AO318_ADC_CH_UNUSED` in bits 27..25 of `bdata`.
There is a maximum of 64 entries in the channel list. (including the optional timestamp request). The subchannels (conversion types) are #defined as follows:

```
#define DQ_TC378_ADC_CH_I            //output current (over 0.1 Ohm shunt)
#define DQ_TC378_ADC_CH_EXT          // voltage on output after the relays
#define DQ_TC378_ADC_CH_TEMP         // temperature channel, degrees C
#define DQ_TC378_ADC_CH_UNUSED       // channel is unused
```

(The above defines appear in powerdna.h.)

Use the `DQ_AO318_MAKE_CL(CH,SUBCH)` helper macro to combine the channel and subchannel to make the entries in the channel list.

This function works using an underlying `DqCmdIoctl()`. It uses a `DQCMD_IOCTL` command with a `DQIOCTL_CVTCHNL` function under the hood.

**Note:**

## 4.23 DNx-RTD-388 layer

### 4.23.1 DqAdv388Write

**Syntax:**

```
int DqAdv388Write(int hd, int devn, uint32 CLSize, uint32 *cl, double
*res_ohms, uint32 delay, uint32 flags, uint32 *binvals)
```

**Command:**

Configure resistance values

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Channel list size; size of `cl`[], `res_ohms`[] (and optionally `binvals`[]) |
| `uint32 *cl` | Channel list (channel numbers), allowable values range 0..7 |
| `double* res_ohms` | List of requested resistance values in corresponding order to channel numbers in `cl` |
| `uint32 delay` | 8-bit delay in 12.5 µS increments prior to setting next resistance value, 0 = no delay.<br>The same delay is applied to all members of `cl` |
| `uint32 flags` | <Reserved> set to 0 |
| `uint32* binvals` | Pointer to user array with size >= `CLSize` to receive encoded resistance data. If `binvals` = NULL then data are automatically written to RTD-388 hardware instead of being passed back as an array |

**Output:**

| | |
|---|---|
| `uint32* binvals` | If `binvals` is not NULL, returns array of size `CLSize` updated with per channel binary encoded resistance data |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device number indicated by `devn` does not exist or is not an RTD-388. |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_RTD388_MAX_WR_PER_PKT`<br>or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Configures RTD-388 resistance values. Users provide a channel list (`cl`) and an array of resistances, and the API converts ohms and channel numbers to equivalent encoded resistance data. For the RTD-

388-1, `res_ohms` values may range from 180 to 3900 Ω. For the RTD-388-100, `res_ohms` values may range from 23 to 390 Ω.

If the return data array `binvals` is NULL, this API immediately writes the encoded resistance data directly to UEI hardware and updates the output resistances on the indicated channels.

If the return data array `binvals` is not NULL, this API stores the array of encoded resistance data in `binvals`, which can later be transferred to the UEI cube or rack for programming channel output resistances using the `DqAdv388WriteBin` API.

Programming a `delay` value causes I/O board logic to wait the programmed delay before each output resistance in `res_ohms` is updated on the corresponding channel.

### *4.23.2 DqAdv388WriteBin*

**Syntax:**

```
int DqAdv388WriteBin(int hd, int devn, uint32 CLSize, uint32* cl_bin)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Size of `cl_bin[]`, 360 entries max. If all writes are to the same channel, the maximum list size = 128, due to channel FIFO size limitations |
| `uint32* cl_bin` | Encoded resistance values (the channel # is in bits [22..20]). The `DqAdv388Write` creates the array passed to this API. |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device number indicated by `devn` does not exist or is not an RTD-388. |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_RTD388_MAX_WR_PER_PKT`<br>or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Writes encoded resistance data to UEI hardware, which updates output resistance on each channel encoded in `cl_bin` values.

**Note:**

This API is used after `DqAdv388Write` and accepts the array of encoded resistance data (`binvals`) that is optionally returned from `DqAdv388Write`.

### 4.23.3 DqAdv388WriteDiag

**Syntax:**

```
int DqAdv388WriteDiag(int hd, int devn, uint32 ch_open, uint32
ch_short, uint32 cs_max)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer/Board inside the IOM |
| `uint32 ch_open` | List of channels to open circuit:<br>'1' in bit position 0 opens ch0, '1' in bit position 1 opens ch1, etc. |
| `uint32 ch_short` | List of channels to short circuit:<br>'1' in bit position 0 shorts ch0, '1' in bit position 1 shorts ch1, etc.<br>Short circuits have priority. If a channel is commanded to be both shorted and open, it will be shorted. |
| `uint32 cs_max` | Maximum amount of time that an output will remain short circuited, in 100 µS increments. Defaults to 0 (see note below for more information. |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device number indicated by `devn` does not exist or is not an RTD-388. |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_RTD388_MAX_WR_PER_PKT` or a channel number in `cl` is too high |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Controls RTD-388 simulated open- and short-circuit diagnostic features. This function also resets the breakers to the default value (closed).

**Note:**

At system power-on, the shorted and open controls are in the normal state, (i.e. the simulated RTDs are not shorted and are not open).

The `cs_max` value controls the amount of time that an output will be short circuited.

- `cs_max`=0 results in short circuit of indefinite duration under software control (channel is shorted by setting corresponding bits in `ch_short;` short circuit is opened by resetting bits in `ch_short`)
- `cs_max` > 0 causes onboard hardware to open the short circuit after the programmed duration (max delay)

### *4.23.4 DqAdv388ADCEnable*

**Syntax:**

```
int DqAdv388ADCEnable(int hd, int devn, int enable, uint32* mask)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int enable` | TRUE to enable; FALSE to disable |
| `uint32* mask` | Pointer to a bit mask of channels to apply ADC enable or disable. Setting `*mask` to zero results in only retrieving the current enable state |

**Output:**

| | |
|---|---|
| `uint32* mask` | Returned bits 7:0 show which channels are enabled or disabled. Note that this value is returned *after* applying changes. |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an RTD-388 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Enables or disables the Guardian ADC on a per channel basis and retrieves present ON/OFF state of each ADC.

**Notes:**

The circuit breaker functionality will also be disabled when the ADCs are disabled. The ADCs and circuit breaker function will be restarted when CONFIG MODE is entered (CONFIG is entered on power-up or set via the `DqCmdSetMode`() API).

### *4.23.5 DqAdv388ReadADC*

**Syntax:**

```
int DqAdv388ReadADC(int hd, int devn, int CLSize, uint32* cl, uint32*
bdata, double* fdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Channel list size |
| `uint32* cl` | Channel list |

**Output:**

| | |
|---|---|
| `uint32* bdata` | 24-bit unscaled binary data (NULL if not required) |
| `double* fdata` | Converted data (NULL if not required) |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an RTD-388 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function requests conversion results for each channel specified in the diagnostic channel list and stores them alongside converted data.

The RTD-388 has 8 resistive output channels. Diagnostic $I_{in}$ and on-die ADC temperature ADC channel readings are available for each output channel.

- Diagnostic measurements:
  The diagnostic ADC reads current and temperature data at 6.5 Hz, resulting in all measurements for all 8 analog output channels updating about 3x a second (every 308 ms).
- Valid data flag/CB tripped flag encoded in `bdata`:
  Bit 31: The most significant bit of the raw data (`bdata`) will be a 1 if that reading has been updated since the last read. If the data has not been updated, the previous reading will be returned.
  Bit 30 of the raw data (`bdata`) will be a 1 if the circuit breaker is in a tripped state.

When creating the `cl` channel list, use the following #defines to specify ADC diagnostic channels:

- Iin current for channels 0 through 7:
  `DQ_RTD388_I_IN_CH_0 to DQ_RTD388_I_IN_CH_7`
- On-die ADC temperature for channels 0 through 7:
  `DQ_RTD388_T_CH_0 to DQ_RTD388_T_CH_7`

(The above constants are defined in powerdna.h.)

This function works using an underlying `DqCmdIoctl()`. It uses a `DQCMD_IOCTL` command with a `DQIOCTL_CVTCHNL` function under the hood.

**Note:**

### *4.23.6 DqAdv388Reengage*

**Syntax:**

`int DqAdv388Reengage(int hd, int devn, uint32 reset)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 reset` | Bits 7:0, reset the breakers for the corresponding channels |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an RTD-388 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function resets the circuit breakers for the channels where a ‘1’ is found in the bit position corresponding to the channel number of the RTD-388. If `reset` is zero, the function will do nothing and return `DQ_SUCCESS.`

If the ADC readings are still beyond the circuit breaker trip limits (set by `DqAdv388SetConfig)`, the breaker will trip again:

- Default $I_{in}$ current limits are approximately ±4.5 mA
- Default ADC on-die temperature limit is 105 ºC

**Note:**

Identical to `DqAdv318Reengage()`API.

## 4.23.7 DqAdv388CBStatus

**Syntax:**

```
int DqAdv388CBStatus(int hd, int devn, int chan_mask, uint32 flags,
uint32 reset, uint32* status)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chan_mask` | Bitmask of channels to apply parameters to |
| `uint32 flags` | Additional behavior modifiers <reserved> |
| `uint32 reset` | Bitmask what CBs to reset |

**Output:**

| | |
|---|---|
| `uint32* status` | Status of tripped CBs. `status[0]` = main status register, `status[1]` and greater contain status of channels enabled by `chan_mask`, reported in ascending order |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an RTD-388 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function resets circuit breakers defined by the bitmask parameter `uint32 reset` and returns circuit breaker status, ADC status, and simulated short circuit status.

The `status[0]` result provides the ADC and Circuit Breaker status for all 8 channels.
Bit definitions for `status[0]` are as follows (defined in powerdna.h):

- Bit[31:24]: Current state of the simulated short circuit control on channels 7-0. Reading a 1 back indicates the channel is shorted (the simulated short circuit relay closed).

```
#define DQ_RTD388_STS_RLSC7        (1L<<31)
#define DQ_RTD388_STS_RLSC6        (1L<<30)
#define DQ_RTD388_STS_RLSC5        (1L<<29)
#define DQ_RTD388_STS_RLSC4        (1L<<28)
#define DQ_RTD388_STS_RLSC3        (1L<<27)
#define DQ_RTD388_STS_RLSC2        (1L<<26)
#define DQ_RTD388_STS_RLSC1        (1L<<25)
#define DQ_RTD388_STS_RLSC0        (1L<<24)
```

- Bit[23:16]: Current status of the circuit breaker on channels 7-0
```
#define DQ_RTD388_STS_CB7_CS       (1L<<23)
#define DQ_RTD388_STS_CB6_CS       (1L<<22)
#define DQ_RTD388_STS_CB5_CS       (1L<<21)
#define DQ_RTD388_STS_CB4_CS       (1L<<20)
#define DQ_RTD388_STS_CB3_CS       (1L<<19)
#define DQ_RTD388_STS_CB2_CS       (1L<<18)
#define DQ_RTD388_STS_CB1_CS       (1L<<17)
#define DQ_RTD388_STS_CB0_CS       (1L<<16)
```
- Bit[15:8]: Sticky status of ADC on channels 7-0: Bit is set to one each time that the ADC state machine for the corresponding channel finishes conversions for both the Iin and temperature and the result from the last conversion is stored into the corresponding register. The bit is cleared after read.
```
#define DQ_RTD388_STS_ADC7_RDY     (1L<<15)
#define DQ_RTD388_STS_ADC6_RDY     (1L<<14)
#define DQ_RTD388_STS_ADC5_RDY     (1L<<13)
#define DQ_RTD388_STS_ADC4_RDY     (1L<<12)
#define DQ_RTD388_STS_ADC3_RDY     (1L<<11)
#define DQ_RTD388_STS_ADC2_RDY     (1L<<10)
#define DQ_RTD388_STS_ADC1_RDY     (1L<<9)
#define DQ_RTD388_STS_ADC0_RDY     (1L<<8)
```
- Bit[7:0]: Sticky status of circuit breaker on channels 7-0: Bit is set to one if circuit breaker is set for the corresponding channel.
```
#define DQ_RTD388_STS_CB7          (1L<<7)
#define DQ_RTD388_STS_CB6          (1L<<6)
#define DQ_RTD388_STS_CB5          (1L<<5)
#define DQ_RTD388_STS_CB4          (1L<<4)
#define DQ_RTD388_STS_CB3          (1L<<3)
#define DQ_RTD388_STS_CB2          (1L<<2)
#define DQ_RTD388_STS_CB1          (1L<<1)
#define DQ_RTD388_STS_CB0          (1L<<0)
```

The `status[1]`through `status[8]` provide additional status data for each channel, (Ch0 corresponds to `status[1],` Ch1 corresponds to `status[2],` etc.):
```
#define DQ_RTD388_CSTS_FHF (21) // =1 when FIFO word count is below watermark
#define DQ_RTD388_CSTS_CFE (20) // =1 when FIFO is empty (reads will be
                            //    ignored)
#define DQ_RTD388_CSTS_FF  (19) // =1 when FIFO is full (writes will be
                            //    ignored)
#define DQ_RTD388_CSTS_RLD (18) // =1 if relays were disconnected by CB
#define DQ_RTD388_CSTS_OC1 (17) // =1 if corresponding "over limit" condition
                            //    is detected
#define DQ_RTD388_CSTS_OC0 (16)
// Bits 15-0 are "sticky" bits
#define DQ_RTD388_CSTS_RLD_S    (1L<<4) // =1 if relays were disconnected by CB
#define DQ_RTD388_CSTS_ADC1DR_S (1L<<3) // =1 if calibrated ADC data is ready for
                                    //    channel 1
#define DQ_RTD388_CSTS_ADC0DR_S (1L<<2) // =1 if calibrated ADC data is ready for
                                    //    channel 0
#define DQ_RTD388_CSTS_OC1_S    (1L<<1) // =1 if corresponding "over limit"
                                    //    condition was ever detected
#define DQ_RTD388_CSTS_OC0_S    (1L<<0)
```

**Note:** Note that bits not shown are reserved.

### *4.23.8 DqAdv388SetConfig*

**Syntax:**

```
int DqAdv388SetConfig(int hd, int devn, int chan_mask, int flags,
pDQRTD388CFG cfg)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chan_mask` | Bitmask of channels to apply parameters to |
| `int flags` | Additional behavior modifiers <reserved> |
| `pDQRTD388CFG cfg` | Structure of configuration parameters |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an RTD-388 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function configures circuit breakers and ADC functionality. The `chan_mask` parameter determines the channels to which the parameters are applied. A '1' in the corresponding bit position enables the parameters for that channel, (e.g. 0x03 applies parameters to channels 0 and 1 only, 0xFF applies parameters to all 8 channels). Parameters are set using the `DQRTD388CFG structure` as follows:

```
typedef struct {
    uint32 prmmask;     // bitmask: which parameters are valid and
                        //     need to be set
    uint32 rdcnt;       // Number of consecutive 'past limit' ADC
                        //     reads that need to occur before
                        //     breaker trips
    uint32 CB_limit_i;  // CB current limit value, code 0x800000..ffffff
    uint32 CB_limit_t;  // CB temperature limit value, code 0x800000..ffffff
    uint32 CB_enables;  // enable/disable circuit breakers
} DQRTD388CFG, *pDQRTD388CFG;
```

The `prmmask` member of the `DQRTD388CFG` structure is a bitfield that indicates which of the other members of the struct are valid and need to be set.

The `prmmask` bits are defined as follows:

```
#define DQ_RTD388_CFG_READ_COUNT    // rdcnt valid to modify
#define DQ_RTD388_CFG_SETCB_LIMIT   // CB_limit_i/t valid to modify
                                    //     CB comparison value
#define DQ_RTD388_CFG_SETCB_ENABLES // CB_enables is valid to modify
```

**Note:**

Please refer to the powerdna.h file for more detail of the `DQRTD388CFG` structure.

### *4.23.9 DqAdv388ReadLastBin*

**Syntax:**

```
int DqAdv388ReadLastBin(int hd, int devn, uint32 *data)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 * data` | Pointer to uint32 variable containing channel number to read from |

**Output:**

| | |
|---|---|
| `uint32 * data` | Returned last write: in the format of binary data read from channel specified in `data.` Binary data is encoded relay states used by hardware (not ohms) |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not RTD-388 |
| `DQ_BAD_PARAMETER_2` | `data` is NULL |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This diagnostic function reads back the last resistance encoding for a specified channel.

The last written value could also be the value from EEPROM that is written at system startup, if no `DqAdv388Write` occurs before `DqAdv388ReadLastBin`. This startup value is normally configured and saved using PowerDNA Explorer.

The read back data will be returned as encoded relay states that indicate which onboard relays are turned on and off to achieve the desired ohms that was programmed with the `DqAdv388Write`.

**Note:**

None.

## 4.24 DNx-DIO-401/402/404/405/406 layers

### 4.24.1 DqAdv40xWrite

**Syntax:**

`int DqAdv40xWrite(int hd, int devn, uint32 data)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 data` | Data for the port |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-401, DIO-402, DIO-404, DIO-405, DIO-43x, DIO-452, DIO-462, DIO-470, AI-218 or AI-228 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Writes a word of digital data bits to set corresponding channels on DIO hardware.

- DIO-402: 24-bits
- DIO-404, DIO-405, DIO-406, DIO-452, DIO-46x: 12-bits
- DIO-430: 30-bits
- DIO-432, DIO-433: 32-bits
- DIO-470: 10-bits
- AI-218 or AI-228: 8-bits (1 per each analog channel: 8 digital I/O)

**Note:**

None.

### 4.24.2 DqAdv40xRead

**Syntax:**

```
int DqAdv40xRead(int hd, int devn, uint32 *data)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 *data` | Pointer to buffer for read data |

**Output:**

| | |
|---|---|
| `uint32 *data` | Data read |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-401, DIO-404/6, DIO-405, DIO-432/3, DIO-462, AI-218 or AI-228 |
| `DQ_BAD_PARAMETER_2` | `data` is NULL |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Reads a word of digital data of corresponding channels on DIO hardware.

- DIO-401: 24-bits
- DIO-404, DIO-405, DIO-406: 12-bits
- AI-218 or AI-228: 8-bits (1 per each analog channel: 8 digital I/O)

For the DIO-432, DIO-433, and DIO-462 layers, returns circuit breaker status to `data[0]` and `data[1]`.

- `data[0]` is the current breaker status
- `data[1]` is the sticky status of each breaker.

For DIO-432, DIO-433, and DIO-462 layers use the `DqAdv462GetAll()`or `DqAdv432GetAll()`functions to retrieve the current DIO state.

**Note:**

None.

### *4.24.3 DqAdv40xReadLastWrite*

**Syntax:**

```
int DqAdv40xReadLastWrite(int hd, int devn, uint32 *Pdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 *Pdata` | Pointer to buffer for read data |

**Output:**

| | |
|---|---|
| `uint32 *Pdata` | Data read, 1 or 2 uint32's depending on layer type |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-402, DIO-403, DIO-404, DIO-405, DIO-406, DIO-430, DIO-432, DIO-433, DIO-452, DIO-462, DIO-463 or DIO-470 |
| `DQ_BAD_PARAMETER_2` | `Pdata` is NULL |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This diagnostic function reads back the last DIO output written to the layer.

The last written value could also be the value from EEPROM that is written at system startup, if no `DqAdv40xWrite` occurs before `DqAdv40xReadLastWrite`. This startup value is normally configured and saved using PowerDNA Explorer.

The read back data will be returned in the least significant bits of either two 32-bit values for a DIO-403 or in a single 32-bit value for all other layers.

The readback value is not affected by any circuit breaker action on DIO-4xx models that support CB's.

**Note:**

None.

### *4.24.4 DqAdv40xReadTs*

**Syntax:**

```
int DqAdv40xReadTs(int hd, int devn, int flags, uint32 *Pdata,
uint32 *tstamp)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int flags` | Additional flags, see below |

**Output:**

| | |
|---|---|
| `uint32 *Pdata` | Pointer to buffer for read data, two 32-bit words for DIO403 and DIO-448, one word for the other supported layers |
| `uint32 *tstamp` | Timestamp |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-401, DIO-402, DIO-403, DIO-404, DIO-405, DIO-406 or DIO-448 |
| `DQ_BAD_PARAMETER` | `Pdata` is NULL |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads the digital input state of the layers listed above in **Return.**

The timestamp value is returned to the buffer pointed to by `uint32 *tstamp.`

Users can optionally reset the timestamp before reading with the `int flags(`optional) parameter`;` use zero when not required:

```
#define DQIOCTL_DIO_RST_TIMESTAMP    //  resets timestamp before reading
```

**Note:**

None.

### 4.24.5 DqAdv40xSetHyst

**Syntax:**

```
int DqAdv40xSetHyst(int hd, int devn, uint16 level0, uint16
level1)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint16 level0` | Low threshold (integers 0…1024, where 1024 is Vcc) |
| `uint16 level1` | High threshold (integers 0…1024, where 1024 is Vcc) |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-401, DIO-402, DIO-404/406, or DIO-405 |
| `DQ_BAD_PARAMETER` | `data` is NULL |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets hysteresis levels for digital inputs in 10-bit steps. For example, setting the high threshold to 512 when VCC=24V will cause voltages of 12V and higher to be considered as a logical "1". By default, hysteresis levels are set to 25% of VCC (low) and 75% of VCC (high).

Hysteresis is a specific feature of DIO-40x layers. To access this feature, you enable it in the configuration word:

```
#define DQ_L401_HYSTEN      (1UL<<18)   // hysteresis programming is enabled
```

**Notes:**

### 4.24.6 *DqAdv40xConfigEvents*

**Syntax:**

```
int DAQLIB DqAdv40xConfigEvents(int hd, int devn, event401_t event,
uint32 pos_edge_mask, uint32 neg_edge_mask)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Async handle to the IOM received from `DqAddIOMPort()` |
| `int devn` | Layer inside the IOM |
| `event401_t event` | Type of event to monitor |
| `uint32 pos_edge_mask` | Mask of up to 24 bits to specify which input line will trigger an event on a positive edge<br>or, when programming periodic events, periodic rate divider |
| `uint32 neg_edge_mask` | Mask of up to 24 bits to specify which input line will trigger an event on a negative edge<br>or, when programming periodic events, optional divider reset |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-40x |
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up events and parameters for the DIO-401, DIO-404, DIO-405, and DIO-406 layers. It sets up firmware to send an asynchronous event notification packet upon reaching certain conditions.

**`event401_t event`:**

`event401_t event` is defined as:

```
typedef enum {
    EV401_CLEAR = 0x1000,          // clear all events
    EV401_DI_CHANGE = 0x101        // digital input change
    EV401_PERIODIC  = 0x102        // periodic event
} event401_t;
```

where

- `EV401_CLEAR:` clears all events that have previously been set along with runtime variables and accumulated events
- `EV401_DI_CHANGE`: when the configured data input line(s) switch states, an event packet will be sent (event packet description is provided below)
- `EV401_PERIODIC`: an event packet (as described below) will be sent continuously at the specified periodic rate. Periodic events are supported in PowerDNA version 4.10.0.24 and later

When specifying periodic events, typical usage is to first set up `EV401_DI_CHANGE` to configure packets being sent on specific DI state changes and then set up `EV401_PERIODIC` to periodically send a packet with the 1/0 state for each DI pin.

Initially all events should be cleared with the following call:

```
ret = DqAdv40xConfigEvents(a_handle, devn, EV401_CLEAR, 0, 0);
```

It clears all events that have previously been set along with runtime variables and accumulated events. When an event happens, the firmware sends back a packet of the following structure:

**`pos_edge_mask / neg_edge_mask:`**
`pos_edge_mask` and `neg_edge_mask` are defined as follows for `EV401_DI_CHANGE` or `EV401_PERIODIC`:

| Parameter | Usage with `EV401_DI_CHANGE` | Usage with `EV401_PERIODIC` |
|---|---|---|
| `pos_edge_mask` | Mask of up to 24 bits to specify which input line will trigger an event on a positive edge | Divider to set the periodic event rate<br>Divider is calculated by multiplying DIO-40x 66MHz base rate by period interval.<br>For example setting periodic events to occur at 1 s intervals:<br>`pos_edge_mask = (uint32)(DQ_DIO403_BASE*(PERIOD_INTERVAL))-1;`<br>where `DQ_DIO403_BASE = 66000000` and `PERIOD_INTERVAL=1.`<br>UEI recommends periods >100 ms. |
| `neg_edge_mask` | Mask of up to 24 bits to specify which input line will trigger an event on a negative edge | Configurable reset of periodic event<br>0: no reset will occur<br>1: enables a reset of the periodic event counter upon an asynchronous event |

**Event packet description:**
Once events are configured and enabled, the firmware sends back a packet when the specified asynchronous event occurs. For periodic events, the packet continues to be sent at the specified periodic rate.

Events for all board types contain the following structure, where `data[]` is cast to the specific event information for the board type being used (in this case the DIO-40x):

```
/* DQCMD_EVENT structure */
typedef struct {
    uint8   dev;                      // device
    uint8   ss;                       // subsystem
    uint16  size;                     // data size
    uint32  event;                    // event type = EV401_DI_CHANGE
    uint8   data[];                   // data
} DQEVENT, *pDQEVENT;
```

where:

`<size>` field contains sizeof(`EV401_ID`)+length (event header + event data).

`<event>` contains the type of the event, e.g. `EV401_DI_CHANGE`.

`<data[]>` if `DQEVENT` contains `EV401_DI_CHANGE` / `EV401_PERIODIC` flexible structure:

```
// Event data for 40x layer
typedef struct {
    uint32 chan;                      // channel information
    uint32 evtype;                    // type of the event
    uint32 tstamp;                    // timestamp of event
    uint32 size;                      // size of the following data in bytes
    uint32 data[];                    // data to follow
} EV401_ID, *pEV401_ID;
```

The timestamp (`tstamp`) returned is different depending whether the `evtype` is `EV401_DI_CHANGE` or `EV401_PERIODIC`:

- For `EV401_DI_CHANGE`, the `tstamp` represents the time when the change of state occurred on the DI pin.
- For `EV401_PERIODIC`, the `tstamp` represents the time when the packet was sent.

The event `data[]` of the returned `pEV401_ID` structure consists of uint32 status words (up to 24 LS bits valid) as follows:

| Parameter | For `EV401_DI_CHANGE` event | For `EV401_PERIODIC` event |
|---|---|---|
| `data[0]` | Rising edge events for input lines 23-0, 1=rising edge event has occurred on corresponding input line | Immediate state of digital inputs (input lines 23-0) |
| `data[1]` | Falling edge events for input lines 23-0, 1=falling edge event has occurred on corresponding input line. | Rising edge events for input lines 23-0, 1=rising edge event has occurred on corresponding input line |
| `data[2]` | `N/A` | Falling edge events for input lines 23-0, 1=falling edge event has occurred on corresponding input line. |

Note that you use the `DqRtAsyncEnableEvents()` API to enable events and the `DqCmdReceiveEvent()` API to receive events.

Event packets are handled by `DqCmdReceiveEvent()` in terms of converting data fields from big endian (network data representation) to little endian if used with Intel-based host computer while `EV401_ID` needs to be converted by the user.

The following code can be used:

```
// shuffle bytes big endian->little endian in place
      void ntoh_pEv40x(pEV401_ID pEv401) {
          uint32 i;

          pEv401->chan = ntohl(pEv401->chan);
          pEv401->evtype = ntohl(pEv401->evtype);
          pEv401->size = ntohl(pEv401->size);
          pEv401->tstamp = ntohl(pEv401->tstamp);
          for (i = 0; i < pEv401->size/sizeof(uint32); i++)
              pEv401->data[i] = ntohl(pEv401->data[i]);
          return;
      }
```

The reason for that is that `DqCmdReceiveEvent()` returns all different types of layer-specific events and as implemented doesn’t do this conversion.

**Note:**

If you need to verify the sequence of received events, you can use `DqCmdReceiveEventPkt()` instead of the `DqCmdReceiveEvent()`API to retrieve the packet ID counter (`dqCounter`).

## 4.25 DNA-DIO-403 layer

### 4.25.1 DqAdv403SetIo

**Syntax:**

```
int DqAdv403SetIo(int hd, int devn, uint32 Cfg)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 Cfg` | I/O Configuration |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-403 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function selects which ports are inputs and which are outputs.

**Note:**

Use constants `DIO403_ENPORT0...DIO403_ENPORT5` ORed together to select what ports should be output.

### *4.25.2 DqAdv403Write*

**Syntax:**

```
int DqAdv403Write(int hd, int devn, uint8 data[DQ_DIO403_PORTS])
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint8 data[]` | Byte array for all ports |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-403 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function writes `data` to DIO-403 ports. Firmware writes data into ports regardless the state of the port (input or output). The user can set the state of the output line by writing into ports and then enable outputs on all ports simultaneously.

**Note:**

Each index in the `data` array corresponds to a port on the DIO-403, and each bit represents a physical channel. For example, setting `data[1]=0xff` will set all 8 physical channels of port 1 to HIGH state.

Use `DIO403_ENPORTx in DqAdv403SetIo()` to select inputs and outputs.

### *4.25.3 DqAdv403Read*

**Syntax:**

```
int DqAdv403Read(int hd, int devn, uint8 data[DQ_DIO403_PORTS])
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint8 data[]` | Byte array for all ports |

**Output:**

| | |
|---|---|
| `uint8 data[]` | Output values for all ports |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-403 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function returns the input status of all DIO-403 ports regardless of state.

**Note:**

Each index in the `data` array corresponds to a port on the DIO-403, and each bit represents a physical channel. For example, reading `data[1]=0xff` will set all 8 physical channels of port 1 to HIGH state.

Use `DIO403_ENPORTx` in `DqAdv403SetIo()` to select inputs and outputs.

### *4.25.4 DqAdv403ConfigEvents*

**Syntax:**

```
int DAQLIB DqAdv403ConfigEvents(int hd, int devn, event403_t event,
uint8 pos_edge_mask[DQ_DIO403_PORTS],
uint8 neg_edge_mask[DQ_DIO403_PORTS])
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Async handle to the IOM received from `DqAddIOMPort()` |
| `int devn` | Layer inside the IOM |
| `event403_t event` | Type of event to monitor |
| `uint8 pos_edge_mask[6]` | Array of 8 bits mask to specify which input line will trigger an Event on a positive edge<br>(array size is `DQ_DIO403_PORTS` ==6) |
| `uint8 neg_edge_mask[6]` | Array of 8 bits mask to specify which input line will trigger an event on a negative edge<br>(array size is `DQ_DIO403_PORTS` ==6) |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-403 |
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up events and parameters which program firmware to send an asynchronous event notification packet upon reaching certain conditions.

`event403_t event` is defined as:

```
typedef enum {
    EV403_CLEAR = 0x1000,          // clear all events
    EV403_DI_CHANGE = 0x101        // digital input change
} event403_t;
```

Initially all events should be cleared with the following call:

```
ret = DqAdv403ConfigEvents(a_handle, devn, EV403_CLEAR, 0, 0);
```

It clears all events that has previously been set along with runtime variables and accumulated events.

When an event happens, the firmware sends back a packet of the following structure:

```
/* DQCMD_EVENT structure */
typedef struct {
    uint8   dev;                      // device
    uint8   ss;                       // subsystem
    uint16  size;                     // data size
    uint32  event;                    // event type = EV403_DI_CHANGE
    uint8   data[];                   // data
} DQEVENT, *pDQEVENT;
```

Where:

`<size>` field contains sizeof(`EV403_ID`)+length (event header + event data)
`<event>` contains the type of the event, e.g. in this case `EV403_DI_CHANGE`.
`<data[]>` if `DQEVENT` contains `EV403_DI_CHANGE` flexible structure:

```
// Event data for 403 layer
typedef struct {
    uint32 chan;                      // channel information
    uint32 evtype;                    // type of the event
    uint32 tstamp;                    // timestamp of event
    uint32 size;                      // size of the following data in bytes
    uint32 data[6];                   // data to follow
} EV403_ID, *pEV403_ID;
```

When DQEVENT contains `EV403_DI_CHANGE`, the event data[] consists of 6 uint32s (24 LS bits valid) as follows:

`data[0]` = rising edge events for input lines 23-0, 1=rising edge event has occurred on corresponding input line.
`data[1]` = falling edge events for input lines 23-0, 1=falling edge event has occurred on corresponding input line.
`data[2]` = rising edge events for input lines 47-24, 1=rising edge event has occurred on corresponding input line.
`data[3]` = falling edge events for input lines 47-24, 1=falling edge event has occurred on corresponding input line.
`data[4]` = event occurred, logical OR of `data[0]` and `data[1]`
`data[5]` = event occurred, logical OR of `data[2]` and `data[3]`

**Note:**

Note that the `DqCmdReceiveEvent()` API is used to wait for the event packets and that while DQEVENT is properly handled by `DqCmdReceiveEvent()` in terms of converting data fields from big endian (network data representation) to little endian if used with Intel-based host computer `EV403_ID` needs to be converted by the user. The following code can be used:

```
// shuffle bytes big endian->little endian in place
void ntoh_pEv403(pEV403_ID pEv403) {
    uint32 i;

    pEv403->chan = ntohl(pEv403->chan);
    pEv403->evtype = ntohl(pEv403->evtype);
    pEv403->size = ntohl(pEv403->size);
    pEv403->tstamp = ntohl(pEv403->tstamp);
    for (i = 0; i < pEv403->size/sizeof(uint32); i++)
        pEv403->data[i] = ntohl(pEv403->data[i]);
    return;
}
```

The reason for that is that `DqCmdReceiveEvent()` returns all different types of layer-specific events and as implemented doesn’t do this conversion.

Additionally, if you need to verify the sequence of received events, you can use `DqCmdReceiveEventPkt()` instead of the `DqCmdReceiveEvent()`API to retrieve the packet ID counter (`dqCounter`).

See the `DqAdv403ConfigEvents32()` API for configuring asynchronous events that additionally periodicially report the states of the input pin.

### *4.25.5 DqAdv403ConfigEvents32*

**Syntax:**

```
int DAQLIB DqAdv403ConfigEvents32(int hd, int devn, event403_t event,
uint32 port0_mask, uint32 port1_mask;
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Async handle to the IOM received from `DqAddIOMPort()` |
| `int devn` | Layer inside the IOM |
| `event403_t event` | Type of event to monitor |
| `uint32 port0_mask` | Enables for input lines 23-0 or periodic rate divider |
| `uint32 port1_mask` | Enables for input lines 47-24 or optional divider reset |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-403 |
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up events and parameters which program firmware to send an asynchronous event notification packet upon reaching certain conditions.

**`event403_t event:`**

`event403_t event` is defined as:

```
typedef enum {
    EV403_CLEAR = 0x1000,           // clear all events
    EV403_DI_CHANGE = 0x101         // digital input change
    EV403_PERIODIC  = 0x102         // periodic event
} event403_t;
```

where

- `EV403_CLEAR`: clears all events that have previously been set along with runtime variables and accumulated events
- `EV403_DI_CHANGE`: when the configured data input line(s) switch states, an event packet will be sent (event packet description is provided below)

- `EV403_PERIODIC`: an event packet (as described below) will be sent continuously at the specified periodic rate. You can optionally reset when the periodic packet is sent by setting `neg_edge_mask[0]`to 1. Periodic events are supported in PowerDNA version 4.10.0.24 and later

When specifying periodic events, typical usage is to first set up `EV403_DI_CHANGE` to configure packets being sent on specific DI state changes and then set up `EV403_PERIODIC` to periodically send a packet with the 1/0 state for each DI pin.

Initially all events should be cleared with the following call:

```
ret = DqAdv403ConfigEvents(a_handle, devn, EV403_CLEAR, 0, 0);
```

**`port0_mask / port1_mask:`**
`port0_mask` and `port1_mask` are defined as follows for `EV403_DI_CHANGE` or `EV403_PERIODIC`:

| Parameter | Usage with `EV401_DI_CHANGE` | Usage with `EV401_PERIODIC` |
|---|---|---|
| `port0_mask` | 24 LS bits<br>1 in the corresponding bit position enables change of state events for input lines 23-0. | Divider to set the periodic event rate<br>Divider is calculated by multiplying DIO-40x 66MHz base rate by period interval.<br>For example setting periodic events to occur at 1 s intervals:<br>`port0_mask = (uint32)(DQ_DIO403_BASE*(PERIOD_INTERVAL))-1;`<br>where `DQ_DIO403_BASE = 66000000` and `PERIOD_INTERVAL=1.`<br>UEI recommends periods >100 ms. |
| `port1_mask` | 24 LS bits<br>1 in the corresponding bit position enables change of state events for input lines 47-24 | Configurable reset of periodic event<br>0: no reset will occur<br>1: enables a reset of the periodic event counter upon an asynchronous event |

**Event packet description:**
Once events are configured and enabled, the firmware sends back a packet when the specified asynchronous event occurs. For periodic events, the packet continues to be sent at the specified periodic rate.

Events for all board types contain the following structure, where `data[]` is cast to the specific event information for the board type being used (in this case the DIO-40x):
It clears all events that has previously been set along with runtime variables and accumulated events.

When an event happens, the firmware sends back a packet of the following structure:

```
/* DQCMD_EVENT structure */
typedef struct {
    uint8   dev;                        // device
    uint8   ss;                         // subsystem
    uint16  size;                       // data size
    uint32  event;                      // event type = EV403_DI_CHANGE
    uint8   data[];                     // data
} DQEVENT, *pDQEVENT;
```

Where:

`<size>` field contains sizeof(`EV403_ID`)+length (event header + event data)

`<event>` contains the type of the event, e.g. `EV403_DI_CHANGE`.

`<data[]>` if `DQEVENT` contains `EV403_DI_CHANGE` flexible structure:

```
// Event data for 403 layer
typedef struct {
    uint32 chan;                        // channel information
    uint32 evtype;                      // type of the event
    uint32 tstamp;                      // timestamp of event
    uint32 size;                        // size of the following data in bytes
    uint32 data[6];                     // data to follow
} EV403_ID, *pEV403_ID;
```

The timestamp (`tstamp`) returned is different depending whether the `evtype` is `EV403_DI_CHANGE` or `EV403_PERIODIC`:

- For `EV403_DI_CHANGE`, the `tstamp` represents the time when the change of state occurred on the DI pin.
- For `EV403_PERIODIC`, the `tstamp` represents the time when the packet was sent.

When DQEVENT contains `EV403_DI_CHANGE` or `EV403_PERIODIC`, the event data[] consists of 6 uint32s (24 LS bits valid) as follows:

| Parameter | For `EV403_DI_CHANGE` event | For `EV403_PERIODIC` event |
|---|---|---|
| `data[0]` | Rising edge events for input lines 23-0, 1=rising edge event has occurred on corresponding input line | Debounced state of digital inputs (input lines 23-0) |
| `data[1]` | Falling edge events for input lines 23-0, 1=falling edge event has occurred on corresponding input line. | Immediate state of digital inputs (input lines 23-0) |
| `data[2]` | Rising edge events for input lines 47-24, 1=rising edge event has occurred on corresponding input line. | Debounced state of digital inputs (input lines 47-24) |
| `data[3]` | Falling edge events for input lines 47-24, 1=falling edge event has occurred on corresponding input line. | Immediate state of digital inputs (input lines 47-24) |
| `data[4]` | Event occurred, logical OR of `data[0]` and `data[1]` | Change of state data<br>(input lines 23-0)<br>if `EV403_DI_CHANGE` is also configured |

|  |  |  |
|---|---|---|
| `data[5]` | Event occurred, logical OR of `data[2]` and `data[3]` | Change of state data<br>(input lines 47-24)<br>if `EV403_DI_CHANGE` is also configured |

Note that you use the `DqRtAsyncEnableEvents()` API to enable events and the `DqCmdReceiveEvent()` API to receive events.

Note that the `DqCmdReceiveEvent()` API is used to wait for the event packets and that while DQEVENT is properly handled by `DqCmdReceiveEvent()` in terms of converting data fields from big endian (network data representation) to little endian if used with Intel-based host computer `EV403_ID` needs to be converted by the user. The following code can be used:

```
// shuffle bytes big endian->little endian in place
void ntoh_pEv403(pEV403_ID pEv403) {
    uint32 i;

    pEv403->chan = ntohl(pEv403->chan);
    pEv403->evtype = ntohl(pEv403->evtype);
    pEv403->size = ntohl(pEv403->size);
    pEv403->tstamp = ntohl(pEv403->tstamp);
    for (i = 0; i < pEv403->size/sizeof(uint32); i++)
        pEv403->data[i] = ntohl(pEv403->data[i]);
    return;
}
```

The reason for that is that `DqCmdReceiveEvent()` returns all different types of layer-specific events and as implemented doesn’t do this conversion.

**Note:**
If you need to verify the sequence of received events, you can use `DqCmdReceiveEventPkt()` instead of the `DqCmdReceiveEvent()`API to retrieve the packet ID counter (`dqCounter`).

## *4.26 DNA-DIO-404/406 layers*

### *4.26.1 DqAdv404SetHyst*

**Syntax:**

```
int DqAdv404SetHyst(int hd, int devn, int ref_volts, float *level0,
float *level1)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ref_volts` | One of the DQ_DIO404_REF_* constants indicating the reference voltage that will be applied |
| `float *level0` | Pointer to the desired level 0 voltage |
| `float *level1` | Pointer to the desired level 1 voltage |

**Output:**

| | |
|---|---|
| `float *level0` | Returns the actual level 0 voltage set |
| `float *level1` | Returns the actual level 1 voltage set |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-404 |
| `DQ_BAD_PARAMETER` | ref_volts is not one of the available constants, or level0 or level1 is NULL |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets the hysteresis levels for a DIO-404 layer.

Hysteresis is a specific feature of DIO-40x layers. To access this feature, you should enable it in the configuration word:

```
#define DQ_L401_HYSTEN      (1UL<<18)   // hysteresis programming is enabled
```

**Note:**

The DACs in the DIO-404 layer are not linear. For each of the supported reference voltages, there is a table mapping DAC 0 and DAC 1 value pairs to level 0 and level 1 voltage values. This function will select the DAC settings that result in level values closest to those specified in the level0 and level1 parameters. The level1 value has priority, which means that the table entry with the closest level1 value is found. If there is more than one table entry whose level1 value is equal to (or equally close to) the specified level1 value, the entry whose level0 value is closest to the one specified is chosen. The DAC values in this table entry are used to set the DACs in the layer, and the actual level0 and level1 values in the table are returned.

The ref_volts parameter must be one of the following constants, which represent the currently supported reference voltages:

```
#define DQ_DIO404_REF_3_3V  3   // 3.3 Volts
#define DQ_DIO404_REF_5V    5   // 5 Volts
#define DQ_DIO404_REF_12V   12  // 12 Volts
#define DQ_DIO404_REF_24V   24  // 24 Volts
#define DQ_DIO404_REF_36V   36  // 36 Volts
```

If a different reference voltage is desired, use DqAdv40xSetHyst to set the raw DAC values manually, but be aware that due to the non-linearity of the DACs, there is no guarantee of what the resulting transition voltages will be.

## 4.27 DNx-MUX-414 / DNR-MUX-418 layers

API in this section apply to both the DNx-MUX-414 and DNR-MUX-418 I/O boards. Where applicable, differences between the two boards are noted.

### 4.27.1 DqAdv414Write

**Syntax:**

```
int DqAdv414Write(int hd, int devn, DQ414W pdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `DQ414W pdata` | Data structure for closing or opening the A, B, and/or C relays and programming sync in, sync out, and/or break-before-make configuration |

**Output:**

N/A

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device number indicated by `devn` does not exist or is not a MUX-414 or MUX-418. |
| `DQ_BAD_PARAMETER` | `pdata.rflags` is not a supported value |
| `DQ_PKT_TOOLONG` | packet data buffer is greater than maximum payload size |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Configures channel multiplexer switches opened or closed, as well as sync in, sync out and break-before-make configuration. Each channel is composed of a COM and 3 switches: A, B, C.

Users program switch connections using the `DQ414W pdata` data structure:

```
typedef struct {
    int32 rflags;
    int32 mux_select[2];
    int32 relay_select[3];
} DQ414W, *pDQ414W;
```

You have two options of how to program the A, B, or C relays.

The typical application will close a single relay per channel, to connect COM to A, B, or C (or open all 3 for open circuit tests) using `mux_select[]`. Alternatively, you can program more than one relay closed per channel using `relay_select[]`.

> **NOTE:** You must use caution if you program using `relay_select. relay_select` allows you to close multiple mux relays per channel. This means if you have sources on A, B, and C, you will short your source equipment. This mode is intended for diagnostic purposes: for example, a sensor connected to COM and multiple listening devices on A, B, and/or C.

You set `rflags` to tell the firmware whether you are programming with `mux_select[]` or `relay_select[].`

`rflags` parameter:

Controls whether `mux_select[]` or `relay_select[]` is used to program switches (choose only one of the `_PORTx` or `_RELAY_x` defines; ORing more than 1 returns `DQ_BAD_PARAMETER`):

- `DQ_MUX414_W_PORT0    // write mux_select[0] (414 or 418)`
- `DQ_MUX414_W_PORT1    // write mux_select[1] (418 only)`
- `DQ_MUX414_W_PORT10   // write mux_select[1] and mux_select[0]`

- `DQ_MUX414_W_RELAY_A    // write relay_select[0]`
- `DQ_MUX414_W_RELAY_B    // write relay_select[1]`
- `DQ_MUX414_W_RELAY_BA   // write relay_select[1]`
  `//    and relay_select[0]`
- `DQ_MUX414_W_RELAY_C    // write relay_select[2]`
- `DQ_MUX414_W_RELAY_CA   // write relay_select[2]`
  `//    and relay_select[0]`
- `DQ_MUX414_W_RELAY_CB   // write relay_select[2]`
  `//    and relay_select[1]`
- `DQ_MUX414_W_RELAY_CBA  // write relay_select[2]`
  `//    and relay_select[1]`
  `//    and relay_select[0]`

Additionally `rflags` can be ORed with the following flags to program sync in, sync out, and/or break-before-make configuration flags (see Note below for more description):

- `DQ_MUX414_W_OPTION_SIN`: sync in handshaking
- `DQ_MUX414_W_OPTION_SOUT`: assert sync out
- `DQ_MUX414_W_OPTION_NO_BBM`: disable break-before-make

`mux_select[]` parameter:

Program each channel independently to close one of the three relays (A, B or C) or open all 3.

`mux_select[0]` programs channels 13 through 0 of MUX-414 or MUX-418

`mux_select[1]` programs channels 17 through 14 of MUX-418.

Which relays are opened and which are closed for a channel maps to 1 of 4 states:

- 1 to close A relay
- 2 to close B relay
- 3 to close C relay
- 0 to open all 3 relays

The mux state for each channel maps to corresponding 2-bits in `mux_select`.

For `mux_select[0]`:

- bit 31:28 = N/A (program 0)
- bit 27:26 = channel 13 mux state
- bit 25:24 = channel 12 mux state
- …
- bit 3:2 = channel 1 mux state
- bit 1:0 = channel 0 mux state

For `mux_select[1]`:

- bit 31:8 = N/A (program 0)
- bit 7:6 = channel 17 mux state
- bit 5:4 = channel 16 mux state
- bit 3:2 = channel 15 mux state
- bit 1:0 = channel 14 mux state

**Example:** For MUX-414, to program channel 0 to close the A-relay (1), channel 2 to close the C-relay (3), channel 4 to close the B relay (2), and all other channels to open all the relays (0), you set the `rflags` for using `mux_select[0]` and program the following:

```
pdata.rflags= DQ_MUX414_W_PORT0;
pdata.mux_select[0] = 0x0000231;
```

`relay_select[]` parameter:

Using `relay_select,` users have the option of closing more than 1 relay per channel.

`relay_select[0]` closes A relays.

`relay_select[1]` closes B relays.

`relay_select[2]` closes C relays.

Each channel is programmed by setting or resetting its corresponding bit in the `relay_select[]` array, (e.g. the A relay for channel 0 corresponds to bit 0 in `relay_select[0]`, the B relay for channel 5 corresponds to bit 5 in `relay_select[1]`, the C relay for channel 5 corresponds to bit 5 in `relay_select[2]`). A ‘1’ corresponds to closing the relay, ‘0’ to opening.

**Example:** to program channel 0 through 13 to close the A relay, and channel 2 to close the C-relay, and all other channels to open all other relays, you set the `rflags` for using `relay_select[0]` through `relay_select[2]` and program the following:

```
pdata.rflags= DQ_MUX414_W_RELAY_CBA;
pdata.relay_select[0] = 0x00003FFF;
pdata.relay_select[1] = 0x00000000;
pdata.relay_select[2] = 0x00000004;
```

**Note:**

In addition to configuring which `mux_` or `relay_` arrays to use for programming switch states, the `rflags` variable also controls using sync in and/or sync out for handshaking or enabling/disabling break-before-make.

- `DQ_MUX414_W_OPTION_SIN`: OR in this flag if you wish to use the sync in pin as a gate for switching relays. This option allows users to control relay switching with an external signal.

- `DQ_MUX414_W_OPTION_SOUT`: OR in this flag to assert the sync out pin. The timing for this pin is controlled via the `DqAdv414SetCfg` API.

- `DQ_MUX414_W_OPTION_NO_BBM`: OR in this flag to disable break-before-make circuitry. When you program a state change for a channel, break-before-make circuitry first opens all 3 relays before closing the relay that you program to close using `mux_select[]` or `relay_select[]`. Timing for the break-before-make circuit is programmed using the `DqAdv414SetCfg` API.

### *4.27.2 DqAdv414ReadADC*

**Syntax:**

`int DqAdv414ReadADC(int hd, int devn, DQ414ADC adata)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `DQ414ADC adata` | Data structure for returning diagnostic data from onboard ADC: internal voltage from 3.3 V or 2.5 V supply, internal temperature in °C, and status |

**Output:**

`N/A`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device number indicated by `devn` does not exist or is not a MUX-414 or MUX-418. |
| `DQ_NO_MEMORY` | not enough memory |
| `DQ_PKT_TOOLONG` | packet data buffer is greater than maximum payload size |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Reads diagnostic data from onboard ADC.

Data is returned as `DQ414ADC` data structure:

```
typedef struct {
    double adc_in;    // Reserved
    double adc_3_3;   // layer selftest, monitor internal 3.3V supply
    double adc_2_5;   // layer selftest, monitor internal 2.5V supply
    double adc_deg_c; // adc temperature in degrees C
    uint32 status;    // status identical to .status
                     //    returned by DqAdv414ReadStatus()
    uint32 timestamp; // timestamp
} DQ414ADC, *pDQ414ADC;
```

**Note:**

Before reading data, call `DqAdv414ReadADC(hd, devn, NULL)` once to initialize the ADC.

### *4.27.3 DqAdv414ReadStatus*

**Syntax:**

```
int DqAdv414ReadStatus(int hd, int devn, pDQ414STATUS rstatus)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `DQ414STATUS rstatus` | Data structure for returning relay positions and status |

**Output:**

`N/A`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device number indicated by `devn` does not exist or is not a MUX-414 or MUX-418. |
| `DQ_NO_MEMORY` | not enough memory |
| `DQ_PKT_TOOLONG` | packet data buffer is greater than maximum payload size |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Reads relay position and status. Data is returned as `DQ414STATUS` data structure:

```
typedef struct {
    uint32 relay_a;
    uint32 relay_b;
    uint32 relay_c;
    uint32 status;
} DQ414STATUS, *pDQ414STATUS;
```

Relay position data is returned as ‘1’ for closed, ‘0’ open.
Status is returned as:

- `Bit17, DQ_MUX414_STS_ADCDR`: ‘1’ means data is ready from ADC
- `Bit 16, DQ_MUX414_STS_OVR`: ‘1’ means overrun (write while busy)
- `Bit 3, DQ_MUX414_STS_BUSY`: ‘1’ means state machine is busy
- `Bit 2, DQ_MUX414_STS_SYNCWAIT`: ‘1’ means output state machine is waiting for the external SYNC/ready
- `Bit 1, DQ_MUX414_STS_RDY`: ‘1’ means relays are settled
- `Bit 0, DQ_MUX414_STS_DI_STS`: reports the logic state of the sync in pin

**Note:**

The status word can also be accessed with `DqAdv40xRead()` if readback of relay positions are not required.

### *4.27.4 DqAdv414SetCfg*

**Syntax:**

`int DqAdv414SetCfg(int hd, int devn, pDQ414CFG cfg)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `DQ414CFG cfg` | Data structure for setting configuration |

**Output:**

`N/A`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device number indicated by `devn` does not exist or is not a MUX-414 or MUX-418. |
| `DQ_BAD_PARAMETER` | not a supported value state |
| `DQ_PKT_TOOLONG` | packet data buffer is greater than maximum payload size |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Configures delays and sync in / sync out modes.

```
typedef struct {
    uint32 on_delay;       // time before next command is accepted
    uint32 off_delay;      // breaking time of break-before-make
```

```
        uint32 di_mode;        // sync in pin operation mode
        uint32 di_polarity;    // Polarity of sync_in strobe
        uint32 sync_out_pw;    // sync_out pin pulse length
        uint32 sync_out_mode;  // sync_out pin mode of operation
        uint32 sync_skip;      // release from sync in waiting
    } DQ414CFG, *pDQ414CFG;
```

`on_delay:` program the time before next command is accepted in 10uS units, range 1..256

`off_delay:` program the breaking time of break-before-make in 10uS units, range 1..256

`di_mode:` program the sync in pin operation mode,  0 = (level) / 1 =edge

`di_polarity:` program the polarity of sync in strobe: 0- (low)falling / 1- (high)rising

`sync_out_pw:` program the sync out pin pulse length for sync_out_mode 6 and 7
0 - 1uS; 1 - 10uS; 2 - 100uS; 3 - 1mS; 4..15 - reserved

`sync_out_mode:` program mode of operation for the sync out pin

- 0 - drive sync out pin with constant logic '0'
- 1 - drive sync out pin with constant logic '1'
- 2..5 - driven by internal SYNC BUS[0]..[3]
- 6 - positive transitioning pulse on relay write (sync out pin normally low)
- 7 - negative transitioning pulse on relay write (sync out pin normally high)
- 8 - logic '1' level on "relays ready" (UEI test adapter expects this)
- 9 - logic '0' level on "relays ready", will pulse high during ‘off_delay’

`sync_skip:` 0= normal operation, 1= stop waiting for a sync_in strobe, self clearing.

## *4.28 DNA-DIO-416 layer*

### *4.28.1 DqAdv416GetAll*

**Syntax:**

`int DqAdv416GetAll(int hd, int devn, pDQDIO416DATAIN data, double *fData)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `pDQDIO416DATAIN data` | Pointer to store all readable parameters from DIO-416 |
| `double *fData` | Pointer to array of 16 doubles to store the current readings (in amps). Null if not required. |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-416 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_BAD_PARAMETER` | `data` is NULL |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads all available information from the DIO-416 layer, including readback from the ADCs that are monitoring output current on every channel. ADC values are provided in raw binary format and also in microvolts as a part of `pDQDIO416DATAIN` structure. The ADC values are also available in the double data type using the `fData` parameter.

DIO-416 is a unique combination of the high-current sourcing/sinking Digital Output and precision Analog Input. The current monitor may be accessed at any time. If current is above the pre-defined current limit, the layer engages a circuit breaker that disconnects load by disabling (turning OFF) the output FET.

**Note:**

The DIO-416 layer provides multiple status parameters that may be accessed via a `DqAdv416GetAll` function call. Information is returned in the `pDQDIO416DATAIN` type:

```
#typedef struct {
   int32 cfg;        // Report list of the disabled channels.
   int32 port0out;   // Current value written to the output port
   int32 adcsts;     // Current status of the ADC configuration/state machines
   int32 port0ocs;   // Over-current interrupt status
```

```
    int32 port0ucs;    // Under-current interrupt status
    int32 rdcnt;       // Current value of the consecutive read counter setting
    int32 adcdata0;    // Result of the user-defined conversion on Low-side ADC
    int32 adcdata1;    // Result of the user-defined conversion on High-side ADC
    int32 disdiv;      // Current value of the re-enable divider
    int32 dout;        // Actual value driven by the DOUT port
                       // (Note: channels 0/2/4/6/8/10/12/14 inverted)
    int32 vccis;       // VCC available on the isolated side
    int32 ocl[DQ_DIO416_CHAN];  // Current value of the over-current limits
    int32 ucl[DQ_DIO416_CHAN];  // Current value of the under-current limits
    int32 adc[DQ_DIO416_CHAN];  // Current value of the conversion results (hex)
    int32 cur[DQ_DIO416_CHAN];  // Current value of the conversion results (uA)
} DQDIO416DATAIN, *pDQDIO416DATAIN;
```

- *cfg* reports a list of the channels currently disabled by the circuit breaker. Bits 31:16 are sticky, they report whether the corresponding channel (15:0) was ever disabled since the last call to this function and bits 15:0 report currently disabled channels.
- *port0out* bits 15:0 report the last values that were assigned to the output ports.
- *adcsts* report the internal status of the ADC state machines and configurations, which are used to set the ADC conversion speed:

  ```
  adcsts &=  (FFF00000 | SPEED_CONSTANT)
  ```
- *port0ocs* bits 15:0 contain a one for every channel for which an overcurrent condition was ever detected
- *port0ucs* bits 15:0 contain a one for every channel for which an undercurrent condition was ever detected
- *rdcnt* number of overcurrent samples read from an ADC prior to disconnection (0-32). A higher number leads to longer delay, but better noise/spike immunity.
- *adcdata0, adcdata1* results of user-defined ADC conversion. User may, as an option, define which channel at what speed he wants to acquire in addition to specifying that all channels will be acquired at a common speed. This feature allows precise reading of the channel of interest, virtually without slowing down circuit breaker performance. By default, data is returned in 24-bit straight binary format where 0xFFFFFF corresponds to 2A and 0x0 to -2A of current.
- *disdiv* divider for the 66MHz which defines the re-enable interval for the auto-retry channels set via cfg parameter. The divider should be selected in such a way that the re-enable interval is at least 2x longer than the expected circuit breaker reaction time (see below)
- *dout* current actual value of the digital output port. Note that low-side (even) channels will be inverted, and that the output may differ from what was set in the *port0out* due to the circuit breaker activity.
- *vccis* used to define presence of the user-supplied power on the isolated side of the DIO-416. A 1 indicates presence of the user-supplied power.
- *ocl[]* binary array of the over-current limits set for each channel individually, uses same binary format as *adcdata0.*
- *ucl[]* binary array of the under-current limits set for each channel individually, uses same binary format as *adcdata0.* Under-current limits may be used to define some potentially failing load and status of this verification is reported in *port0ucs*
- *adc[]* result of the last ADC conversion on all channels, uses same binary format as *adcdata0*

- *cur[]* result of the last ADC conversion on all channels, calibrated and converted into the microvolts. Values above/below +/-2000000uV indicate ADC or channel failure

ADC Speed selection constants (note that faster ADC speed also reduces accuracy):

```
//                            Code      ADC speed Rate/channel p-p noise    Break time mS
//                                                             (mA @ +/-2A) @ 4 decision samples
#define DQ_L416_ADCSPD_190    (1)    // 3.52KHz   190Hz/ch       5.5          20
#define DQ_L416_ADCSPD_130    (2)    // 1.76KHz   130Hz          5            25
#define DQ_L416_ADCSPD_85     (3)    // 880Hz      85Hz          3.6          40
#define DQ_L416_ADCSPD_45     (4)    // 440Hz      45Hz          2.9          70
#define DQ_L416_ADCSPD_22     (5)    // 220Hz      22Hz          1.25        130
#define DQ_L416_ADCSPD_12     (6)    // 110Hz      12Hz          1.1         250
#define DQ_L416_ADCSPD_6_5    (7)    // 55Hz        6.5Hz        0.84        500
#define DQ_L416_ADCSPD_3_2    (8)    // 27.5Hz      3.2Hz        0.77       1000
#define DQ_L416_ADCSPD_1_6    (9)    // 13.75Hz     1.6Hz        0.36       2000
#define DQ_L416_ADCSPD_0_8   (15)    // 6.875Hz     0.8Hz        0.26       4000
```

### *4.28.2 DqAdv416SetAll*

**Syntax:**

`int DqAdv416SetAll(int hd, int devn, pDQDIO416DATAOUT pdata)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `pDQDIO416DATAOUT data` | Pointer to the structure with initialization data for the DIO-416 |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-416 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function allows setting any of the configurable parameters of the DIO-416 layer.

The DIO-416 is a unique combination of high-current sourcing/sinking Digital Output and precision Analog Input and a digitally programmable circuit breaker. The default configuration for the circuit breaker may be changed by using the `DqAdv416SetAll` function.

**Note:**

The configuration should be set using `DQDIO416DATAOUT` type. For parameters that should be actually programmed on the layer, *Xset* field should be set to 1; it should be set to 0 for parameters that should not be affected. In some cases, `DqAdv416GetAll` should be called to retrieve values of the current parameters of the layer, and then the configuration should be updated and written to the layer.

```
* structure for SETPARAM data */
```

```
typedef struct {
    int32 cfgset;          // =1 if "cfg" should be updated
    int32 cfg;             // Set disconnection mode
    int32 discfgset;       // =1 if "discfg" should be updated
    int32 discfg;          // Override circuit breaker
    int32 adcspdset;       // =1 if "adcspd" should be updated
    int32 adcspd;          // Set ADC Timing
    int32 port0ocmset;     // =1 if "port0ocm" should be updated
    int32 port0ocm;        // Set over-current interrupt mask
    int32 port0ucmset;     // =1 if "port0ucm" should be updated
    int32 port0ucm;        // Set under-current interrupt mask
    int32 rdcntset;        // =1 if "rdcnt" should be updated
    int32 rdcnt;           // Set number of "failed" samples prior breaker engagement
    int32 adccfg0set;      // =1 if "adccfg0" should be updated
    int32 adccfg0;         // "User" ADC conversion control word for low-side ADC
    int32 adccfg1set;      // =1 if "adccfg1" should be updated
    int32 adccfg1;         // "User" ADC conversion control word for high-side ADC
    int32 disdivset;       // =1 if "disdiv" should be updated
    int32 disdiv;          // 66MHz divider for the re-enable counter
} DQDIO416DATAOUT, *pDQDIO416DATAOUT;
```

- *cfg* disconnection mode configuration, bitmask, bits 15:0 represent channels 15:0. A one in a corresponding bit enables auto-retry mode of the circuit breaker. The auto-retry period is defined by the *disdiv* parameter.
- *discfg* allows override of the circuit breaker. Note that 1) Overriding circuit breaker may permanently damage the layer if currents above the rated 1A will be flowing through the circuitry for a prolonged period of time and 2) Updating this parameter will not re-enable already disabled channels if those channels are not in auto-retry mode. Only a write to the digital output port will re-enable disabled channels.
- *adcspd* used to set ADC conversion speed: *adcspd = adcsts & (FFF00000 | SPEED_CONSTANT)*. Note that faster speed improves circuit breaker disconnection time but affects ADC accuracy.
- *port0ocm* bits 15:0 contain a one for every channel that should be included into the over-current status (*port0ocs)*. This parameter does not affect the circuit breaker.
- *port0ucm* bits 15:0 contain a one for every channel that should be included into the under-current status (*port0ocs)*.
- *rdcnt* number of overcurrent samples read from ADC prior to disconnection (0-32). A higher number leads to longer delay, but better noise/spike immunity.
- *adccfg0, adccfg1* Normally, two available ADC converters perform a cycle that includes eight conversions each, one for every channel connected to the ADC. ADCs are not synchronized and run completely independent from each other. As an additional feature, it is possible to write an LTC2448 command into the *adccfg0*, *adccfg1* parameter and get the corresponding conversion result in *adcdata0/adcdata1*. Please refer to LTC2448 documentation for details. This feature may be used for debugging or diagnostic purposes, especially in situations where a DNA cube is at remote location and an extended diagnostic is needed.
- *disdiv* divider for the 66MHz which defines the re-enable interval for the auto-retry channels set via the cfg parameter. A divider should be selected in a way that the re-enable interval is at least 2x longer than the expected circuit breaker reaction time (see below).

ADC Speed selection constants (note that faster ADC speed also reduces accuracy):

```
//                          Code      ADC speed Rate/channel p-p noise    Break time mS
//                                                             (mA @ +/-2A) @ 4 decision samples
#define DQ_L416_ADCSPD_190    (1)    // 3.52KHz   190Hz/ch       5.5          20
#define DQ_L416_ADCSPD_130    (2)    // 1.76KHz   130Hz          5            25
#define DQ_L416_ADCSPD_85     (3)    // 880Hz      85Hz          3.6          40
#define DQ_L416_ADCSPD_45     (4)    // 440Hz      45Hz          2.9          70
#define DQ_L416_ADCSPD_22     (5)    // 220Hz      22Hz          1.25        130
#define DQ_L416_ADCSPD_12     (6)    // 110Hz      12Hz          1.1         250
#define DQ_L416_ADCSPD_6_5    (7)    // 55Hz        6.5Hz        0.84        500
#define DQ_L416_ADCSPD_3_2    (8)    // 27.5Hz      3.2Hz        0.77       1000
#define DQ_L416_ADCSPD_1_6    (9)    // 13.75Hz     1.6Hz        0.36       2000
#define DQ_L416_ADCSPD_0_8   (15)    // 6.875Hz     0.8Hz        0.26       4000
```

### *4.28.3 DqAdv416SetLimit*

**Syntax:**

`int DqAdv416SetLimit(int hd, int devn, int limitid, double limitvalue)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int limitid` | Select Over- Under- range limit to set<br>0..15 - over-range limits for the channels 0-15<br>16..31 - under-range limits for the channels 16-31 |
| `Double limitvalue` | Desired current limit, Amps (-2..2) |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-416 |
| `DQ_BAD_PARAMETER` | limitid is not in the range of 31:0 or current limit is above/below `DQ_L416_MAXCURRENT/DQ_L416_MINURRENT` |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function should be used to program the circuit breaker over- and/or under- current limit on the DIO-416 layer.

**Note:**

The current limits should be set individually for the over- and under- current conditions on every channel. Thus, up to 32 calls of the DqAdv416SetLimit may be required to program the layer.

## 4.29 DNA-DIO-432/433 layers

To write data to the DIO-432/433 layers, use the `DqAdv40xWrite()` function.
To read the circuit breaker disconnect(trip) status, you may use either the `DqAdv40xRead()` function or the `DqAdv432GetAll()` function.

### 4.29.1 DqAdv432GetAll

**Syntax:**

```
int DqAdv432GetAll(int hd, int devn, pDQDIO432DATAIN data,
pDQDIO432CVTD fdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `pDQDIO432DATAIN data` | Pointer to store input data from DIO-432/433 |
| `pDQDIO432CVTD fdata` | Pointer to store converted values of current and voltage on every DIO-432/433 channel |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-432 or DIO-433 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function call returns two structures (structures should be allocated in the user application memory)

First structure `DQDIO432DATAIN` is defined as:

```
typedef struct {
    uint32 portout;              // Current value written to the output port (31-0 ch)
    uint32 portocs;              // Over-current status (31-0 ch)
    uint32 portucs;              // Under-current status (31-0 ch)
    uint32 rdcnt;                // Current value of the consecutive read counter setting
                                 // Number of "failed" reads prior to disabling the channel
    uint32 disdiv;               // Current value of the re-enable divider (32 bit)
    uint32 dout;                 // Actual value driven by the DOUT port (31-0 ch)
    uint32 vccis;                // VCC available (0/1) on the isolated side
    uint32 adc_i[DQ_DIO432_CHAN];  // Current value of the measured current (raw)
    uint32 adc_v[DQ_DIO432_CHAN];  // Current value of the measured voltage (raw)
```

} DQDIO432DATAIN, *pDQDIO432DATAIN;

This structure returns information about the current setting and status of the DIO-432/433 layer.

- <`portocs`> bits represent channels with the measured current above specified limit.
- <`portucs`> bits represent channels with the measured current below the specified limit or negative current flowing in opposite direction. Channel 0 is represented by bit 0, etc.
- <`rdcnt`> returns the programmed value of how many consecutive over or under current measurements the layer should make before the electronic circuit breaker disables this channel.
- <`disdiv`> returns the programmed value of the re-enable divider. Re-enable divider is derived from 66MHz timebase and specifies number of 15.15ns clock cycles before trying to re-enable channel.
- <`dout`> returns actual value driven by DOut port, taking disabled channels into consideration.
- <`vccis`> reports whether external power was applied to the DIO-432/433 card.
- <`adc_i`> is an array of raw 24-bit values from current-measuring A/D converters.
- <`adc_v`> is an array of raw 24-bit values from voltage-measuring A/D converters.

Next structure returns actual calibrated currents and voltages on each channel.

```
typedef struct {
    double current[DQ_DIO432_CHAN];  // Current value of the measured current (Amperes)
    double voltage[DQ_DIO432_CHAN];  // Current value of the measured voltage (Volts)
} DQDIO432CVTD, *pDQDIO432CVTD;
```

**Note:**

1. Use DqAdv40xWrite() to write output values for DIO-432/433 layers. Use channel 0 of DQ_SS0OUT to include writing to this port in DMap or ACB operations.
2. Input subsystem is accessible in DMap mode and following channels are defined:
   0..31 – input current for lines 0..31 (raw, 32-bit)
   32..63 – input voltage for lines 0..31 (raw, 32-bit)
   64 – over-current status of port 0
   65 – over-current status of port 1
   66 – actual value driven to port 0 (written value less lines with tripped circuit-breakers)
   67 – actual value driven to port 1

### *4.29.2 DqAdv432Reengage*

**Syntax:**

```
int DqAdv432Reengage(int hd, int devn, uint32 data)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 data` | Bitwise mask of circuit breakers to reset, 1= reset |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-432 or DIO-433 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function resets tripped circuit breakers. A 1 in bit position 0 resets breaker for channel 0, etc.

All application programs that use the circuit breaker features should call this function at startup. This will ensure that any breakers left in the tripped state by a previous application are properly reset.

**Note:**

This function is new starting with revision 4.9.0.204 and 4.8.0.35. In the previous revisions, the circuit breakers were reset by writing to them again with `DqAdv40xWrite().` Starting with the above mentioned revisions, the circuit breakers are reset with `DqAdv432Reengage()`.

Please note that `DqAdv432Reengage()` also sets a Boolean value inside the IOM. If the user has an application that was written to the older API (before the above mentioned revisions), then the IOM firmware and DAQLib library will behave as before, provided that the user does not call this function. Once the user calls `DqAdv432Reengage()` , the new circuit breaker functionality will be in effect.

### *4.29.3 DqAdv432SetAll*

**Syntax:**

`int DqAdv432SetAll(int hd, int devn, pDQDIO432DATAOUT data)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `pDQDIO432DATAOUTN data` | Pointer to store input data from DIO-432 |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-432 or DIO-433 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up most of DIO-432/433 configuration parameters.

Configuration parameters are set in the `DQDIO432DATAOUTN` structure, which is defined as:

```
/* structure for SETPARAM data (WRITE) */
typedef struct {
    uint32 cfgmask;   // What parameters to set
    uint32 cfg;       // 00 Set disconnection mode
    uint32 discfg;    // 08 Override circuit breaker
    uint32 adcspd;    // 0c Set ADC Timing
    uint32 portocm;   // 10 Set over-current detection mask
    uint32 portucm;   // 14 Set under-current detection mask
    uint32 rdcnt;     // 18 Set number of "failed" samples prior breaker engagement
    uint32 disdiv;    // 24 66MHz divider for the re-enable counter
} DQDIO432DATAOUT, *pDQDIO432DATAOUT;
```

- <`cfgmask`> flags select what parameter should be written. Following flags are defined:
  - `#define DQDIO432_CFGSET      (1L<<0)    // =1 to set "cfg" parameter`
  - `#define DQDIO432_DISCFGSET   (1L<<1)    // =1 to set "discfg" parameter`
  - `#define DQDIO432_ADCSPDSET   (1L<<2)    // =1 to set "adcspdset" parameter`
  - `#define DQDIO432_PORT0OCMSET (1L<<3)    // =1 to set "port0ocmset" parameter`
  - `#define DQDIO432_PORT0UCMSET (1L<<4)    // =1 to set "port0ucmset" parameter`
  - `#define DQDIO432_RDCNTSET    (1L<<5)    // =1 to set "rdcntset" parameter`
  - `#define DQDIO432_DISDIVSET   (1L<<6)    // =1 to set "disdivset" parameter`

- <`cfg`> selects the disconnection mode. Set the bit corresponding to the desired channel to 0 for user re-enable mode (i.e., to re-enable output on the channel after overcurrent conditions, the user must write to the output port using `DqAdv40xWrite()`. Set 1 to allow logic to re-enable output after the counter value programmed in <disdiv> expires. (This is a countdown counter. Each tick takes 15.15ns. Use the `DqAdv432GetAll()` function to retrieve the current value of the re-enable counter divider).

- <`discfg`> overrides the circuit breaker. Set the corresponding bit to 1 for override of the over- and undercurrent detection circuitry. Thus, if the corresponding bit in <`discfg`> is set to 1, the circuit-breaker cannot disconnect that channel. However, the user can still read over- and undercurrent detection status in `DqAdv432GetAll()`.

- <`adcspd`> selects the acquisition rate for circuit breaker ADCs. You can select the acquisition rate from the following list (applies to whole layer):

| Code | ADC speed | Rate per channel | P-P noise, mA @ +/-2A range | Break time, ms with 4 consecutive measurements, ms |
|---|---|---|---|---|
| 1 | 3.52kHz | 190Hz | 5.5 | 20 |
| 2 | 1.76kHz | 130Hz | 5 | 25 |
| 3 | 880Hz | 85Hz | 3.6 | 40 |
| 4 | 440Hz | 45Hz | 2.9 | 70 |
| 5 | 220Hz | 22Hz | 1.25 | 130 |
| 6 | 110Hz | 12Hz | 1.1 | 250 |
| 7 | 55Hz | 6.5Hz | 0.84 | 500 |
| 8 | 27.5Hz | 3.2Hz | 0.77 | 1000 |
| 9 | 13.75Hz | 1.6Hz | 0.36 | 2000 |
| 15 | 6.875Hz | 0.8Hz | 0.26 | 4000 |

- <`portocm`> and <portucm> specify bit-wise over- and under-current detection mask. Only bits set to 1 in these parameters will be reported back in case of over/under-current conditions in the <`portocm`> and <`portucm`> fields of the `DQDIO432DATAIN` structure filled in a `DqAdv432GetAll()` call.

- <`rdcnt`> specifies the number of consecutive samples above the selected threshold to be acquired before disconnecting the line. Valid values are 1 through 31; the default value is 4.

- <`disdiv`> specifies the number of 15.15ns clocks to wait before the logic will try to re-enable a tripped channel. <`cfg`> selects which channels need to be re-enabled automatically.

**Note:**

1. For DMap operations, these parameters are mapped into special channel numbers of DQ_SS0IN:
   Channels [0..31] return the last measured current (16 MSBs) on lines 0..31
   Channels [32..63] return the last measured voltage (16 MSBs) on lines 0..31

### *4.29.4 DqAdv432SetLimit*

**Syntax:**

```
int DqAdv432SetLimit(int hd, int devn, int limitid,
double limitvalue)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int limitid` | Channel to set limit for (0…31) |
| `double limitvalue` | Value of the limit in Amps (-2.0…2.0) |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-432 or DIO-433 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up over-current limits for DIO-432/433 channels. Under-current limits are not supported.

**Notes:**

- While `limitvalue` can be set to values between -2.0A and +2.0A, the layer itself only supports 600mA of current per output line. Reverse current is a sign of incorrect connection and may cause layer failure.

### *4.29.5 DqAdv432SetPWM*

**Syntax:**

```
int DqAdv432SetPWM(int hd, int devn, uint32 period_us, int count,
pDQDIOPWM settings)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 period_us` | PWM period (length of the full cycle) |
| `int count` | Number of DQDIOPWM entries |
| `pDQDIOPWM settings` | Structure defining PWM settings |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-432 or DIO-433 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

DNA-432/433 supports normal, soft start, soft stop and PWM modes of operation. Soft start and soft stop modes are designed to prolong the life of incandescent indicator bulbs connected to the outputs of the layer. In normal mode, the output FET switches from closed state to open state and back. In soft modes, the FET switches on and off, increasing for a start duty cycle and decreasing for a stop duty cycle, until it reaches 100% or 0% duty cycle. In PWM mode, the layer output produces a pulse train of the selected period with the selected duty cycle.

<`period_us`> is selected for the whole layer. It defines the total PWM period.
<`count`> specifies the size of the array of `DQDIO432PWM` structures to follow.
<`pDQDIOPWM`> is a pointer to the array of `DQDIOPWM` structures.

To specify the mode of operation for each channel, the user should pass an array of one or more `DQDIOPWM` structures:

```
typedef struct {
      uint8   channel;           // channel number
      uint8   mode;              // mode of operation
      uint32  duty_cycle_length; // duty cycle or length of the full soft-
start/soft-stop
                                 // cycle
} DQDIOPWM, *pDQDIOPWM;
```

<`mode`> can be one of the following:

```
#define DQDIO432_PWM_DISABLED   0   // PWM and soft start disabled
#define DQDIO432_PWM_SOFTSTART  1   // use PWM to soft-start (0->1 transition)
#define DQDIO432_PWM_SOFTSTOP   2   // use PWM to soft-stop (1->0 transition)
#define DQDIO432_PWM_SOFTBOTH   3   // use PWM to soft-start and soft-stop
#define DQDIO432_PWM_MODE       4   // use output in PWM mode
#define DQDIO432_PWM_MODE_GATED 5   // use output in gated PWM mode(logic >=
02.11.0A)
```

In the soft-start/soft-stop mode, <`duty_cycle_length`> defines the length of full soft-start/soft-stop cycle. The <`duty_cycle_length` > should be no more than 255 * <`period_us`>.
In PWM mode, <`duty_cycle_length` > defines the duty cycle on the output with 1/256 resolution (8-bit).

For the DIO-432, a duty_cycle_length of 0 corresponds to 1/256% and 255 to 100%. It is not possible to directly set duty cycle to 0% on a DIO-432. To attain 0% duty cycle with a DIO-432, set the `PWM_MODE` for the channel to `DQDIO432_PWM_DISABLED` and set the corresponding output to 0 using the `DqAdv40xWrite()` function.

For the DIO-433, a `duty_cycle_length` of 0 corresponds to 255/256% (99.6%) and 255 to 0%. It is not possible to directly set duty cycle to 100% on a DIO-433. To attain 100% duty cycle with a DIO-433, set the `PWM_MODE` for the channel to `DQDIO432_PWM_DISABLED` and set the corresponding output to 1 using the `DqAdv40xWrite()` function.

Gated PWM mode(5) operates like PWM mode(4) except that setting the corresponding output to 0 using the `DqAdv40xWrite()` function will turn the output OFF and setting the output to 1 will turn the PWM signal ON.

## *4.30 DNA-DIO-448 layer*

### *4.30.1 DqAdv448Read*

**Syntax:**

```
int DqAdv448Read(int hd, int devn, int read_db, uint32
data[DQ_DIO448_PORTS])
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int read_db` | Read channel levels after debouncing |

**Output:**

| | |
|---|---|
| `uint32 data[2]` | Pointer to two 32-bit array to store channel data |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-448 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

The function reads the current state of digital ports. The DIO-448 layer has two digital ports, each 24-bits wide packed into 32-bit words.

<`read_db`> selects between reading current data (FALSE == 0) and debounced value (TRUE=1). Debouncer delay is programmed using DqAdv448SetDebouncer().

<`data[2]`> array to store two uint32s, representing the current state of digital inputs for ports 0 and 1 (24 lines per port)

**Note:**

### *4.30.2 DqAdv448ReadAdc*

**Syntax:**

```
int DqAdv448ReadAdc(int hd, int devn, uint32 clsize, uint32* cl,
                    uint32* rdata, double* vdata, uint32* status)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 clsize` | Channel list size |
| `uint32 *cl` | Channel list |

**Output:**

| | |
|---|---|
| `uint32* rdata` | Raw data from A/D converter |
| `double* vdata` | Array to store measured voltage on the specified channels |
| `uint32* status` | Pointer to store status word from the layer (DIO, ISO lines), NULL if not required (reserved) |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-448 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function requests conversion results for each channel specified in the channel list and stores them alongside converted data.

The following channels are defined (use them for DMap operations as well):

```
DQL_CHAN448_PWR0        0x30  // user power 0/TCPOS
DQL_CHAN448_PWR1        0x31  // user power 1/TCNEG
DQL_CHAN448_GNDPIN      0x32  // user ground pin
DQL_CHAN448_VREF25      0x33  // Vref 2.500V (reserved)

DQL_CHAN448_ADC0        0x40  // Normal channels 0x40..0x6F
```

Note:

Reading status information delays a reply to the call and is currently not implemented. Pass NULL as a <`status`> if you don’t want status to be read from isolated side logic.

### *4.30.3 DqAdv448SetAll*

**Syntax:**

```
int  DqAdv448SetAll(int hd, int devn, pDQDIO448DATAOUT pdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `pDQDIO448DATAOUT pdata` | Pointer to parameters to set |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-448 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up the parameters specified in <`pdata`>. The first word in this structure is a bit field <`cfgmask`> that specifies what parameters in the structure actually contains value and have to be written into the layer hardware.

**Note:**

This function is reserved for future functionality extension.

### *4.30.4 DqAdv448SetLevels*

**Syntax:**

```
int DqAdv448SetLevels(int hd, int devn, uint32 clsize, uint32* cl,
float l_low, float l_high)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 clsize` | Channel list size |
| `uint32* cl` | Channel list, channels 0..0x2f are valid |
| `float l_low` | Low logic level (0V..+30V) |
| `float l_high` | High logic level (0V..+30V) |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-448 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up low and high levels for the digital comparator circuitry for each channel specified in the channel list.

Upon startup, the firmware programs low and high levels from the values stored in E$^2$PROM separately for each 24-bit port. These default values are 5.6V for low and 11.25V for high level. The hysteresis is programmed to change the detected logic level from low to high when the input signal is above low and high limits. The logic level changes back to low when the input level is below low level. If the signal is below high level but above low level, the detector keeps the previous logic level. This feature increases immunity of the layer to digital noise in the input lines.

**Note:**

### *4.30.5 DqAdv448SetDebouncer*

**Syntax:**

```
int  DqAdv448SetDebouncer(int hd, int devn, uint32 clsize, uint32*
cl, uint32 debouncer)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 clsize` | Channel list size |
| `uint32* cl` | Channel list, channels 0..0x2f are valid |
| `uint32 debouncer` | 16-bit debouncer value in 100us intervals |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-448 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up debouncing time for the digital comparator circuitry for each channel specified in the channel list.

Upon startup, the firmware programs low and high levels. The hysteresis is programmed to change the detected logic level from low to high when the input signal is above low and high limits.

The debouncer enhances this capability by requiring the signal to stay above high or below low logic levels for a certain time before input can change logic state. The debouncer is a 16-bit value that defines the time (in 100us intervals) for which the logic level of each channel should stay stable before the register changes state. For example, a value of 10000 means that the signal level on that channel should stay for 1s before a debounced value will change state.

**Note:**

To access debounced values, use channel 2 for port 0 and channel 3 for port 1 in DMap mode. In DqAdv448Read() set `<read_db>` to TRUE to read debounced port values. If debouncing delay is set to 0, debounced values are equal to the current reading.

## 4.31 DNA-DIO-449 layer

### 4.31.1 DqAdv449ConfigEvents

**Syntax:**

```
int DAQLIB DqAdv449ConfigEvents(int handle, int devn,
event449_t event, uint32 port0_mask, uint32 port1_mask)
```

**Command:**

Specify what asynchronous notification events user wants to receive from the layer

**Input**

| | |
|---|---|
| `int handle` | Async handle to the IOM received from `DqAddIOMPort()` |
| `int devn` | Device number |
| `event449_t event` | Event type |
| `uint32 port0_mask` | Enables for input lines 23-0 or periodic rate divider |
| `uint32 port1_mask` | Enables for input lines 47-24 or optional divider reset |

**Output**

None

**Returns**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-449 |
| `DQ_BAD_PARAMETER` | configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description**

This function sets up events and their parameters, which program firmware to send asynchronous event notification packets when conditional events occur.

**`event449_t event:`**

`Event449_t event` is defined as:

```
typedef enum {
    EV449_CLEAR     = 0x1000,   // clear all events
    EV449_DI_CHANGE = 0x101,    // digital input change of state
    EV449_PERIODIC  = 0x102     // periodic event
} event449_t;
```

where

- `EV449_CLEAR:` clears all events that have previously been set along with runtime variables and accumulated events
- `EV449_DI_CHANGE:` when the configured data input line(s) switch states, an event packet will be sent (event packet description is provided below)

- `EV449_PERIODIC`: an event packet (as described below) will be sent continuously at the specified periodic rate. You can optionally reset when the periodic packet is sent by setting `port1_mask` to 1. Periodic events are supported in PowerDNA version 4.10.0.24 and later

When specifying periodic events, typical usage is to first set up `EV449_DI_CHANGE` to configure packets being sent on specific DI state changes and then set up `EV449_PERIODIC` to periodically send a packet with the 1/0 state for each DI pin.

Initially all events should be cleared with the following call:

```
ret = DqAdv449ConfigEvents(a_handle, devn, EV449_CLEAR, 0, 0);
```

**`port0_mask / port1_mask:`**

`port0_mask` and `port1_mask` are defined as follows for `EV449_DI_CHANGE` or `EV449_PERIODIC`:

| Parameter | Usage with `EV449_DI_CHANGE` | Usage with `EV449_PERIODIC` |
|---|---|---|
| `port0_mask` | 24 LS bits<br>1 in the corresponding bit position enables change of state events for input lines 23-0. | Divider to set the periodic event rate<br>Divider is calculated by multiplying DIO-449 66MHz base rate by period interval.<br>For example setting periodic events to occur at 1 s intervals:<br>`port0_mask = (uint32)(DQ_DIO449_BASE*(PERIOD_INTERVAL))-1;`<br>where `DQ_DIO449_BASE = 66000000` and `PERIOD_INTERVAL=1`.<br>UEI recommends periods >100 ms. |
| `port1_mask` | 24 LS bits<br>1 in the corresponding bit position enables change of state events for input lines 47-24 | Configurable reset of periodic event<br>0: no reset will occur<br>1: enables a reset of the periodic event counter upon an asynchronous event |

**Event packet description:**

Once events are configured and enabled, the firmware sends back a packet when the specified asynchronous event occurs. For periodic events, the packet continues to be sent at the specified periodic rate.

Events for all board types contain the following structure, where `data[]` is cast to the specific event information for the board type being used (in this case the DIO-449):

```
/* DQCMD_EVENT structure */
typedef struct {
    uint8   dev;                    // device
    uint8   ss;                     // subsystem
    uint16  size;                   // data size
    uint32  event;                  // event type
    uint8   data[];                 // data
} DQEVENT, *pDQEVENT;
```

Where:

`<size>` field contains sizeof(`EV449_ID`)+length.

`<event>` contains the type of the event, e.g. `EV449_DI_CHANGE` or `EV449_PERIODIC`

`<data[]>` for `DQEVENT` containing `EV449_DI_CHANGE` or `EV449_PERIODIC`, `data` maps to the following flexible structure:

```
// Event data for 449 layer
typedef struct {
    uint32 chan;            // channel information
    uint32 evtype;          // type of the event
    uint32 tstamp;          // timestamp of event
    uint32 size;            // size of the following data in bytes
    uint32 data[6];         // see description below
} EV449_ID, *pEV449_ID;
```

The timestamp (`tstamp`) returned is different depending whether the `evtype` is `EV449_DI_CHANGE` or `EV449_PERIODIC`:

- For `EV449_DI_CHANGE`, the `tstamp` represents the time when the change of state occurred on the DI pin.
- For `EV449_PERIODIC`, the `tstamp` represents the time when the packet was sent.

The event `data[]` of the returned `pEV449_ID` structure consists of 6 uint32 values (24LS bits valid) as follows:

| Parameter | For `EV449_DI_CHANGE` event | For `EV449_PERIODIC` event |
|---|---|---|
| `data[0]` | Rising edge events for input lines 23-0, 1=rising edge event has occurred on corresponding input line | Debounced state of digital inputs for port 0 (input lines 23-0) |
| `data[1]` | Falling edge events for input lines 23-0, 1=falling edge event has occurred on corresponding input line. | Immediate state of digital inputs for port 0 (input lines 23-0) |
| `data[2]` | Rising edge events for input lines 47-24, 1=rising edge event has occurred on corresponding input line. | Debounced state of digital inputs for port 1 (input lines 47-24) |
| `data[3]` | Falling edge events for input lines 47-24, 1=falling edge event has occurred on corresponding input line. | Immediate state of digital inputs for port 1 (input lines 47-24) |
| `data[4]` | Event occurred, logical OR of `data[0]` and `data[1]` | Change of state data<br>for port 0 (input lines 23-0)<br>if `EV449_DI_CHANGE` is also configured |
| `data[5]` | Event occurred, logical OR of `data[2]` and `data[3]` | Change of state data<br>for port 1 (input lines 47-24)<br>if `EV449_DI_CHANGE` is also configured |

Note that you use the `DqRtAsyncEnableEvents()` API to enable events and the `DqCmdReceiveEvent()` API to receive events.

Event packets are handled by `DqCmdReceiveEvent()` in terms of converting data fields from big endian (network data representation) to little endian if used with Intel-based host computer while EV449_ID needs to be converted by the user with the following code:

```
// shuffle bytes big endian->little endian in place
void ntoh_pEv449(pEV449_ID pEv449) {
    uint32 i;

    pEv449->chan = ntohl(pEv449->chan);
    pEv449->evtype = ntohl(pEv449->evtype);
    pEv449->size = ntohl(pEv449->size);
    pEv449->tstamp = ntohl(pEv449->tstamp);
    for (i = 0; i < pEv449->size/sizeof(uint32); i++)
        pEv449->data[i] = ntohl(pEv449->data[i]);
    return;
}
```

The reason for this is that `DqCmdReceiveEvent()` returns all different types of layer-specific events and as implemented doesn’t do this conversion.

**Note:**
If you need to verify the sequence of received events, you can use `DqCmdReceiveEventPkt()` instead of the `DqCmdReceiveEvent()`API to retrieve the packet ID counter (`dqCounter`).

### *4.31.2 DqAdv449GetModeGainLevels*

**Syntax:**

```
int DqAdv449GetModeGainLevels(int hd, int devn, uint8* mode_gain,
float* l_level, float* h_level)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `uint8* mode_gain` | Pointer to array of 48 uint8, NULL if not required. (See below) |
| `float* l_level` | Pointer to array of 48 float, low logic level voltage, NULL if not required. |
| `float* h_level` | Pointer to array of 48 float, high logic level voltage, NULL if not required. |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-449 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Retrieves state of AC/DC configuration, gain setting and logic levels for all 48 input lines. The pointer mode_gain should point to an array of 48 uint8, pointer may be NULL if mode and gain information are not required. Bits 1 and 0 of mode_gain elements are the gain setting ( see defined constants `DQ_DIO449_GAIN1` .. `DQ_DIO449_GAIN10` in powerdna.h)
Bit 2 is the AC/DC selection: 0=AC, 1=DC. See `DQ_DIO449_GET_MODE_DC` in powerdna.h.

**Note:**

### *4.31.3 DqAdv449Read*

**Syntax:**

```
int DqAdv449Read(int hd, int devn, int read_db, uint32
data[DQ_DIO449_PORTS])
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int read_db` | Read channel levels after debouncing |

**Output:**

| | |
|---|---|
| `uint32 data[2])` | Pointer to two 32-bit array to store channel data |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-449 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

The function reads the current state of digital ports. The DIO-449 layer has two digital ports, each 24-bits wide packed into 32-bit words.

- `<read_db>` selects between reading current data (FALSE == 0) and debounced value (TRUE==1). Debouncer delay is programmed using `DqAdv449SetDebouncer()`.
- `<data[]>` array to store two uint32s, representing the current state of digital inputs for ports 0 and 1 (24 lines per port)

**Note:**

### *4.31.4 DqAdv449ReadAdc*

**Syntax:**

```
int DqAdv449ReadAdc(int hd, int devn, uint32 clsize, uint32* cl,
uint32* rdata, double* vdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 clsize` | Channel list size |
| `uint32 *cl` | Channel list |

**Output:**

| | |
|---|---|
| `uint32* rdata` | Raw data from A/D converter |
| `double* vdata` | Array to store measured voltage on the specified channels |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-449 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function requests ADC conversion results for each channel specified in the channel list and stores them alongside converted data.
The following channels are defined (use them for DMap operations as well):

```
DQDIO449_PORT0_IND        0x0  // read debounced dio port0 as an ADC channel
DQDIO449_PORT1_IND        0x1  // read debounced dio port1 as an ADC channel
DQDIO449_PORT0_IN         0x2  // read dio port0 as an ADC channel
DQDIO449_PORT1_IN         0x3  // read dio port1 as an ADC channel

DQL_CHAN449_ADCCHAN_START  0x40 // ADC channels 0x40..0x6F
```

Note:

Minimum array length of `rdata` and `vdata` is the value in `clsize.`

### *4.31.5 DqAdv449ReadBlock*

**Syntax:**

```
int DqAdv449ReadBlock(int hd, int devn, uint32 channel,
uint16* bData)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 channel` | Channel number, 0..47 |

**Output:**

| | |
|---|---|
| `uint16* bData` | returned array of 257 values |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-449 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads a block of the 256 most recent ADC readings from the given `channel`. The 257th value is the gain setting which determines the scaling factors to be applied to the data. The returned gain setting is in the range of 0..3 as defined in the following definitions:

#define DQ_DIO449_GAIN1 (0) // gain of 1
#define DQ_DIO449_GAIN2 (1) // gain of 2
#define DQ_DIO449_GAIN5 (2) // gain of 5
#define DQ_DIO449_GAIN10 (3) // gain of 10

To convert readings to volts use the following formulae.

For gain of 1:
(bData * DQ_DIO449_STEP) - DQ_DIO449_OFFSET
For gain of 2:
(bData * DQ_DIO449_STEP_2) - DQ_DIO449_OFFSET_2
For gain of 5:
(bData * DQ_DIO449_STEP_5) - DQ_DIO449_OFFSET_5
For gain of 10:
(bData * DQ_DIO449_STEP_10) - DQ_DIO449_OFFSET_10

**Note:** See powerdna.h for other defines

### *4.31.6 DqAdv449SetAveragingMode*

**Syntax:**

```
int DqAdv449SetAveragingMode(int hd, int devn, uint32 avg_port0,
uint32 avg_port1, uint32* dcmode)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 avg_port0` | Bits 0..23 set averaging mode for lines 0..23 |
| `uint32 avg_port1` | Bits 0..23 set averaging mode for lines 24..47 |
| `uint32* dcmode` | pointer to Boolean to set DC averaging mode |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-449 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This command allows selecting the mode of averaging that is used for the digital inputs.
Using all periods in the buffer provides a more accurate reading for steady state signals, but will delay detection of the change of state due to the averaging.

In AC mode, when fast change-of-state detection is required it is recommended to use data from the one last period only. In most of other cases and especially when acquiring low level DC signals averaging all periods greatly improves data quality.

- For bits 0..23 in `avg_port0` and `avg_port1`:
  =0 - (default) calculate RMS/DC based on all periods in 256 sample buffer. (slower,accurate)
  =1 - calculate RMS/DC for the last acquired complete period only. (faster, less accurate)

- For `*dcmode`: pointer to boolean to set DC averaging mode for all inputs in DC mode,
  false= use last immediate value(faster), `avg_portN` parameters are ignored.
  true= use averaged value (slower), default=true.
  (Pointer may be NULL if unused, DC averaging mode will remain unchanged).

**Note:**

### *4.31.7 DqAdv449SetDebouncer*

**Syntax:**

```
int DqAdv449SetDebouncer(int hd, int devn, uint32 clsize, uint32* cl,
uint32 debouncer)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 clsize` | Channel list size |
| `uint32* cl` | Channel list, channels 0..0x2f are valid |
| `uint32 debouncer` | 12-bit debouncer value in 100us intervals |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-449 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up debouncing time for the digital comparator circuitry for each channel specified in the channel list.

Upon startup, the firmware programs low and high levels. The hysteresis is programmed to change the detected logic level from low to high when the input signal is above low and high limits.

The debouncer enhances this capability by requiring the signal to stay above high or below low logic levels for a certain time before input can change logic state. The debouncer is a 12-bit value that defines the time (in 100us intervals) for which the logic level of each channel should stay stable before the register changes state. For example, a value of 1000 means that the signal level on that channel should stay for 0.1s before a debounced value will change state.

**Note:**

To access debounced values, use channel 2 for port 0 and channel 3 for port 1 in DMap mode. If debouncing delay is set to 0, debounced values are equal to the current reading.

### 4.31.8 DqAdv449SetGDacs

**Syntax:**

```
int DqAdv449SetGDacs(int hd, int devn, uint32 clsize, uint32* cl,
uint32 debouncer)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 dlsize` | DAC list size, minimum 1, maximum 48 |
| `uint32* dl` | DAC list, combined channel number and 8-bit DAC value, for channels 0..47 (0..0x2F) |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-449 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets Guardian DAC output voltage level for any or all 48 input lines.

The `dl` array (DAC list), combines the channel number and an 8-bit DAC voltage level setting. Channel numbers run from 0x0..0x2F. Combine the channel number and DAC voltage setting using the `DQ_DIO449_LNCL_G_DAC()` helper macro, (e.g. the following will set the DAC on channel 0 with an output voltage of ~168 V: `dl[0] = 0 | DQ_DIO449_LNCL_G_DAC(127)`).

When using the Guardian DACs to inject voltages for self-test (all Guardian Mux settings =1), use the following table to determine the DAC setting and resulting voltage reading (e.g. resulting voltages can be read with `DqAdv449ReadAdc()`).

The following values are not calibrated, so the readings are approximate.

| DAC setting | Gain index (Gain setting) | DC Voltage reading |
|---|---|---|
| 0-127 | 0 (1) | 0 - 168 |
| 0-63 | 1 (2) | 0 - 83 |
| 0-23 | 2 (5) | 0 - 31 |
| 0-11 | 3 (10) | 0 - 15 |

**Note:**

If field wiring is left connected during the self-test, the input signal will affect the voltage read using the `DqAdv449ReadAdc()` API, (i.e. the input voltage will contribute to the voltage read by approximately (`0.22*inputVoltage`), depending on source impedance.)

To connect the Guardian DAC to the digital input channel, use the `DqAdv449SetGMux()` API.

To set the gain of the digital input channel(s), use the `DqAdv449SetLevels()` API

### *4.31.9 DqAdv449SetGMux*

**Syntax:**

```
int DqAdv449SetGMux(int hd, int devn, uint16 gme4, uint32  muxval[2])
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint16 gme4` | Guardian Mux Enable, see below for bit field descriptions |
| `uint32 muxval[2]` | Selects between self-test modes, either hard or soft or override, has no effect when corresponding gme4 bit is 0. |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-449 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets up Guardian Muxes for 48 digital input lines for muxing in / out DIO-449 on-board self-test / diagnostic circuitry:

- `gme4` is used to mux in/out the Guardian DAC reference voltage and/or a -15 V reference
- `muxval[]` selects connecting the Guardian DAC reference voltage directly or through a 33 kΩ resistor

**Note**:

Bit descriptions for `gme4` and for `muxval[]` are provided below:

**`uint16 gme4` - Guardian Mux Enable:**

- **`gme4` bits 0..11** enables a reference voltage from the Guardian DAC to be injected into the input circuit for self-test purposes; each bit controls a bank of 4 channels.
  For example:
  `gme4=0x1` connects Guardian DACs on channels 0..3;
  `gme4=0x2` connects Guardian DACs on channels 4..7;
  `gme4=0x3` connects Guardian DACs on channels 0..7;
  `gme4=0x4` connects Guardian DACs on channels 8..11;
  etc.
  By default bits 0..11 are 0, which sets up all channels for standard DI use (the Guardian DACs are not connected).

- **`gme4` bit 12** enables a -15 V supply to be connected to the input circuit (useful for testing open circuits/monitoring contacts). This bit controls the -15 V state for all channels.
  Setting **`gme4` bit 12** to 0 connects the -15 V source to all channels.
  Setting **`gme4` bit 12** to 1 disconnects the -15 V source to all channels.
  The -15 V is connected by default. When the -15 V supply is enabled (in-circuit), this results in all open circuit inputs measuring approximately -3.3 V. When disabled, the ADC reading of open circuit inputs will read approximately 0 volts.

**`uint16 muxval[]`** –

- Selects connecting the Guardian DAC reference voltage directly (corresponding `muxval[]` bit=1) or through a 33 kΩ resistor (corresponding `muxval[]` bit=0).
  Note that the Guardian Mux Enable bit (`gme4`) for the corresponding channel must be set for `muxval` to have an effect.
- the 24 LSBs of **`muxval[0]`** map to DI input lines 23..0
- the 24 LSBs of **`muxval[1]`** map to DI input lines 47..24

When a channel's `gme4` bit is 1, the `muxval[]` bit for that channel identifies which of the two self-test functions will drive the input (drive directly or drive through a 33 kΩ resistor):

- Hard override (setting `muxval[]` bit/channel=1) results in the Guardian DAC for the corresponding channel to drive the input line directly.
  **This is the recommended test mode.**
- Soft override (setting `muxval[]`bit/channel =0) results in the Guardian DAC for the corresponding channel to drive the input line through a 33 kΩ resistor (not the recommended test mode).

Examples:

- To connect all Guardian DACs directly to input circuits on all channels and leave the -15 V supply connected on all channels:
```
gme4 =0x0FFF; // bit 12=0; bits 11..0 are 1
muxval[0]=0xFFFFFF;
muxval[1]=0xFFFFFF;
```
- To connect all Guardian DACs directly to input circuits on all channels and disconnect the -15 V supply on all channels:
```
gme4 =0x1FFF; // bit 12=1; bits 11..0 are 1
muxval[0]=0xFFFFFF;
muxval[1]=0xFFFFFF;
```
- To disconnect -15 V supply on all channels and disconnect Guardian DACs on all channels:
```
gme4 =0x1000;
```

To set the voltage level that the Guardian DAC will drive, use the `DqAdv449SetGDacs` API.

### 4.31.10 DqAdv449SetLevels

**Syntax:**

```
int DqAdv449SetLevels(int hd, int devn, uint32 clsize,
uint32* cl, float l_low, float l_high)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 clsize` | Channel list size |
| `uint32* cl` | Channel list, see below |
| `float l_low` | logic "low" level from 0 V to 150 V |
| `float l_high` | logic "high" level from 0 V to 150 V |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-449 |
| `DQ_BAD_PARAMETER` | l_low or l_high is negative in AC mode or is greater than the maximum value allowed by the gain range |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets gain, mode and low and high logical levels for each channel in the channel list.

To fill in the channel list, use the `DQ_LNCL_GAIN` helper macro.
For example:

```
cl[0] = (DQL_CHAN449_ADCCHAN_START + (channel number))  |
          DQ_LNCL_GAIN(DQ_DIO449_GAIN1 | DQ_DIO449_MODE_DC);
```

Constants for `DQ_LNCL_GAIN` helper macro include:

| | |
|---|---|
| `DQ_DIO449_GAIN1` | // code = 0, gain of 1, max input range = +/-215.2 VDC |
| `DQ_DIO449_GAIN2` | // code = 1, gain of 2, max input range = +/-107.6 VDC |
| `DQ_DIO449_GAIN5` | // code = 2, gain of 5, max input range = +/-43.04 VDC |
| `DQ_DIO449_GAIN10` | // code = 3, gain of 10, max input range = +/-21.52 VDC |
| `DQ_DIO449_MODE_DC` | |
| `DQ_DIO449_MODE_AC` | |

**Note:**

Internally, this function translates the floating point input value to the integer value used by the hardware. This translation process requires the gain and the AC/DC setting in order to calculate the correct integer value, which will make the digital inputs switch at the desired levels. The gain and mode values are stored by the layer and used to perform the acquisition of the data. If a subsequent `DqAdv449ReadAdc()` is performed using <u>different</u> gain and mode values, those changes will affect the low and high logical level settings. In that case the logic level settings will need to be updated by calling this function again.

The logic level must be appropriate for all channels specified in the channel list. Errors will be returned if negative values are used for AC channels or if the voltage is higher than the maximum voltage allowed for the gain range. Multiple `DqAdv449SetLevels()`calls must be used if mixed AC and DC channels and differing gain ranges are used.

## *4.32 DNx-MUX-461 layer*

### *4.32.1 DqAdv461SetChannel*

**Syntax:**

```
int DqAdv461SetChannel (int hd, int devn, uint32 channel_num,
uint32 relay_select, uint32 dmm_mode, uint32 sync)
```

**Command:**

Select a multiplexer channel and connect output to DMM.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 channel_num` | Select channel 0…12 |
| `uint32 relay_select` | Enable channel on bus A and/or bus B.<br>In 4-wire resistance mode, select one of:<br>`DQ_MUX461_SETCHAN_RL_DISABLE //open relay on bus A and B`<br>`DQ_MUX461_SETCHAN_RL_A_B    //close relay on bus A and B`<br>In all other modes, select one of:<br>`DQ_MUX461_SETCHAN_RL_DISABLE //open relay on bus A and B`<br>`DQ_MUX461_SETCHAN_RL_A       //close relay on bus A`<br>`DQ_MUX461_SETCHAN_RL_B       //close relay on bus B` |
| `uint32 dmm_mode` | Select one of the following measurement modes:<br>`DQ_MUX461_SETCHAN_DMM_DISABLE //disable all relays`<br>`DQ_MUX461_SETCHAN_DMM_V       //voltage measurement mode`<br>`DQ_MUX461_SETCHAN_DMM_I       //current measurement mode`<br>`DQ_MUX461_SETCHAN_DMM_RES2    //2-wire resistance mode`<br>`DQ_MUX461_SETCHAN_DMM_RES4    //4-wire resistance mode` |
| `uint32 sync` | Enable one or more SYNC pins by ORing in these constants:<br>`DQ_MUX461_SETCHAN_SYNC_IN     //enable SYNC IN pin as gate for switching relays`<br>`DQ_MUX461_SETCHAN_SYNC_OUT    //enable SYNC OUT` |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not a MUX- |

| | |
|---|---|
| | 461 |
| `DQ_BAD_PARAMETER` | `channel_num` is too high, `relay_select` is invalid, or `dmm_mode` is invalid |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
This function configures multiplexer switches open or closed and enables sync in and sync out functionality for the write.

Only one channel may be connected to the output at any given time. In a 2-wire DMM mode, the possible channels are A0…A12, B0,…B12. In a 4-wire DMM mode, the possible channels are 0…12 (both Ax and Bx relays are closed). The function automatically closes the appropriate D relays for the selected `dmm_mode`.

**Notes:**

- By default, this function opens all relays before closing the programmed relays. The break-before-make timing can be changed using `DqAdv461SetCfg()`. You can optionally disable the break-before-make on D relays only.
- Configure sync modes and timing using `DqAdv461SetCfg()`.

### *4.32.2 DqAdv461ReadADC*

**Syntax:**

```
int DqAdv461ReadADC (int hd, int devn, pDQ461ADC adc_data);
```

**Command:**

Read diagnostic data from onboard ADC.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `pDQ461ADC adc_data` | Data structure containing ADC data |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not a MUX-414/418/461 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function returns diagnostic ADC data in a `DQ461ADC` structure:

```
typedef struct {
    double adc_24;      // voltage of internal 24V supply
    double adc_3_3;     // voltage of internal 3.3V supply
    double adc_5VR;     // internal relay driver voltage
    double adc_deg_c;   // ADC temperature in degrees C
    uint32 status;      // identical to status returned by DqAdv461ReadStatus()
    uint32 timestamp;   // timestamp
} DQ461ADC, *pDQ461ADC;
```

**Notes:**

- Before reading data, call `DqAdv461ReadADC(hd, devn, NULL)` once to initialize the ADC.

### *4.32.3 DqAdv461SetCfg*

**Syntax:**

`int DqAdv461SetCfg(int hd, int devn, pDQ461CFG cfg)`

**Command:**

Configure relay driver voltage, break-before-make, and sync modes.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `pDQ461CFG cfg` | Pointer to data structure containing configuration settings |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not a MUX-461 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function sets MUX-461 configuration with data contained in a `DQ461CFG` structure:

```
typedef struct {
    uint32 d_bbm_mode;              // enable break-before-make for "D" relays
    uint32 on_delay;                // time before next command is accepted
    uint32 off_delay;               // breaking time of break-before-make
    uint32 sync_in_mode;            // SYNC IN pin operation mode
    uint32 sync_in_polarity;        // Polarity of SYNC IN strobe
    uint32 sync_out_pw;             // SYNC OUT pulse length
    uint32 sync_out_mode;           // SYNC OUT pin operation mode
    uint32 sync_skip;               // release from SYNC IN waiting
} DQ461CFG, *pDQ461CFG;
```

- `<d_bbm_mode>` enables break-before-make on “D” relays. By default “D” relays switch on every `DqAdv461SetChannel()` command. If you are always using the same DMM mode, disable break-before-make to increase the life span of these relays. Note: break-before-make on “A” and “B” relays is always enabled.

| | |
|---|---|
| `DQ_MUX461_SETCFG_D_BBM_DISABLE` | Disable break-before-make on “D” relays. |
| `DQ_MUX461_SETCFG_D_BBM_ENABLE` | Enable break-before-make on “D” relays. |

- `<on_delay>` programs the time before next command is accepted in 100uS units, range 1..256. Default is 100uS.

- `<off_delay>` programs the breaking time of break-before-make in 100uS units, range 1..256. Default is 100uS.

- `<sync_in_mode>` programs the mode of operation for the SYNC IN pin. If the `DQ_MUX461_SETCHAN_SYNC_IN` flag is enabled in `DqAdv461SetChannel()`, the new relay state will not take effect until the sync_in mode and polarity conditions are met.

| | |
|---|---|
| `DQ_MUX461_SETCFG_SYNC_IN_MODE_LEVEL` | Trigger on a level. |
| `DQ_MUX461_SETCFG_SYNC_IN_MODE_EDGE` | Trigger on an edge. |

- `<sync_in_polarity>` programs the polarity of sync in strobe.

| | |
|---|---|
| `DQ_MUX461_SETCFG_SYNC_IN_POLARITY_LOW` | Check for low level or falling edge. |
| `DQ_MUX461_SETCFG_SYNC_IN_POLARITY_HIGH` | Check for high level or rising edge. |

- `<sync_out_pw>` programs the SYNC OUT pulse width for modes `DQ_MUX461_SETCFG_SYNC_OUT_MODE_HIGH_PULSE` and `LOW_PULSE`:

| | |
|---|---|
| `DQ_MUX461_SETCFG_SYNC_OUT_PW_1US` | 1 microsecond |
| `DQ_MUX461_SETCFG_SYNC_OUT_PW_10US` | 10 microseconds |
| `DQ_MUX461_SETCFG_SYNC_OUT_PW_100US` | 100 microseconds |
| `DQ_MUX461_SETCFG_SYNC_OUT_PW_1MS` | 1 millisecond |

- `<sync_out_mode>` programs the mode of operation for the SYNC OUT pin:

| | |
|---|---|
| `DQ_MUX461_SETCFG_SYNC_OUT_MODE_LOW` | Drive constant logic ‘0’ on SYNC OUT. |
| `DQ_MUX461_SETCFG_SYNC_OUT_MODE_HIGH` | Drive constant logic ‘1’ on SYNC OUT. |
| `DQ_MUX461_SETCFG_SYNC_OUT_MODE_HIGH_PULSE` | SYNC OUT is normally low, generates high pulse on relay write. |

| | |
|---|---|
| `DQ_MUX461_SETCFG_SYNC_OUT_MODE_LOW_PULSE` | SYNC OUT is normally high, generates low pulse on relay write. |
| `DQ_MUX461_SETCFG_SYNC_OUT_MODE_HIGH_ON_RDY` | SYNC OUT is normally low, goes high when relays are in ready state. |
| `DQ_MUX461_SETCFG_SYNC_OUT_MODE_LOW_ON_RDY` | SYNC OUT is normally high, goes low when relays are in ready state. |

Use the `DQ_MUX461_SETCHAN_SYNC_OUT` flag in `DqAdv461SetChannel()` to enable sync out functionality.

- `<sync_skip>:` clears the `DQ_MUX461_SETCHAN_SYNC_IN` flag from the previous `DqAdv461SetChannel()` command. This allows the new relay state to be written without waiting for the sync in strobe. `Sync_skip` automatically resets itself to 0 after clearing the flag.

| | |
|---|---|
| `DQ_MUX461_SETCFG_SYNC_NO_SKIP` | Normal operation: If `DQ_MUX461_SETCHAN_SYNC_IN` was used in a previous write, layer will continue waiting for SYNC IN to apply latest write. |
| `DQ_MUX461_SETCFG_SYNC_SKIP` | If `DQ_MUX461_SETCHAN_SYNC_IN` was used in a previous write, skip waiting for SYNC IN to apply latest write. |

### 4.32.4 DqAdv461ReadStatus

**Syntax:**

`int DqAdv461ReadStatus (int hd, int devn, pDQ461STATUS rstatus);`

**Command:**

Read relay positions and status.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `pDQ461STATUS rstatus` | Data structure containing relay status |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not a MUX-461 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function returns relay status information in a `DQ461STATUS` structure:

```
typedef struct {
    uint32 port0_write;     // last write
    uint32 port0_state;     // current state
    uint32 status;          // status word
    uint32 a_state;         // current state of A12:A0 relays (1=closed)
    uint32 b_state;         // current state of B12:B0 relays (1=closed)
    uint32 d_state;         // current state of D4:D1 relays (1=closed)
} DQ461STATUS, *pDQ461STATUS;
```

`<port0_write>` and `<port0_state>` are returned as:

- `Bit 31, DQ_MUX461_SIN`: 1 = wait for the selected SYNC IN strobe before applying the new relay settings. See `sync_in_mode` and `sync_in_polarity` in `DqAdv461SetCfg()`.
- `Bit 30, DQ_MUX461_D4`: 1 = turn on D4 relay.
- `Bit 29, DQ_MUX461_D3`: 1 = turn on D3 relay.
- `Bit 28, DQ_MUX461_B12:` 1 = turn on B12 relay.

  …
- `Bit 16, DQ_MUX461_B0:` 1 = turn on B0 relay.
- `Bit 15, DQ_MUX461_SOUT`: 1 = assert SYNC OUT. See `sync_out_mode` in `DqAdv461SetCfg().`
- `Bit 14, DQ_MUX461_D2`: 1 = turn on D2 relay.
- `Bit 13, DQ_MUX461_D1:` 1= turn on D1 relay.
- `Bit 12, DQ_MUX461_A12:` 1 = turn on A12 relay.

  …
- `Bit 0, DQ_MUX461_A0:` 1 = turn on A0 relay.

`<status>` is returned as:

- `Bit 17, DQ_MUX461_STS_ADCDR`: 1 = new unread data is ready from ADC
- `Bit 16, DQ_MUX461_STS_OVR`: 1 = overrun (a write occurred while busy)
- `Bit 3, DQ_MUX461_STS_BUSY`: 1 = state machine is busy
- `Bit 2, DQ_MUX461_STS_SYNCWAIT`: 1 = output state machine is waiting for the external SYNC IN trigger (if configured) or the internal “relays ready”
- `Bit 1, DQ_MUX461_STS_RDY`: 1 = relays are ready
- `Bit 0, DQ_MUX461_STS_DI_STS`: reports the logic state of the SYNC IN pin

**Notes:**

- The status word can also be accessed with `DqAdv461ReadADC()` if readback of relay positions is not required.

### *4.32.5 DqAdv461GetRelayCounts*

**Syntax:**

`int DqAdv461GetRelayCounts (int hd, int devn, pDQ461COUNT count);`

**Command:**

Read the number of relay cycles.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `pDQ461COUNT count` | Data structure containing relay counts |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not a MUX-461 |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function returns the number of times each relay has been energized. The counts are updated internally every 11 minutes. Relay count information is stored in a `DQ461COUNT` structure:

```
typedef struct {
    int32 a_count[DQ_MUX461_A_RELAY_COUNT];   // a_count[0]= # of A0 cycles, etc.
    int32 b_count[DQ_MUX461_B_RELAY_COUNT];   // b_count[0]= # of B0 cycles, etc.
    int32 d_count[DQ_MUX461_D_RELAY_COUNT];   // d_count[0]= # of D1 cycles, etc.
} DQ461COUNT, *pDQ461COUNT;
```

## 4.33 DNA-DIO-452/462/463 layers

All functions in this section apply to the DIO-462 and DIO-463. The `DqAdv462GetAll()` function is the only function in this section that may be used with a DIO-452.

Additional information regarding API for the DIO-452, DIO-462 and DIO-463 layers:

- use the `DqAdv40xWrite()` function to control the relays
- use `DqAdv40xReadLastWrite()` diagnostic function to retrieve the last value written to a layer

### 4.33.1 DqAdv462ReadAdc

**Syntax:**

```
int DqAdv462ReadAdc(int hd, int devn, uint32 CLSize, uint32* cl,
uint32* bdata, double* fdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 CLSize` | Channel list size |
| `uint32 *cl` | Channel list |

**Output:**

| | |
|---|---|
| `uint32* rdata` | Raw data from A/D converter |
| `double* vdata` | Array to store measured values from the specified channels |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-462 or DIO-463 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function requests conversion results for each channel specified in the channel list and stores them alongside converted data.

There are 12 relay channels on the DIO-462/3 with 5 measurements or subchannels possible for each channel. This makes a maximum of 60 entries in the channel list. The subchannels are #defined as follows:

```
DQ_DIO462_SUBCH_V_NO    (0)      // Vno , DC voltage on normally open contact
DQ_DIO462_SUBCH_I_AC    (1)      // Iac , AC current on common
DQ_DIO462_SUBCH_I_DC    (2)      // Idc , DC current on common
DQ_DIO462_SUBCH_V_NC    (3)      // Vnc , DC voltage on normally closed contact or
                                    unused terminal on DIO-463
DQ_DIO462_SUBCH_THERM   (4)      // temperature C
```

Use the `DQ_DIO462_MAKE_CL(CH,SUBCH)` helper macro to combine the channel and subchannel to make the entries in the channel list.

Note:

The DIO-462-800 is a version of the DIO-462 without the voltage monitoring capability on subchannels 0 and 3.

### *4.33.2 DqAdv462GetAll*

**Syntax:**

`int DqAdv462GetAll(int hd, int devn, pDQDIO462DATAIN data)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `pDQDIO462DATAIN data` | pointer to store all readable parameters from DIO-462 |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-452 , DIO-462 or DIO-463 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads available information from the DIO-462 or DIO-463 layer, except readback from the ADCs that are monitoring every channel. When used with a DIO-452, only the `portout` parameter is populated.

**Note:**

The DIO-462 layer provides multiple status parameters that may be accessed via a DqAdv462GetAll() function call. Information is returned in the pDQDIO462DATAIN type:

```
#typedef struct {
   int32 cfg;         // Report list of the disabled channels.
   int32 portout;     // Current value written to the output port
   int32 dissts;      // disable status
   int32 portocs;     // Over-current interrupt status
   int32 portucs;     // Under-current interrupt status
   int32 adcsts;      // Reports combined status of the ADC subsystem
   int32 adccfg[5];   // ADC configuration values

   int32 rdcnt;       // Current value of the consecutive read counter setting
} DQDIO462DATAIN, *pDQDIO462DATAIN;
```

*cfg* reports a list of the channels currently disabled by the circuit breaker. Bits 31:16 are sticky, they report whether the corresponding channel (15:0) was ever disabled since the last call to this function and bits 15:0 report currently disabled channels.
*portout* bits 11:0 reports the last values that were assigned to the output ports.
*dissts* bits 11:0 report the current disable status, bits 23:12 report sticky status, indicating a 1 if a channel has been disabled since the last DqAdv462GetAll().
*portocs* bits 11:0 contain a one for every channel for which an overcurrent condition was ever detected
*portucs* bits 11:0 contain a one for every channel for which an undercurrent condition was ever detected
*adcsts* report the combined status of the ADC system.
*rdcnt* number of overcurrent samples read from an ADC prior to disconnection (0-32). A higher number leads to longer delay and better noise/spike immunity.

### *4.33.3 DqAdv462SetAll*

**Syntax:**

`int DqAdv462SetAll(int hd, int devn, pDQDIO462DATAOUT pdata)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `pDQDIO462DATAOUT data` | pointer to the structure with initialization data for the DIO-462 |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-462 or DIO-463 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function allows setting any of the configurable parameters of the DIO-462 or DIO-463 layer.

**Note:**

The configuration should be set using DQDIO462DATAOUT type. For parameters that should be actually programmed on the layer, the corresponding bit in cfgmask should be set to 1 Unused and undefined bits should be set to 0.

```
* structure for SETPARAM data */
```

```
typedef struct {
    int32 cfgmask;         // bit field, show which of the following params are valid
    int32 cfg;             // Set disconnection mode
    int32 discfg;          // Override circuit breaker
    int32 portocm;         // Set over-current interrupt mask
    int32 portucm;         // Set under-current interrupt mask
    int32 rdcnt;           // Set number of "failed" samples prior breaker engagement
    int32 adccfg[5]        // ADC conversion control words
    int32 dcdccfg;         // Set frequency of DC-DC converters powering ADC’s
    int32 dcdcx[4];        // Set duty cycle and phase for the ADC DC-DC converters
} DQDIO462DATAOUT, *pDQDIO462DATAOUT;

// bit defines for the contents of DQDIO462DATAOUT.cfgmask
#define DQDIO462_CFGSET         (1L<<0)    // =1 if "cfg" contains valid data
#define DQDIO462_PORTOCMSET     (1L<<3)    // =1 if "portocm" contains valid data
#define DQDIO462_PORTUCMSET     (1L<<4)    // =1 if "portucm" contains valid data
#define DQDIO462_RDCNTSET       (1L<<5)    // =1 if "rdcntset" contains valid data
#define DQDIO462_ADCCFG0SET     (1L<<7)    // =1 if "adccfg[0]" contains valid data
#define DQDIO462_ADCCFG1SET     (1L<<8)    // =1 if "adccfg[1]" contains valid data
#define DQDIO462_ADCCFG2SET     (1L<<9)    // =1 if "adccfg[2]" contains valid data
#define DQDIO462_ADCCFG3SET     (1L<<10)   // =1 if "adccfg[3]" contains valid data
#define DQDIO462_ADCCFG4SET     (1L<<11)   // =1 if "adccfg[4]" contains valid data
#define DQDIO462_DCDCCFGSET     (1L<<12)   // =1 if "dcdccfg" contains valid data
#define DQDIO462_DCDCXSET       (1L<<13)   // =1 if "dcdcx" contains valid data
```

*cfg* disconnection mode configuration, bitmask, bits 11:0 represent channels 11:0. A one in a corresponding bit enables auto-retry mode of the circuit breaker. The auto-retry period is one second.

*portocm* bits 11:0 contain a one for every channel that should be included into the over-current status (*portocs)*. This parameter does not affect the circuit breaker.

*portucm* bits 11:0 contain a one for every channel that should be included into the under-current status (*portocs).*

*rdcnt* number of overcurrent samples read from ADC prior to disconnection (0-32). A higher number leads to longer delay, but better noise/spike immunity.

*Adccfg[5]* Normally, the ADC converters use a default configuration that sets up the ADC converter settings for operation that works well for most applications. This setting is included in case the ADC settings ever need to the changed to accommodate special applications. Please refer to LTC2486 documentation for details.

*dcdccfg* divider for the internal 66MHz clock which defines the frequency of the DC-DC converters that powers the ADC subsystem. This value and the following value (dcdcx[4]) are normally changed together as a set. In the event that these values need to be adjusted, compatible settings will be provided by UEI in order to prevent damage to the unit.

*dcdcx[4]* duty cycle and phase control for the ADC subsystem DC-DC converters.

### *4.33.4 DqAdv462SetLimit*

**Syntax:**

`int DqAdv462SetLimit(int hd, int devn, int limitid, double limitvalue)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |

| | |
|---|---|
| `int devn` | Layer inside the IOM |
| `int limitid` | Select Over or Under- range limit to set<br>0..11 - over-range limits for channels 0-11<br>16..27 - under-range limits for channels 0-11 |
| `int limitch` | Which subchannel to use for limit comparison, see below. |
| `double limitvalue` | desired limit |

**Output:**

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-462 or DIO-463 |
| `DQ_BAD_PARAMETER` | limitid is not in the range of 0..11 or 16..27 , or limitvalue is above/below allowable values |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function should be used to program the circuit breaker over- and/or under- current limit on the DIO-462 and DIO-463 layer.

*int limitch* selects which subchannel to use for comparison and circuit breaker function. Use one of the following defines:

DQ_DIO462_SUBCH_V_NO // limitvalue Unit Of Measure is volts
DQ_DIO462_SUBCH_I_AC // limitvalue UOM is amps
DQ_DIO462_SUBCH_I_DC // limitvalue UOM is amps
DQ_DIO462_SUBCH_V_NC // limitvalue UOM is volts
DQ_DIO462_SUBCH_THERM // limitvalue UOM is degreesC
DQ_DIO462_DISABLE_BREAKER // when used as over-range value, circuit breaker function is disabled

**Note:**

The current limits should be set individually for the over- and under- current conditions on every channel. Thus, up to 24 calls of DqAdv462SetLimit() may be required to completely program the layer.

## 4.34 DNA-DIO-470 layer

The DIO-470 layer uses the `DqAdv40xWrite` function to control the relays.

### 4.34.1 DqAdv470Settings

**Syntax:**

```
int DqAdv470Settings(int hd, int devn, int delay, int dyn_ctrl)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int delay` | Set limiting delay. Sets hardware to prevent software from switching relays more often than this number of milliseconds. Legal range from 5 to 65540. Default value is 7. |
| `int dyn_ctrl` | Boolean value, turns ON/OFF the dynamic control of relay coil voltage. Default value is ON |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a DIO-470 |
| `DQ_BAD_PARAMETER` | delay is not in the range of 5...65540 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function should be used to program the settings on the DIO-470 layer.

**Note:**

None.

## *4.35 Serial-500 layer (ColdFire IOM only)*

### *4.35.1 DqAdv500SetConfig*

**Syntax:**

```
int DqAdv500SetConfig(int hd, int devn, int sending, uint32
config)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int sending` | `TRUE` for the sending subsystem, `FALSE` for the receiving subsystem |
| `uint16 config` | configuration flags |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a V-500 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets the configuration options for a V-500 layer.

**Note:**

None

### *4.35.2 DqAdv500SetTxCondition*

**Syntax:**

```
int DqAdv500SetTxCondition(int hd, int devn, uint32 cmd, uint8
*value)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 cmd` | what to set |
| `uint8 *value` | value to write |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a V-500 |
| `DQ_BAD_PARAMETER` | `cmd` is not one of the `DQL_IOCTL500_SET_MSG` values, or `value` is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets the method by which the device divides the received data stream into discrete messages for transfer back to the host.

The following values are defined for the `cmd` parameter:

```
#define DQL_IOCTL500_SET_MSG_TIMER  1    // set timer interval
#define DQL_IOCTL500_SET_MSG_TERM   2    // set msg terminator (null terminated char)
#define DQL_IOCTL500_SET_MSG_LEN    3    // set msg length
```

**Note:**

None

## 4.36 DNA-SL-501 and DNA-SL-508 layers

DNA-SL-501 and DNA-SL-508 are serial layers. SL-501 layer has four serial 232/422/485 ports and SL-508 has eight ports.

### 4.36.1 DqAdv501BaseClock

**Syntax:**

```
int DqAdv501BaseClock(int hd, int devn, int chnl, int baseclock)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `int chnl` | device channel |
| `int baseclock` | base clock frequency |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-501 |
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets the base clock frequency for an individual 501 channel.

Baseclock may be set to one of two values `DQ_L501_BASE_66` or `DQ_L501_BASE_24`

**Note:**

The 24MHz base clock source for non-standard baud rates is only available in logic version 0x01020E05 or later.

### 4.36.2 DqAdv501ChangeChannelCfg

**Syntax:**

```
int DqAdv501ChangeChannelCfg (int hd, int devn, uint32 channel, int
config)
```

**Command:**

Configure channel parameters

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `uint32 channel` | device channel |
| `int config` | channel configuration (baud, parity, etc.) |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by devn does not exist or is not a SL-501, 508, or CSDB-509 card |
| `DQ_SEND_ERROR` | unable to send the command to the IOM |
| `DQ_RECV_ERROR` | unable to receive replies from the IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Waits for TX FIFO to complete and then changes parameters for selected channel

**Notes:**

Tx FIFO output can take a long time if it is full.
The cube will not reply until the output is completed
The worst case is almost 7s for 2048 characters at 300 baud.

### *4.36.3 DqAdv501ChangeChannelCfgExt*

**Syntax:**

```
int DqAdv501ChangeChannelCfgExt (int hd, int devn, uint32 channel,
uint32 config, uint32 ext_flags, uint32* extcfg)
```

**Command:**

Configure additional channel parameters

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `uint32 channel` | device channel |
| `uint32 config` | channel configuration (baud, parity, etc.) |
| `unit32 ext_flags` | extended configuration flags, reserved |
| `unit32* extcfg` | configuration data array, reserved |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-501, 508, or CSDB-509 card |
| `DQ_SEND_ERROR` | unable to send the command to the IOM |
| `DQ_RECV_ERROR` | unable to receive replies from the IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Waits for TX FIFO to complete and then changes parameters for selected channel

**Notes:**
This function extends functionality of `DqAdv501ChangeChannelCfg()` by adding optional configuration flags and parameters. If not used assign 0 and NULL to the last two parameters; the rest of the parameters are the same as in `DqAdv501ChangeChannelCfg()`.

### *4.36.4 DqAdv501ChangeChannelParity*

**Syntax:**

```
int DqAdv501ChangeChannelParity (int hd, int devn, uint32
channel, int config)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `uint32 channel` | device channel |
| `int config` | parity setting |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-501 |
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**
Waits for TX FIFO to complete and then changes parity for selected channel. (changes parity only; the full config word needs to be supplied)

**Note:**

Tx FIFO output can take a long time if it is full.
The cube will not reply until the output is completed
The worst case is almost 7s for 2048 characters at 300 baud.

### *4.36.5 DqAdv501SetChannelCfg*

**Syntax:**

```
int DqAdv501SetChannelCfg(int hd, int devn, int chnl, uint32
config)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `int chnl` | device channel |
| `uint32 config` | configuration flags |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-501 |
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets the configuration properties for an individual 501 channel.

**Note:**

Use the macro DQCFG_501 to set the configuration flags.

For example, the following configure the serial port in RS-232 mode, 57600 bps, 8 data bits, no parity and 1 stop bit.

```
config = DQCFG_501(DQ_SL501_OPER_NORM, DQ_SL501_MODE_232,
DQ_SL501_BAUD_57600, DQ_SL501_WIDTH_8, DQ_SL501_STOP_1, DQ_SL501_PARITY_NONE);
```

Parity and Framing errors are not reported by default. To report them add the following flags:

```
config |= DQ_SL501_ERROR << DQ_SL501_ERROR_SH;
```

### *4.36.6 DqAdv501SetChannelCfgExt*

**Syntax:**

```
int DqAdv501SetChannelCfgExt(int hd, int devn, int chnl, uint32
config, uint32 ext_flags, uint32* extcfg)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `int chnl` | device channel |
| `uint32 config` | configuration flags |
| `uint32 ext_flags` | extended configuration flags |

| | |
|---|---|
| `uint32* extcfg` | extended configuration (not implemented, set to 0) |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an SL-501 or an SL-508 |
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets the configuration properties and extended configuration flags for an individual 501 or 508 channel.

**Note:**

Use the macro DQCFG_501 to set the configuration flags.

For example, the following configure the serial port in RS-232 mode, 57600 bps, 8 data bits, no parity and 1 stop bit.

```
      config = DQCFG_501(DQ_SL501_OPER_NORM, DQ_SL501_MODE_232,
DQ_SL501_BAUD_57600, DQ_SL501_WIDTH_8, DQ_SL501_STOP_1, DQ_SL501_PARITY_NONE);
```

Parity and Framing errors are not reported by default. To report them add the following flags:

```
      config |= DQ_SL501_ERROR << DQ_SL501_ERROR_SH;
```

The echo of the transmitted character in half duplex mode may be suppressed using the ext_flags parameter. This feature requires a 501 or 508 layer with logic version 02.10.5A or newer.

```
      ext_flags = DQ_SL501_SUPRESS_HD_ECHO;
```

### *4.36.7 DqAdv501SetBaud*

**Syntax:**

```
int DqAdv501SetBaud(int hd, int devn, int chnl, uint32 baud,
uint32 *realbaud)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `int chnl` | device Channel |
| `uint32 baud` | channel baud rate |

| | |
|---|---|
| `uint32* realbaud` | pointer for actual baud rate of device |

**Output:**

| | |
|---|---|
| `uint32* realbaud` | actual baud rate of device after prescaler calculation |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an SL-501 or SL-508 |
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets a baud rate other than that defined by DQ_SL501_BAUD_XXX. To do this replace the DQ_SL501_BAUD_XXX constant with the DQ_SL501_BAUD_CUST constant in the `config` config parameter that is passed to DqAdv501SetChannelCfg(). Range of `baud` should be between 300bps – 1.5Mbps.

This function may also be used to set the baud using the on-layer PLL. This applies to SL-501and SL-501-804 layers with logic revision 0x0102006 or higher. To use the PLL, first call the DqAdv501SetChannelCfg() function with (1L<< DQ_SL501_BAUD_PLL_SH) added to the `config` value. The upper baud limit using the PLL is 2Mbps for the SL-501 and 4Mbps for the SL-501-804

**Note:**

The requested baud and actual baud rates will be different depending on speed. The user should compare *baud* with *realbaud* to ensure that the rate is within application-specific error limits.

### *4.36.8 DqAdv501SetTimeout*

**Syntax:**

```
int DqAdv501SetTimeout(int hd, int devn, int chnl, uint16
timeout)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Device Channel |
| `uint16 timeout` | Receive FIFO timeout in milliseconds |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-501 |
| `DQ_BAD_PARAMETER` | invalid parameter |

| | |
|---|---|
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets the length of time the receive FIFO should wait before flushing the buffer and sending data to the host. Time is in milliseconds with a max of 65535.

**Note:**

None

### 4.36.9 DqAdv501SetTermString

**Syntax:**

```
int DqAdv501SetTermString(int hd, int devn, int chnl, uint16
len, uint8 *buf)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `int chnl` | device channel |
| `uint16 len` | length of termination string |
| `uint8* buf` | pointer to termination string |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an SL-501 |
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets the termination string for received messages. The Cube will send the received serial data to the host PC or UEIPAC's CPU after the termination string has been found.

**Note:**

The maximum termination string length is limited by packet size and should be less than 470 bytes.

### 4.36.10 DqAdv501SetWatermark

**Syntax:**

```
int DqAdv501SetWatermark(int hd, int devn, int chnl, uint32 dir,
uint16 len)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `int chnl` | device channel |
| `uint32 dir` | watermark FIFO direction |
| `Uint16 len` | watermark position |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-501 |
| `DQ_BAD_PARAMETER` | invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function permits user configuration of internal TX and RX FIFO watermarks to better control dataflow from the IOM.

**Note:**

Parameter dir is DQL_IOCTL501_SETTXWM or DQL_IOCTL501_SETRXWM and defaults to 1 and 0 as default. Watermark range is between 0 and `2047`.

### *4.36.11 DqAdv501SetTermLength*

**Syntax:**

```
int DqAdv501SetTermLength(int hd, int devn, int chnl, uint16
len)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `int chnl` | device channel |
| `uint16 len` | termination length for received messages |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-501 |

| | |
|---|---|
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets the length of the received message. The Cube will send the received serial data to the host PC or UEIPAC's CPU upon receipt of `len` characters.

**Note:**

The maximum termination string length is limited by packet size and should be less than 470 bytes.

### *4.36.12 DqAdv501SetCharDelay*

**Syntax:**

```
int DqAdv501SetCharDelay(int hd, int devn, int chnl, uint32
delay_src, double delay_us)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `int chnl` | device channel |
| `uint32 delay_src` | source of the delay clock |
| `double delay_us` | delay between characters in microseconds |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-501 |
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets delay between Tx characters. Select <delay_src> from:

```
DQ_SL501_DELAYMODE_DISABLE     0    // disabled (Tx done)
DQ_SL501_DELAYMODE_INTERNAL    1    // internal per channel clock
DQ_SL501_DELAYMODE_TMR01       2    // TMR1 for char and TMR0 for frame
DQ_SL501_DELAYMODE_SYNC02      3    // SYNC2 for char and SYNC0
```

**Notes:**

- Make sure that if the delay is enabled, it is longer than the duration of the transmitted character, including start and stop bits.
- When character delay is enabled, you must use `DqAdv501WriteTxFIFO()` to send data.
- Implemented in logic version 02.0E.05 or later

### *4.36.13 DqAdv501SetFrameDelay*

**Syntax:**

```
int DqAdv501SetFrameDelay(int hd, int devn, int chnl, uint32
delay_mode, uint32 length, uint32 delay_src, double delay_us, double
repeat_us)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `int chnl` | device channel |
| `uint32 delay_mode` | mode of frame delays |
| `uitn32 length` | string length if DQ_SL501_FRAMEDELAY_FIXEDLEN selected |
| `uint32 delay_src` | source of the delay clock |
| `double delay_us` | delay between frames in microseconds |
| `double repeat_us` | delay before the last frame gets repeated and also frame rate in `DQ_SL501_FRAMEDELAY_REPEAT` mode |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-501 |
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets delay between Tx frames.

Select <delay_mode> from:

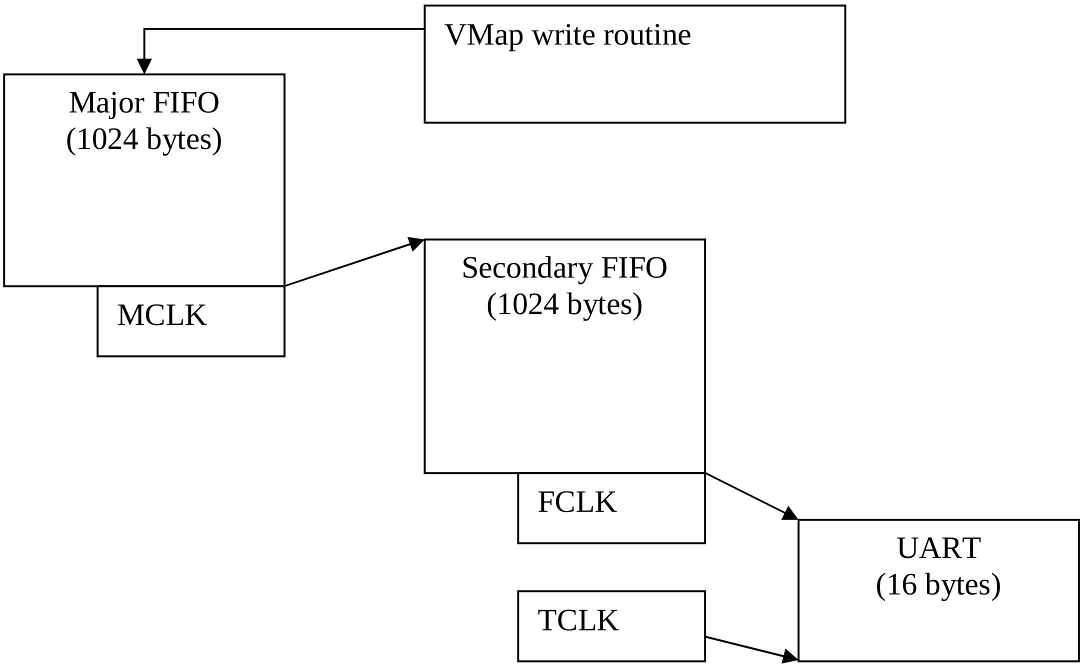
| | | |
|---|---|---|
| `DQ_SL501_FRAMEDELAY_DISABLE` | `0` | // disabled |
| `DQ_SL501_FRAMEDELAY_FIXEDLEN` | `1` | // fixed length |
| `DQ_SL501_FRAMEDELAY_VMAP_LEN` | `2` | // frame size is determined by the amount of VMap data |
| `DQ_SL501_FRAMEDELAY_ZERO_CHAR` | `3` | // frame ends upon zero character (ASCIIZ string) |
| `DQ_SL501_FRAMEDELAY_REPEAT` | `8` | // "repeat" mode - repeat last placed frame if not updated on time |

Select <delay_src> from:

```
DQ_SL501_DELAYMODE_DISABLE     0    // disabled (Tx done)
DQ_SL501_DELAYMODE_INTERNAL    1    // internal per channel clock
```

```
DQ_SL501_DELAYMODE_TMR01       2    // TMR1 for char and TMR0 for frame
DQ_SL501_DELAYMODE_SYNC02      3    // SYNC2 for char and SYNC0 for frame
```

The SL-508 layer has the Major FIFO unit on top of the Secondary FIFO unit. The Major FIFO is intended for implementation of the auto-repeat feature that keeps on sending serial messages to the avionics devices with hard-realtime deadlines. The internal structure of the output channel is as follows:



The Auto-repeat mode requires data to be put into Major FIFO instead writing directly into the Secondary FIFO. In such a case, the Major-Secondary FIFO system acts as a double-buffered scheme.

In frame repeat mode, the device works as follows:

1. VMap routine tries to lock the Major FIFO. The Major FIFO can be locked at any time except when it is transferring data to the Secondary FIFO at the rate of 150ns per byte
2. Once the Major FIFO is locked, its content is reset and the firmware writes data to the FIFO. The lock-write-unlock procedure prevents a situation in which the Major FIFO gets partially overwritten and packet integrity is lost. At the same time, it prevents the situation that occurs when data is accumulated in the Major FIFO. If, for some reason, (for example, if the VMap rate is faster than the selected frame rate) the data written into the Major FIFO in a previous pass hasn't had a chance to be transferred into the Secondary FIFO, it will be overwritten and lost.
3. Data should be written in the `DQ_SL501_FRAMEDELAY_VMAP_LEN` mode and firmware adds a "frame" bit (0x100) to the first entry in the VMap packet.
4. When firmware completes a write into the Major FIFO, it releases it.
5. When MCLK data is transferred into Secondary FIFO.
6. If the firmware doesn't lock the Major FIFO when the next MCLK arrives, the Major FIFO data is transferred into the Secondary FIFO again. Thus, if the realtime program stops sending data to the layer, it will continue to output that last received frame, using MCLK as a timebase.

7. If the frame bit is set and the secondary FIFO waits for FCLK to output data.
8. Characters are put into the UART FIFO using TCLK to introduce inter-character delay.

To initiate repeat mode, one should call `DqAdv501SetFrameDelay()` as follows (please note that an additional parameter <`delay_us_major_fifo`> was added to this function since the Stage I release):

```
DqAdv501SetFrameDelay(hd0, DEVN, CH0,
      DQ_SL501_FRAMEDELAY_VMAP_LEN|DQ_SL501_FRAMEDELAY_REPEAT, 0,
      DQ_SL501_DELAYMODE_INTERNAL,
      delay_us_secondary_fifo,
      delay_us_major_fifo);
```

In other words, one should specify <`delay_us_major_fifo`> as an inter-frame delay and <`delay_us_secondary_fifo`> as a delay if you would like to put multiple sub-frames into the secondary FIFO and specify delay between them. Due to the fact that `DQ_SL501_FRAMEDELAY_VMAP_LEN` defines a frame as a single VMap write, `DQ_SL501_FRAMEDELAY_ZERO_CHAR` or `DQ_SL501_FRAMEDELAY_FIXEDLEN` constants should be used.

**Notes:**

- Make sure that if the delay is enabled, it is longer than duration of the transmitted character including start and stop bits.
- When frame delay is enabled, you must use `DqAdv501WriteTxFIFO()` to send data.
- Implemented in logic version 02.0F.01 or later

### *4.36.14 DqAdv501SetParity9*

**Syntax:**

`int DqAdv501SetParity9(int hd, int devn, int chnl, uint32 parity_mode)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `int chnl` | device channel |
| `uint32 parity_mode` | parity mode |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-501 |
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |

| | |
|---|---|
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets/clears the use of parity as 9th bit.

**Note:**

Implemented in logic version 0x01021006 or later.

Use the following defined constants as the `parity_mode` value:

```
DQ_SL501_PARITY9_DISABLE    0    // disabled
DQ_SL501_PARITY9_ENABLED    4    // enabled
```

### 4.36.15 DqAdv501GetStatus

**Syntax:**

```
int DqAdv501GetStatus(int hd, int devn, uint32* errors,
uint32* count)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `uint32* errors` | Array of Parity Error and Frame Error counters for each channel |
| `uint32* count` | Number of uint32 words in the `errors` array (8 for 501 and 16 for 508) |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an SL-501 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Requests the error counts for parity and frame errors for each channel.

**Notes:**

Error counters are cleared in the device once this function is called.

### *4.36.16 DqAdv501PauseAndResume*

**Syntax:**

```
int DqAdv501 PauseAndResume (int hd, int devn, uint32
channel_mask, int todo)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 channel_mask` | channel to change configuration (mask) |
| `int todo` | operation to perform |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an SL-501 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

If <todo> == DQL_IOCTL501_CHANGE_PAUSE pause receive
If <todo> == DQL_IOCTL501_CHANGE_RESUME resume receive
If 501/508 device is in messaging mode this function call also finalizes whatever ongoing message was accumulating the data

**Notes:**

None

### *4.36.17 DqAdv501Enable*

**Syntax:**

```
int DqAdv501Enable(int hd, int devn, int enable)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `int enable` | TRUE (1) = enable, FALSE (0) = disable |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an SL-501 or an SL-508 |
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function starts/stops a 501 or 508 device.

**Note:**

None

### 4.36.18 DqAdv501ClearFifo

**Syntax:**

```
     int DAQLIB DqAdv501ClearFifo(int hd, int devn, uint32 ch_mask,
int action)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `uint32 ch_mask` | mask of channels to clear FIFO for |
| `int action` | one or a combination of DQL_IOCTL501_CHANGE_FINCLEAR and DQL_IOCTL501_CHANGE_FOUTCLEAR |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-501/508 |
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function clears input and/or output FIFOs on 501/508 layers

**Note:**

Function can be applied to more than one channel on the layer by OR-ing channel mask with (1L<<channel_number). A user doesn't have to call this function during normal operations because DqAdv501Enable() does cleaning for you. This function is for operation mode only. For example, when SL-501 layer operates in VMap mode and input FIFO needs to be cleared of received data before resynchronization with external device

### *4.36.19 DqAdv501RecvMessage*

**Syntax:**

```
int DqAdv501RecvMessage (int hd, int devn, int chan, uint8
*data, uint16 *size, int *success, uint8* errorcode, int *avail)
```

**Command:**

Get message received on a 501/508 channel.

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `int chan` | channel number to get message from |
| `uint8* data` | data buffer to store received message |
| `uint16* size` | number of bytes in data buffer |

**Output:**

| | |
|---|---|
| `uint8* data` | received message |
| `uint16* size` | number of bytes written to buffer, returns 0 if no message |
| `int* success` | TRUE if message was read successfully |
| `uint8* errorcode` | if success=FALSE returns 501 specific error code, timeout = 0x40 + `chan` |
| `int* avail` | TRUE if more messages are available |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-501/508 |
| `DQ_BAD_PARAMETER` | `chan`, `id`, `data`, or `size` is NULL |
| `DQ_NOT_ENOUGH_ROOM` | `data` buffer is not large enough to hold the received message |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function is only for use with point-by-point/immediate DAQ mode. Messages received by a 501/508 layer are stored in the `data` message buffer. The buffer must have a large enough `size` to hold the maximum number of characters you expect in the message. The message length is determined by one of the following three conditions:

1. **Termination String** – If defined, the message length is the number of characters between two termination strings. Set the termination string using `DqAdv501SetTerminationString()`.
2. **Termination Length –** If the termination string is not defined or not found, the message length is equal to the termination length value set by `DqAdv501SetTerminationLength()`.
3. **Timeout –** If neither termination string nor termination length conditions have been met, the firmware will return all received characters once the timeout expires. The timeout duration is set using `DqAdv501SetTimeout()`.

**Notes:**
Maximum message size is limited to 470 bytes in a 576-byte Ethernet frame and 1412 bytes in a 1518-byte frame.

### *4.36.20 DqAdv501SendBreak*

**Syntax:**

```
int DqAdv501SendBreak(int hd, int devn, int chnl, uint32
duration);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `int chnl` | device Channel |
| `uint32 duration` | duration in milliseconds |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an SL-501 or an SL-508 |
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**
This function is for immediate mode usage only. It sets the break condition for the specified number of milliseconds.

**Notes:**
This break output is not dependent upon having break (i.e. `DQ_SL501_BREAK_SH`) enabled with `DqAdv501SetChannelCfg()`. Any channel can output a break regardless of configuration.

### *4.36.21 DqAdv501SendMessage*

**Syntax:**

```
int DqAdv501SendMessage(int hd, int devn, int chan, uint8 *data,
uint16 size, uint16 *written, int *success, uint8 *errorcode);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `int chan` | device Channel |
| `uint8* data` | buffer of message to send |
| `uint16 size` | length of message |

**Output:**

| | |
|---|---|
| `uint16* written` | number of bytes written |
| `int *success` | TRUE if message was written |
| `uint8* errorcode` | if success=FALSE returns 501 specific error code |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an SL-501 or an SL-508 |
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function is for immediate mode usage only. It writes a block of data to a serial port. If (size == written), all data fits into the outbound FIFO.

**Note:**

The maximum message size is limited to 470 bytes in a 576-byte Ethernet frame.
For 1518-byte frame, the maximum size depends on the board under use:
SL-501 – 1400
SL-508 – 1024

### *4.36.22 DqAdv501SendMessageParity9*

**Syntax:**

```
int DqAdv501SendMessageParity9(int hd, int devn, int chnl,
uint16 *data, uint16 size, uint16 *written, int *success,
uint8 *errorcode);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `int chnl` | device Channel |
| `uint16* data` | buffer of message to send with parity (bit 9) encoded |
| `uint16 size` | length of message |

**Output:**

| | |
|---|---|
| `uint16* written` | number of bytes written |
| `int *success` | TRUE if message was written |
| `uint8* errorcode` | if success=FALSE returns 501 specific error code |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an SL-501 or an SL-508 |
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function is for immediate mode usage only. It writes a block of data to a serial port. If (`size == written`), all data fits into the outbound FIFO.

**Note:**

The version of `DqAdv501SendMessage()` with parity included with the data.

### *4.36.23 DqAdv501ReadRecvFIFO / DqAdv501ReadRxFIFO*

**Syntax:**

```
int DqAdv501ReadRecvFIFO(int hd, int devn, int chnl, int
requested, uint8* data, int* returned, int* remained)
```

**Command:**

Read data from the receive FIFO.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Channel number message was received from |
| `int requested` | Number of characters to read from receive FIFO |

**Output:**

| | |
|---|---|
| `uint8* data` | Received characters |
| `int* returned` | Number of bytes returned (each character is stored as a byte) |

| | |
|---|---|
| `int* remained` | Bytes of unread data remaining in receive FIFO |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not a SL-501/508/509 |
| `DQ_BAD_PARAMETER` | `chnl`, `data`, or `written` is invalid |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function reads characters from the RX FIFO on a SL-501/508/509 layer. `Returned` may be less than `requested` if the RX FIFO is short on data or if the termination string has been found.

**Notes:**

- Only for use with point-by-point/immediate DAQ mode
- The termination string can span across multiple function calls. If one call returns the beginning of the termination string, the next call will watch for the remainder of the string.

### *4.36.24 DqAdv501WriteTxFIFO*

**Syntax:**

```
int DqAdv501WriteTxFIFO(int hd, int devn, int chnl, uint32
requested, uint8* data, int* written, int* remained_free)
```

**Command:**

Write data to the transmit FIFO.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Channel number to write message to |
| `uint32 requested` | Number of characters to write to TX FIFO |
| `uint8* data` | Buffer of message to send |

**Output:**

| | |
|---|---|
| `int* written` | Number of bytes written (each character stored as a byte) |
| `int* remained` | Number of bytes remaining free in the TX FIFO |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not a SL-501/508/509 |

| | |
|---|---|
| `DQ_BAD_PARAMETER` | `chnl`, `data`, or `written` is invalid |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**
This function is only for use with Point-by-Point DAQ mode. It writes a block of data to the TX FIFO on a SL-501/508/509 layer. If `requested == written`, all data fit into the FIFO.

**Notes:**

### *4.36.25 DqAdv501FlowControl*

**Syntax:**

```
     int DqAdv501FlowControl (int hd, int devn, int chnl, uint32
cts_cfg, uint32 rts_cfg, uint32* cts_state)
```

**Command:**
IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | The channel that gets its flow control set |
| `uint32 cts_cfg` | The CTS configuration |
| `uint32 rts_cfg` | The RTS configuration |

**Output:**

| | |
|---|---|
| `uint32* cts_state` | The CTS state of all the channels |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-501/8 |
| `DQ_BAD_PARAMETER` | `cts_cfg` or `rts_cfg` is not one of the listed defined constants |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**
This function sets the RS-232 flow control on the specified channel and returns the CTS state for all channels on the layer.

Inputs:

`cts_cfg` Configures the operation of the CTS pin. The following constants are defined:

| | |
|---|---|
| #define DQL_IOCTL501_CTS_NO_CHANGE | (1) |
| #define DQL_IOCTL501_CTS_IGNORE | (2) |
| #define DQL_IOCTL501_CTS_AUTOFLOW | (3) |

DQL_IOCTL501_CTS_NO_CHANGE will keep the current CTS configuration.

When configured with DQL_IOCTL501_CTS_IGNORE , the transmitter will always send the data regardless of the state of the CTS input.

With DQL_IOCTL501_CTS_AUTOFLOW the transmitter will check the state of the state of the CTS input before sending data. A high state on the CTS pin will enable data flow and a low state will stop data flow.

`rts_cfg` Configures the operation of the RTS pin. The following constants are defined:

| | |
|---|---|
| #define DQL_IOCTL501_RTS_NO_CHANGE | (1) |
| #define DQL_IOCTL501_RTS_PIN_LOW | (2) |
| #define DQL_IOCTL501_RTS_PIN_HIGH | (3) |
| #define DQL_IOCTL501_RTS_AUTOFLOW_NORMAL_TIMING | (4) |
| #define DQL_IOCTL501_RTS_AUTOFLOW_GATED_TIMING | (5) |

DQL_IOCTL501_RTS_NO_CHANGE will keep the current RTS configuration.

DQL_IOCTL501_RTS_PIN_LOW will set the RTS pin to the low state.

DQL_IOCTL501_RTS_PIN_HIGH will set the RTS pin to the high state.

DQL_IOCTL501_RTS_AUTOFLOW_NORMAL_TIMING and DQL_IOCTL501_RTS_AUTOFLOW_GATED_TIMING will normally keep the RTS pin high and will set the RTS pin low when receive FIFO contains the number of characters set by the `DqAdv501SetWatermark()` function. When the receive FIFO is emptied the RTS pin will be returned to the high state. The NORMAL and GATED timing refers to when the RTS pin with change state. With normal timing the RTS pin is deasserted (set low) after the first data bit of the character that reaches the watermark level is present on the serial input line. With gated timing the RTS pin is deasserted after the middle of the stop bit of character that reaches the watermark level. Note that the watermark level is one less than the number of characters received. For instance, if you wanted to halt the data flow after 8 characters were received you would call the `DqAdv501SetWatermark()` function with the `len` parameter set to 7.

Output:

When the value of `uint32* cts_state` is not NULL, it will return a bit field value that indicates the CTS pin state of all channels on the layer. Bit 0 of the returned value indicates the state of CTS pin on channel 0, bit 1 for channel 1 ,etc.
When the layer is an SL-501 bits 0..3 will be meaningful and an SL-508 uses bits 0..7 .
A ‘1’ in the bit position shows that the pin is in the high state and indicates that the transmitter is enabled when CTS_AUTOFLOW is selected.

**Note:** This function is implemented only on SL-501 and SL-508 layers with logic revision 0x0102049 or higher.

### 4.36.26 DqAdv501ConfigEvents

**Syntax:**

```
int DAQLIB DqAdv501ConfigEvents (int handle, int devn, int channel, int flags,
event501_t event, uint32* param)
```

**Command:**

Specify what asynchronous notification events user wants to receive from the layer

| **Input** | |
|---|---|
| int handle | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel number |
| int flags | Behavior modification flags |
| event501_t event | Event type |
| uint32* param | Additional parameters depends on event type |
| **Output** | |
| None | |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an SL-501 or SL-508 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**

This function sets up events and their parameters which program firmware to send asynchronous event notification packet upon reaching certain conditions.

event501_t event is defined as:

```
typedef enum {
    EV501_CLEAR = 0x1000,           // clear all events

    EV501_TO = 0x101,               // no activity on the bus over certain pre-programmed
```

```
                                  // time (+ all accumulated data)
    EV501_IN_FIFO,                // input FIFO above watermark (data included) (cannot be
                                  // used together with EV501_RX)
    EV501_RX,                     // data received with check for length and terminator
                                  // (+ data)
    EV501_OUT_FIFO,               // output FIFO below watermark (cannot be used together
                                  // with EV501_TX)
    EV501_TX,                     // data transmission completed
    EV501_BUS_ERROR,              // bus or protocol errors
    EV501_RX_OVERFLOW,            // RX FIFO overflow - cleared
    EV501_RX_UNXP,                // unexpected RX data (most likely receiveing error)
    EV501_TX_UNXP                 // unexpected TX data (most likely transmitting error)
} event501_t;
```

<flags> can contain the follwing bits:

```
#define DQEVENT_NOREPLY              (1L<<31)    // do not reply to this event
#define DQEVENT_ONCE                 (1L<<30)    // fire event only once
#define DQEVENT_NOSTORE              (1L<<29)    // do not store UDP port to
generate events
```

If `DQEVENT_NOREPLY` bit is set a function call `DqAdv501ConfigEvents()` doesn't wait for the answer from the firmware on the success of this function call. This comes handy in the situation when user wants to change event parameter (e.g. watermark level) on the fly to avoid increase in the network traffic.

If `DQEVENT_ONCE` is send the firmare sends event packet only once and then event becomes disabled until a next call to `DqAdv501ConfigEvents()` re-enables it.

`DQEVENT_NOSTORE` is required to tell firmware not to change the reply UDP port. For example, user can create multiple UDP ports using `DqAddIOMPort()` to listen to incoming events on the different ports. Thus, when the function called from the handle other than the handle event is expected upon this bit need to be set. Do not set this bit for the first time it is used.

To clear all events use the following call:

```
ret = DqAdv501ConfigEvents(a_handle, devn, channel, 0, EV501_CLEAR, NULL);
```

It clears all events that has previously been set along with runtime variables and accumulated events.

`EV501_IN_FIFO` sends an event packet when the amount of data in the secondary input FIFO exceeds the watermark level programmed. To set up processing of this event call:
`DqAdv501ConfigEvents(handle, devn, channel, 0, EV501_IN_FIFO, &wm_level);` where wm_level is the required watermark level, [1.. DQ_L508_FIFOSZ-1].

SL-501 has a FIFO bigger than the amount of data can be stored in one packet. If the amount of data received in the FIFO exceeds watermark but cannot be sent in one packet multiple packets will be sent back-to-back until the level in the FIFO drops below watermark. The maximum watermark level should be determined with the consideration of having enough spare capacity to store more incoming characters while interrupt is processed depends on the baud rate. A good estimate is that watermark should be set below the full capacity of the FIFO by 80us worth of data. For 1Mbaud this would be equal to 8 to 10 characters.

The watermark level can be programmed using `DqAdv501SetWatermark(int hd, int devn, int chnl, uint32 dir, uint16 len).` Another way to change watermark level is to call `DqAdv501ConfigEvents(handle, devn, channel, 0, EV501_IN_FIFO, &wm_level);` When this function is called in the thread to change watermark level on the fly `(DQEVENT_NOSTORE)` flags needs to be used to avoid changing UDP port. `DQEVENT_ONCE and DQEVENT_NORELPY` can be added if single-shot event is required and no confirmation reply from the firmware is desired (decreases traffic but decreases reliability as well thus recommended to be used on the busy applications only).

The `EV501_IN_FIFO` event will send minimum of the amount the data requested in `wm_level`, even if the FIFO contains more data. Packet is processed when an interrupt on watermark level is triggered by the incoming Rx data.

When this event happens the firmware sends back a packet of the following structure:

```
/* DQCMD_EVENT structure */
typedef struct {
    uint8   dev;                    // device
    uint8   ss;                     // subsystem
    uint16  size;                   // data size
    uint32  event;                  // event type or command and additional flags
    uint8   data[];                 // data
} DQEVENT, *pDQEVENT;
```

<size> field contains sizeof(EV501_ID)+length.
<event> contains the type of the event, e.g. in this case EV501_IN_FIFO.
<data[]> if DQEVENT contains EV501_ID flexible structure:

```
// Event data for 501 layer
typedef struct {
    uint32 chan;                    // channel information
    uint32 evtype;                  // type of the event
    uint32 tstamp;                  // timestamp of event
    uint32 size;                    // size of the following data in bytes
    uint32 avail;                   // number of bytes available
    uint8 data[];                   // data to follow (one character takes one
byte)
} EV501_ID, *pEV501_ID;
```

Please notice that while DQEVENT is properly handled by `DqCmdReceiveEvent()` in terms of converting data fields from big endian (network data representation) to little endian if used with Intel-based host computer while EV501_ID needs to be converted by the user with the following code:

```
// shuffle bytes big endian->little endian in place
void ntoh_pEv501(pEV501_ID pEv501) {
    uint32 i;

    pEv501->chan = ntohl(pEv501->chan);
    pEv501->evtype = ntohl(pEv501->evtype);
    pEv501->size = ntohl(pEv501->size);
    pEv501->tstamp = ntohl(pEv501->tstamp);
    for (i = 0; i < pEv501->size/sizeof(uint8); i++)
        pEv501->data[i] = (pEv501->data[i]);
    return;
```

}
The reason for that is that `DqCmdReceiveEvent()` returns all different types of layer specific events and as implemented doesn’t do this conversion.

EV501_ID contains:
pEv501->evtype = isr; // actual bits in the board ISR register
pEv501->avail = in_fifo - length; // amount of data still available in the FIFO
pEv501->size = length; // amount of received data

Timeout-based EV501_TO event is set to send an event packet with data when there is no activity on the receiver in excess of the programmed timeout period.
Timeout is programmed using `DqAdv501SetTimeout(int hd, int devn, int chnl, uint16 timeout)`. Timeout is programmed as a number of milliseconds from 1 to 65535 (i.e. 16 bit). If timeout is programmed as zero timeout function is disabled.

Timeout resolution can be changed from a millisecond down to microsecond using DqAdv501SetChannelCfgExt() call instead of DqAdv501SetChannelCfg() and setting extended configuration flags:

```
// extended configuration bits, 'ext_flags' parameter for DqAdv501SetChannelCfgExt()
#define DQ_SL501_SUPRESS_HD_ECHO    (1L<<1)     // suppress echo in half-duplex mode,
requires logic 02.10.5A or newer
#define DQ_SL501_TIMEOUT_DIV1       (1L<<6)     // timeout divider clock selector - valid
when CCFG_TOE=1
#define DQ_SL501_TIMEOUT_DIV0       (1L<<5)     // 00 = 1mS, 01 = 100us, 10 = 10us, 11 = 1us
(available in logics >= 2.10.22)
#define DQ_SL501_USE_8x_CLK         (1L<<7)     // use 8x baud clock mode instead of 16x UART
clock (501 only)
```

Since it is possible to set up timeout to overload firmware there is a mechanism in ISR routine that disables any offending interrupts and sets a flag STS_FW_OVERLOAD in firmware status filed STS_FW.

The maximum amount of data to be sent upon timeout event is programmed when timeout event is selected: `DqAdv501ConfigEvents(handle, devn, channel, 0, EV501_TO, &msg_sz_in);` <msg_sz_in> specifies this parameter.

Both `EV501_IN_FIFO` and `EV501_TO` can be used in pair when the message length is not known ahead of time. Select parameter for `EV501_IN_FIFO` for the minimum expected length of the message and set timeout long enough to receive the remainder of the message of the maximum length. Upon receiving `EV501_IN_FIFO` arm `EV501_TO` event with `DQEVENT_ONCE` flag. When timeout expires the event packet will be sent only once with the minimum of the data available in the FIFO and the amount of data specified as a parameter in <msg_sz_in>.

`EV501_RX` event allows receiving a string when defined a termination length or a terminator. Please notice that this function puts higher load onto the firmware because each character is evaluated upon receiving. It is incompatible with `EV501_IN_FIFO` since it requires firmware to read data into a secondary queue and if the `EV501_IN_FIFO` event is set FIFO will be emptied and sent to the host before code for `EV501_RX` takes control.

`EV501_RX` packet contains the following information:

```
pEv->event = EV501_RX;
pEv501->evtype = isr;
pEv501->avail = total_len - termlen;
pEv501->size = termlen;    // length of the string teminated by the programmed character
```

Synonimous `EV501_TX` and `EV501_OUT_FIFO` are the opposite to `EV501_IN_FIFO`. Firmware sends event when the level in the Tx secondary FIFO falls below selected Tx watermark. When the level in the Tx FIFO falls to zero the interrupt is automatically disabled. It makes sense only to use this event with `DQEVENT_ONCE` and enable after storing more data to Tx FIFO to avoid continuous event generation.

The data in the

```
pEv501->evtype = isr;
pEv501->size += 0;    // adjust packet length
pEv->size += 0;
pEv501->avail = in_fifo;  // current level in TX FIFO
```

Special event packets:

This event is automatically sent when `EV501_IN_FIFO` is enabled and FIFO overflows.
Upon this situation all data is purged from the FIFO and the following data is sent in the event packet:

```
pEv->event = EV501_RX_OVERFLOW;
pEv501->evtype = isr;
pEv501->avail = in_fifo; // amount of data accumulated in RX FIFO
pEv501->size = 0;
```

EV501_BUS_ERROR , EV501_RX_UNXP and EV501_TX_UNXP events are generated when Rx or Tx receives unexpected data (error condition). They should not be received in normal operations.
The data in the packet is:

```
pEv501->evtype = isr;
pEv501->avail = in_fifo;    // amount of the data in RX or TX FIFO
pEv501->size = 0;
```

## 4.37 DNA-CAN-503 layer

### 4.37.1 DqAdv503Enable

**Syntax:**

```
int DqAdv503Enable(int hd, int devn, int enable)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int enable` | TRUE to start the CAN-503, FALSE to stop |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CAN-503 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Starts/Stops all configured channels on the CAN-503.

**Note:**

None

### 4.37.2 DqAdv503RecvMessage

**Syntax:**

```
int DqAdv503RecvMessage(int hd, int devn, int *chan, uint32 *id,
uint8 *data, uint16 *size, int *success, uint8 *error_code, int
*avail)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `int *chan` | channel number message was received on |
| `uint32 *id` | id value of message. Will return 0 if there was no message |
| `uint8 *data` | message data buffer |
| `uint16 *size` | number of bytes written to buffer. Will return 0 if there was no message |
| `int *success` | returns 0 if the CAN port or bus is in error state |
| `uint8 *error_code` | returns the error code for this CAN bus |
| `Int *avail` | returns the number of messages available in the receive FIFO |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CAN-503 |

| | |
|---|---|
| `DQ_BAD_PARAMETER` | `chan, id, data,` or `size` is NULL |
| `DQ_NOT_ENOUGH_ROOM` | the size indicated by the `size` parameter is not big enough to hold the received message |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function gets a message received by a 503 layer on a CAN network.

**Note:**

When the bus is in error state, error_code contains one of the following values:

- DQ_CAN503_ERR_WARN: warning, bus errors were detected
- DQ_CAN503_ERR_AE: Arbitration error, two nodes tried to send at the same time.
- DQ_CAN503_ERR_BE: The number of bus errors is greater than 128, bus is now in error state.
- DQ_CAN503_ERR_OR: Receive FIFO overflowed
- DQ_CAN503_ERR_TO: Timeout, no message was received.

### *4.37.3 DqAdv503SendMessage*

**Syntax:**

```
int DqAdv503SendMessage(int hd, int devn, int chan, uint32 id,
uint8 *data, uint16 size, int *success, uint8 *error_code)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chan` | channel number |
| `uint32 id` | id value of message. Lower 11 or 29 bits are used depending on standard or extended packet |
| `uint8 *data` | message data |
| `uint16 size` | number of bytes in `data` parameter |
| `int *success` | returns 0 if the CAN port or bus is in error state |
| `uint8 *error_code` | returns the error code for this CAN bus |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CAN-503 |
| `DQ_BAD_PARAMETER` | `chan` is an invalid channel number, `data` is NULL, or `size` is greater than 8, which is the maximum CAN message data size |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |

| | |
|---|---|
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sends a message from a 503 layer on a CAN network.

**Note:**

When the bus is in error state, error_code contains one of the following values:

- DQ_CAN503_ERR_WARN: warning, bus errors were detected
- DQ_CAN503_ERR_AE: Arbitration error, the message to send collided with a message sent by another node.
- DQ_CAN503_ERR_BE: The number of bus errors is greater than 128, bus is now in error state.
- DQ_CAN503_ERR_OR: Transmit FIFO overflowed.

### *4.37.4 DqAdv503WriteTxFIFO*

**Syntax:**

```
int DqAdv503WriteTxFIFO(int hd, int devn, int chnl, uint32
requested, uint8* data, int* written, int* remained_free)
```

**Command:**

Write data to the transmit FIFO.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Channel number to write message to |
| `uint32 requested` | Number of characters to write to TX FIFO |
| `uint8* data` | Buffer of message to send |

**Output:**

| | |
|---|---|
| `int* written` | Number of bytes written (each character stored as a byte) |
| `int* remained` | Number of bytes remaining free in the TX FIFO |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | Error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not a CAN-503 |
| `DQ_BAD_PARAMETER` | `chnl`, `data`, or `written` is invalid |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function is only for use with Point-by-Point DAQ mode. It writes a block of data to the TX FIFO on a CAN-503 layer. If `requested == written`, all data fit into the FIFO.

**Notes:**

### *4.37.5 DqAdv503SetConfig*

**Syntax:**

```
int DqAdv503SetConfig(int hd, int devn, int sending, uint32
config)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int sending` | `TRUE` for the sending subsystem, `FALSE` for the receiving subsystem |
| `uint32 config` | configuration flags |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CAN-503 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets the configuration options for a 503 layer.

**Note:**

### *4.37.6 DqAdv503SetFilter*

**Syntax:**

```
int DqAdv503SetFilter(int hd, int devn, int channel, int flags,
int filter_size, uint32* filter)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | Channel number 0..3 |
| `int flags` | reserved |
| `int filter_size` | number of filter pairs in filter |
| `uint32* filter` | list of filter pairs, 32 pairs max |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CAN-503 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets a message id filter for a channel of a CAN-503 layer.

The filter is comprised of pairs of message id values. The pairs represent the start and end id values that the filter will pass. For example the filter: { 0x02, 0x04, 0x10, 0x10, 0x100, 102 } will allow messages with ids 0x02, 0x03, 0x04, 0x10, 0x100, 0x101 and 0x102 and reject all others

Calling this function with a `filter_size` of zero will clear any previous filter and allow all messages to pass.

**Note:**

### 4.37.7 DqAdv503SetMode

**Syntax:**

```
int DqAdv503SetMode(int hd, int devn, int chnl, uint32 cmd,
uint8 *value)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Channel to configure |
| `uint32 cmd` | what parameter to set |
| `uint8 *value` | value depending on `cmd` parameter<br>if cmd is `DQL_IOCTL503_SET_MASK`, value is 8-byte buffer containing 4-byte accept code and 4-byte accept mask<br>if cmd is `DQL_IOCTL503_SET_RATE`, value is two bytes for rate; byte 0 is BTR0 and byte 1 is BTR1<br>if cmd is `DQL_IOCTL503_SET_OPER_MODE`, value is one byte mode value<br>if cmd is `DQL_IOCTL503_SETTXDND` value is one uint32 coded over 4 bytes<br>if cmd is `DQL_IOCTL503_SETFFRQST` value is one uint32 coded over 4 bytes<br>if cmd is `DQL_IOCTL503_SETERRHDL` value is one uint32 coded over 4 bytes |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CAN-503 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets the communication mode for a 503 layer.

The following parameters are defined for cmd:

```
#define DQL_IOCTL503_SET_MASK      (0x1)  // set the ID accept mask and accept code
#define DQL_IOCTL503_SET_RATE      (0x2)  // set the communication bitrate
#define DQL_IOCTL503_SET_OPER_MODE (0x3)  // set the channel mode, normal or passive
#define DQL_IOCTL503_SETTXWM       (0x4)  // sets the TX watermark
#define DQL_IOCTL503_SETRXWM       (0x5)  // sets the RX watermark
#define DQL_IOCTL503_SETTXDND      (0x6)  // sets TX do-not-disturb
#define DQL_IOCTL503_SETFFRQST     (0x7)  // sets number of secondary FIFO
processings per second
#define DQL_IOCTL503_SETERRHDL     (0x8)  // sets error interrupt handling
```

**Note:**

**Acceptance mask and filters** are configured using two uint32:

```
uint32 modeparams[2];

code = 0x00000000;
mask = 0xFFFFFFFF;

modeparams[0] = htonl(code);
modeparams [1] = htonl(mask);

DqAdv503SetMode(hd,devn,chnl,DQL_IOCTL503_SET_MASK,(uint8*)modeparams)
```

The acceptance mask defines the bits that are relevant (0 is relevant, 1 is not) for comparison with the acceptance code.
The way you pack the bits in the mask and code depends on whether the CAN interface is configured in Basic or Extended mode and also on whether you want to filter incoming frames that use 11-bit IDs or frames that use 29-bit IDs.
The algorithm for defining the acceptance code and mask is described in the document at http://www.nxp.com/acrobat_download/applicationnotes/AN97076.pdf, starting at page 19.

**Example 1, 11-bit filtering:**

```
The following shows how to make a common filter configuration that
rejects 0x241, 0x514, and 0x4E1 and accepts 0x541 and 0x641.

0x241 010 0100 0001 ; lay out the bits to reject, convert hex to
; binary
0x514 101 0001 0100
0x4E1 100 1110 0001 ; lay out the bits to accept
0x541 101 0100 0001
```

```
0x641 110 0100 0001
Now shift the groupings left by one place and pad with 21 x’s on the
right (to make it a full 32-bit word).
reject 0100 1000 001x xxxx xxxx xxxx xxxx xxxx
“ 1010 0010 100x xxxx xxxx xxxx xxxx xxxx
“ 1001 1100 001x xxxx xxxx xxxx xxxx xxxx
accept 1010 1000 001x xxxx xxxx xxxx xxxx xxxx
“ 1100 1000 001x xxxx xxxx xxxx xxxx xxxx
‘row a’: 1xx0 100x 0x1x xxxx xxxx xxxx xxxx xxxx
      ; for each column, if both accept bits are the same
      ; and at least one reject bit is different,
      ; then copy the accept bit, otherwise make it x
1000 1000 0010 0000 0000 0000 0000 0000 ; make code: copy ‘row a’ and
; make all x’s be zero
8 8 2 0 0 0 0 0 ; convert binary to hex,
; the acceptance code = 0x88200000
0110 0001 0101 1111 1111 1111 1111 1111 ; make mask: put a 1 where
; ‘row a’
; has an x. The others become zero.
6 1 5 F F F F F ; convert binary to hex
; the acceptance mask = 0x615FFFFF
```

**Example 2, 29-bit filtering:**

```
     In this example, you want to receive messages with ID’s 18FEEE00,
     18FEF100 and 0CF00400 and receive as few other messages as possible.
     0x18FEEE00 1 1000 1111 1110 1110 1110 0000 0000 ;lay out the bits
     0x18FEF100 1 1000 1111 1110 1111 0001 0000 0000 ; “
     0x0CF00400 0 1100 1111 0000 0000 0100 0000 0000 ; “
     1100 0111 1111 0111 0111 0000 0000 0xxx ; shift groupings left, pad
     ; 3 x’s
     1100 0111 1111 0111 1000 1000 0000 0xxx ; “
     0110 0111 1000 0000 0010 0000 0000 0xxx ; “
     x1x0 0111 1xxx 0xxx xxxx x000 0000 0xxx ; find what’s common to all
3,
     ; else x
     0100 0111 1000 0000 0000 0000 0000 0000 ; make code, take common,
     ; x’s -> 0
     4 7 8 0 0 0 0 0 ; convert code to hex
     ;code is 0x47800000
     1010 0000 0111 0111 1111 1000 0000 0111 ; make mask, x’s become 1’s,
     ; else 0
     A 0 7 7 F 8 0 7 ; convert mask to hex
     ; mask is 0xA077F807
```

**Custom bit rate** is configured using two bytes **BTR0** and **BTR1**. The NXP SJA-1000 datasheet describes how to calculate the proper value for those two registers.

**Channel operating mode** is configured using one byte. Set it to 0 for normal mode or 1 for passive mode (port works in receive only mode without acknowledging CAN frames)

**Transmit do not disturb delay** adds a delay between the transmission of outgoing CAN frames. It is configured using one uint32 value. Use following formula to convert delay in microseconds:

txdonotdisturb = htonl(my_delay_us*(DQ_L503_BASE/1000000)-1);

**Secondary RX FIFO rate** specifies how often the logic process data sitting in the secondary RX FIFO. Itis configured using one uint32 value that specifies the processing rate in Hz:

rxfiforate = htonl(my_rate_in_hz)

**Inhibit error handling** is configured using one uint32 value:

DQ_L503_ALERR_INHIBIT: inhibit handling of arbitration lost error
DQ_L503_BEERR_INHIBIT: inhibit handling of bus error
DQ_L503_RST_ON_ERRPAS: reset and reenable controller on error passive
DQ_L503_RST_ON_BUSOFF: reset and reenable SJA1000 if it took itself off the bus

### *4.37.8 DqAdv503SetChannelCfg*

**Syntax:**

`int DqAdv503SetChannelCfg(int hd, int devn, int chnl, uint32 config)`

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Channel to configure |
| `uint32 config` | Configuration flags |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CAN-503 |
| `DQ_BAD_PARAMETER` | `cmd` is not one of the `DQL_IOCTL503_SET` constants, or `value` is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets the configuration options for a 503 layer

**Note:**

Configuration flags are assembled as follow:
RATE | OPERATING_MODE | PORT_MODE

Available rates are:

`DQ_CAN503_RATE_10K`
`DQ_CAN503_RATE_20K`

`DQ_CAN503_RATE_50K`
`DQ_CAN503_RATE_100K`
`DQ_CAN503_RATE_125K`
`DQ_CAN503_RATE_250K`
`DQ_CAN503_RATE_500K`
`DQ_CAN503_RATE_800K`
`DQ_CAN503_RATE_1M`

Available operating modes:

`DQ_CAN503_OPER_NORMAL`: Normal mode (CAN port acknowledges incoming frames)
`DQ_CAN503_OPER_LISTEN`: Listen only passive mode (no acknowledgement of incoming frames)

Available port modes:

`DQ_CAN503_MODE_BASIC`: CAN 2.0a – basic CAN (11 bit identifier)
`DQ_CAN503_MODE_XTEND`: CAN 2.0b – extended CAN (29 bit identifier)

### *4.37.9 DqAdv503SetWatermark*

**Syntax:**

```
int DqAdv503SetWatermark(int hd, int devn, int chnl, uint32
rx_or_tx, uint32 messages)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int devn` | layer inside the IOM |
| `int chnl` | device channel |
| `uint32 rx_or_tx` | watermark fifo direction |
| `uint32 messages` | watermark position |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-503 |
| `DQ_BAD_PARAMETER` | invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Allows user configuration of internal TX and RX FIFO watermarks to better control dataflow from the IOM.

**Note:**

Parameter dir is DQL_IOCTL503_SETTXWM or DQL_IOCTL503_SETRXWM and defaults to 1 and 0 as default.

### *4.37.10 DqAdv503ResetChannel*

**Syntax:**

```
int DqAdv503ResetChannel(int hd, int devn, int channel)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | Channel to reset |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CAN-503 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function resets SJA-1000 chip associated with the specified channel.

**Note:**

None

### *4.37.11 DqAdv503ParseVmapMsg*

**Syntax:**

```
int DqAdv503ParseVmapMsg(uint32 mode, uint8* rx, uint32* tstamp,
uint32* identifier, uint8 *buf, int *len)
```

**Input:**

| | |
|---|---|
| `uint32 mode` | Current mode of CAN port (same flags passed to DqAdv503SetChannelCfg() |
| `uint8* rx` | Received 16-bytes message (from VMap buffer) |

**Output:**

| | |
|---|---|
| `uint32* tstamp` | Message timestamp in 10us resolution |
| `uint32* identifier` | CAN frame ID |
| `uint8 *buf` | CAN frame payload (up to 8 bytes) |
| `int *len` | Number of bytes stored in payload buffer |

**Return:**

The number of bytes parsed in the RX buffer

**Description:**

This function converts CAN message from the 16-byte per message hardware representation to frame ID,payload and timestamp.

**Note:**

This is a helper function to use the CAN-503 layer in VMAP mode.

### *4.37.12 DqAdv503MakeVmapMsg*

**Syntax:**

```
int DqAdv503MakeVmapMsg(uint32 mode, uint32 identifier, uint8*
buf, int len, uint8* tx)
```

**Input:**

| | |
|---|---|
| `uint32 mode` | Current mode of CAN port (same flags passed to DqAdv503SetChannelCfg() |
| `uint32 identifier` | CAN frame ID |
| `uint8 *buf` | CAN frame payload (up to 8 bytes) |
| `int len` | Number of bytes stored in payload buffer |

**Output:**

| | |
|---|---|
| `uint8* tx` | Buffer formatted for transmission |

**Return:**

The number of bytes stored in the TX buffer

**Description:**

This function formats a CAN message from from frame ID and payload to 16-byte per message hardware representation.

**Note:**

This is a helper function to use the CAN-503 layer in VMAP mode.

### *4.37.13 DqAdv503ConfigEvents*

**Syntax:**

```
int DAQLIB DqAdv503ConfigEvents(int hd, int devn, int channel, int flags,
event503_t event, uint32* param)
```

**Command:**

Specify what asynchronous notification events user wants to receive from the layer

| **Input** | |
|---|---|
| int handle | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel |
| int flags | Behavior modification flags |
| event503_t event | Event type |
| uint32* param | Additional parameters depends on event type |
| **Output** | |
| None | |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not CAN-503 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |

| | |
|---|---|
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**

This function sets up events and their parameters which program firmware to send asynchronous event notification packet upon reaching certain conditions.

event503_t event data is defined as:

```
// Event data for 501 layer
typedef struct {
    uint32 chan;                    // channel information
    uint32 evtype;                  // type of the event
    uint32 tstamp;                  // timestamp of event
    uint32 size;                    // size of the following data in bytes
    uint32 avail;                   // number of messages available
                                    // (including just sent messages)
    uint8 data[];                   // data to follow (one character takes one byte)
} EV503_ID, *pEV503_ID;
```

When this event happens the firmware sends back a packet of the following structure:

```
/* DQCMD_EVENT structure */
typedef struct {
    uint8   dev;                    // device
    uint8   ss;                     // subsystem
    uint16  size;                   // data size
    uint32  event;                  // event type or command and additional flags
    uint8   data[];                 // data
} DQEVENT, *pDQEVENT;
```

<size> field contains sizeof(EV503_ID)+length.
<event> contains the type of the event, e.g. in this case EV503_IN_FIFO.
<data[]> if DQEVENT contains EV503_ID flexible structure:

Please notice that while DQEVENT is properly handled by `DqCmdReceiveEvent()` in terms of converting data fields from big endian (network data representation) to little endian if used with Intel-based host computer while EV503_ID needs to be converted by the user with the following code:

```
// shuffle bytes big endian->little endian in place
void ntoh_pEv503(pEV503_ID pEv503) {
    uint32 i;

    pEv503->chan = ntohl(pEv503->chan);
    pEv503->evtype = ntohl(pEv503->evtype);
    pEv503->size = ntohl(pEv503->size);
    pEv503->tstamp = ntohl(pEv503->tstamp);
    for (i = 0; i < pEv503->size/sizeof(uint8); i++)
        pEv503->data[i] = (pEv503->data[i]);
    return;
}
```

The following event types are defined:

```
typedef enum {
    EV503_CLEAR = 0x1000,           // clear all events

    EV503_RX_PERIODIC = 0x101,      // periodic event
    // FIFO events
    EV503_IN_FIFO,                  // input FIFO above watermark (data included)
    EV503_OUT_FIFO,                 // output FIFO below watermark

    // Bus errors
    EV503_ERROR_BUS,                // bus or protocol errors
    EV503_ERROR_WARNING,            // bus warning level has been reached
    EV503_ERROR_PASSIVE,            // bus in passive state
    EV503_ERROR_BUSOFF,             // chip is taqken off the bus

    // Firmware errors
    EV503_RX_OVERFLOW,              // RX FIFO overflow - cleared
    EV503_RX_UNXP,                  // unexpected RX error (most likely receiveing error)
    EV503_TX_UNXP                   // unexpected TX error (most likely transmitting error)
} event503_t;
```

<flags> can contain the follwing bits:

```
#define DQEVENT_NOREPLY             (1L<<31)    // do not reply to this event
#define DQEVENT_ONCE                (1L<<30)    // fire event only once
#define DQEVENT_NOSTORE             (1L<<29)    // do not store UDP port to
generate events
```

If `DQEVENT_NOREPLY` bit is set a function call `DqAdv503ConfigEvents()` doesn’t wait for the answer from the firmware on the success of this function call. This comes handy in the situation when user wants to change event parameter (e.g. watermark level) on the fly to avoid increase in the network traffic.
If `DQEVENT_ONCE` is send the firmare sends event packet only once and then event becomes disabled until a next call to `DqAdv503ConfigEvents()` re-enables it.
`DQEVENT_NOSTORE` is required to tell firmware not to change the reply UDP port. For example, user can create multiple UDP ports using `DqAddIOMPort()` to listen to incoming events on the different ports. Thus, when the function called from the handle other than the handle event is expected upon this bit need to be set. Do not set this bit for the first time it is used.

To clear all events use the following call:

```
ret = DqAdv503ConfigEvents(a_handle, devn, channel, 0, EV503_CLEAR, NULL);
```

It clears all events that has previously been set along with runtime variables and accumulated events.

`EV503_RX_PERIODIC` - event is sent evety time counter set up using 66MHz timebase base clock expires:

```
ev_param[0] = DQ_LN_1s_TIMESTAMP;   // frequency divider
ev_param[1] = DQ_L503_RX_FIFO_MAXMSG;    // get all messages
DqAdv503ConfigEvents(handle, DEVN, channel, 0, EV503_RX_PERIODIC, ev_param)
```

This event takes two parameters to set up – a frequency divider (in the above smippet one of the timestamp constant is used for 1s timeout value) and the number of the messages to receive upon timeout. The minimum time between messages allowed is 10us.
The event packet is sent when there is no activity on the bus for the duration of the selected timeout. Since it is possible to set up timeout to overload firmware there is a mechanism in ISR routine that disables any offending interrupts and sets a flag STS_FW_OVERLOAD in firmware status filed STS_FW.
Each CAN message contains 16 bytes of data.

When event packet is sent the following data is stored:

```
pEv503->evtype = dobj->chnls[j].mode; // current channel mode
pEv503->avail = in_fifo/DQ_L503_RX_FIFO_MSGSZ32;    // in messages
pEv503->size = messages*sizeof(uint32);    // number of copied bytes
pEv503->chan = channel;   // message channel
pEv503->tstamp = current_timestamp;
```

`EV503_IN_FIFO` - input FIFO above watermark. The watermark level is set in messages:

```
ev_param[0] = 1;    // one message per event
DqAdv503ConfigEvents(handle, DEVN, channel, 0, EV503_IN_FIFO, ev_param)
```

The minimum number of messages is 1 and the maximum number is limited by the size of the Rx FIFO and amount of data can be sent in one packet (DQ_L503_RX_FIFO_MAXMSG = = 90). If the amount of data received in the FIFO exceeds watermark but cannot be sent in one packet multiple packets will be sent back-to-back until the level in the FIFO drops below watermark.

`EV503_OUT_FIFO` - output FIFO below watermark. The event is sent upon Tx FIFO half-empty and empty interrupt.

```
pEv503->evtype = dobj->chnls[j].mode;  // current channel mode
pEv503->avail = in_fifo;  // free space in the Tx FIFO, messages
pEv503->size = 0;
pEv503->chan = channle;
pEv503->tstamp = current_timestamp;
```

Upon an error interrupt received from CAN 2.0B Philips SJA1000 chip one of the following events can be generated:

`EV503_ERROR_BUS` - bus or protocol errors.
`EV503_ERROR_WARNING` - bus warning level has been reached
`EV503_ERROR_PASSIVE` - bus in passive state
`EV503_ERROR_BUSOFF` - chip is taken off the bus
`EV503_RX_UNXP` - unexpected RX error (most likely receiveing error)

If one of these events occurs the following information is sent in the event packet:

```
data32[0] = ((REC<<24)|(TEC<<16)|(ECC<<8)|(ALC));
data32[1] = ((MCR<<16)|(IR<<8)|(SR));
```

REC is Rx Error Counter
TEC is Tx Error Counter

ECC is Error Code Capture (see section 6.4.9 of SJA1000 datasheet to decode at what part of the message an error has accured)
ALC is Arbitration Lost Counter
MCR is SJA1000 Mode Control Register data
SR is SJA1000 Status Register
IR is SJA1000 Interrupt Register

`EV503_RX_OVERFLOW` or `EV503_TX_UNXP` events are sent automatically upon processing either EV503_RX_PERIODIC or EV503_IN_FIFO:

`EV503_RX_OVERFLOW` - RX FIFO overflow - cleared
`EV503_TX_UNXP` - unexpected TX error (most likely transmitting error)

Data sent:

pEv503->evtype = isr;
pEv503->avail = in_fifo;
pEv503->size = 0;
pEv503->chan = chan;
pEv503->tstamp = current_timestamp;

Since the FIFO is not corrupted upon overflow (the FIFO size is 1024 uint32s and each message takes four uint32 words) it is not cleared and status STS_FW adds STS_FW_BUF_OVER flag. New messages are not accepted into the FIFO until user retrives messages from it to clear up the space.

CAN messages in FIFO are stored in SJA1000 data format and needs to be decoded as follows:

```
// Function to convert basic CAN messages from the event message
// ---------------------------------------------------------------------------
int DqAdv503BasicRxEventConvert(uint8* message, uint32* tstamp, uint32 *pktid, uint8 *buf,
int *len) {
    int i, msglen, val, tid;
    uint8* rx =  message;
    uint32* rx32 = (uint32*)message;
    tid = 0;

    *tstamp = (rx32[0]>>8);

    // get second byte with msglen + RF flag + couple identifier bits
    val = rx[4];
    msglen = min(CAN_LEN(val), 8);

    // get first byte with identifier
    tid = val >> 5;
    tid |= rx[3] << 3; // upper

    // copy msglen bytes
    for (i = 0; i < msglen; i++) {
        val = rx[i+5];
        buf[i] = val;
    }

    *pktid = tid;  // return identifier
    *len = msglen; // return length

    return DQ_SUCCESS;
}

// ------------------------------------------------------------------------------------
// Function to convert Pelican read from the event message
int DqAdv503PelicanRxEventConvert(uint8* message, uint32* tstamp, uint32 *pktid, uint8 *buf,
int *len) {
    int i, msglen, val, tid, ofs;
    uint8* rx =  message;
    uint32* rx32 = (uint32*)message;

    // Get timestamp first
    *tstamp = (rx32[0]>>8);
    val = rx[3];    // HDR0
    msglen = CAN_LEN(val);// mask off length 0x0f;
    tid = 0;

    if (CAN_EFF(val)) { // 29 bits
        tid |= (rx[4] & 0xff) << 21;
        tid |= (rx[5] & 0xff) << 13;
        tid |= (rx[6] & 0xff) << 5;
        tid |= (rx[7] & 0xf8) >> 3;
        ofs = 8;
    } else {    // 11 bits
        tid |= (rx[4] & 0xff) << 3;
        tid |= (rx[5] & 0xe0) >> 5;
        ofs = 6;
    }

    if (msglen > 8) {
        msglen = 8;
    }

    for (i = 0; i < msglen; i++) {
        val = rx[i+ofs];
        buf[i] = val;
    }

    *pktid = tid; // return identifier
    *len = msglen; // return length

    return DQ_SUCCESS;
}
```

## 4.38 DNA-SL-504 layer

### 4.38.1 DqAdv504Enable

**Syntax:**

`int DqAdv504Enable(int hd, int devn, int enable_mask)`

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int enable_mask` | “1” in the channel position bit to enable, “0” to disable |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-504 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Starts/Stops requested channels on the SL-504.

**Note:**

For example to enable channels 1 and 3 use enable_mask = (1<<1)|(1<<3)

### 4.38.2 DqAdv504GetStatus

**Syntax:**

```
int DAQLIB DqAdv504GetStatus(int hd, int devn, int chan_mask,
pSL504_INT_STAT status)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chan_mask` | Channel mask to specify which channels to include in status request |
| `pSL504_INT_STAT status` | Pointer to the structure to store status/statistics information |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-504 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Requests status and accumulated statistics from SL-504.

**Note:**

For example to request status from channels 1 and 3 use chan_mask = (1<<1)|(1<<3)

### 4.38.3 DqAdv504SetConfig

**Syntax:**

```
int DAQLIB DqAdv504SetConfig(int hd, int devn, int channel, pSL504_SETCFG
pCfg)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | Channel to set configuration for |
| `pSL504_SETCFG` | Pointer to the configuration structure |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-504 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Set channel configuration for SL-504

`SL504_SETCFG` is defined as follows:

```
typedef struct {
    uint32 protocol;        // protocol used: HDLC/sync/async/flags
    uint32 modeflags;       // additional flags for mode selection
    uint32 physical;        // physical interface RS-232/422/485

    // fill the following section for HDLC or RawSync modes
    uint32 hdlc_flags;      // additional synchronous mode flags
    uint32 hdlc_encod;      // encoding type - NRZ, MII, etc.
    uint32 hdlc_baud;       // RxC or TxC clock rate in baud
    uint32 hdlc_clk_src;    // Which clock to synchronize Tx from
    uint32 hdlc_crc_mode;   // None, CRC-16/32/CCITT, user-supplied or automatic
    uint32 hdlc_flt_mode;   // Filter type
    uint32 hdlc_filter;     // HDLC address filter value (8 bits)
    uint32 hdlc_preamble;   // preamble
    uint32 hdlc_prmbl_sz;   // size of preamble
    uint32 hdlc_idle_ch;    // idle character (default 0x7E)

    // fill the following section for asynchronous mode
    uint32 async_baud;      // baud rate
    uint32 async_char_sz;   // 5 to 8 bits before parity bit, 9th bit is after parity
    uint32 async_start;     // number of start bits (=0 is default means 1 bit)
    uint32 async_stop;      // number of stop bits (=0 is default means 1 bit)
    uint32 async_parity;    // parity used
    uint32 async_msglen;    // watermark level defines message lenght
    uint32 async_tout;      // timeout forces to transmit everything accumulated (in ms)

} SL504_SETCFG, *pSL504_SETCFG;
```

This structure is used for all modes of operation albeit only a part of it is valid for each mode.

For HDLC mode:

<protocol> can be selected from:

```
// <protocol>
#define SL504_PROT_HDLC         1   // HDLC protocol support
```

```
#define SL504_PROT_ASYNC        2   // Asycnrhonous mode (character-based)
#define SL504_PROT_SYNC         3   // <reserved> Bit-synchronous, external sync, no framing or
synchronization
#define SL504_PROT_ISOCHRONOUS  4   // <reserved> isochronous mode
#define SL504_PROT_NINEBIT      5   // <reserved> 9th bit address mode
#define SL504_PROT_RESV0        6   // <reserved>
#define SL504_PROT_BIMONOSYNC   7   // <reserved> monosync or bisync mode
#define SL504_PROT_T_BISYNC     8   // <reserved> transparent bisync mode
#define SL504_PROT_S_MONOSYNC  10   // <reserved> slaved monosync mode
```

Only two protocols are currently implemented:

`SL504_PROT_HDLC` – works with HDLC and SDLC protocol

`SL504_PROT_ASYNC` – works with asynchronous protocol

Once protocol is selected there are a few protocol modifiers for HDLC mode:

<modeflags>

```
#define SL504_HDLC_LOOP         (1L<<0)   // <reserved> HDLC loop mode
#define SL504_USE_CTS           (1L<<4)   // CTS input controls transmission
#define SL504_USE_DCD           (1L<<5)   // DCD enables receiver

#define SL504_INHIBIT_TX        (1L<<6)   // Do not enable transmitter
#define SL504_INHIBIT_RX        (1L<<7)   // Do not enable receiver
```

SL-504 layer has two inputs per channel: one is CTS and the second one is DCD. By selecting bits 4 and 5 user gives control over transmission enable to CTS input and/or for receiver enable to DCD line. User can also force transmitter and/or receiver into disabled state if appropriate flags are added to the <modeflags> word.

<physical> applies to all modes of operation and selects the physical interface SL-504 board uses to communicate with other devices.

```
#define SL504_PHY_RS232         1   // RS-232 up to 230k baud
#define SL504_PHY_RS485         2   // RS-485 up to 4Mbit
#define SL504_PHY_RS422         3   // RS-422 multidrop
#define SL504_PHY_V35           4   // balanced current data and clock and
                                    // unbalanced voltage DCD and CTS

#define SL504_PHY_TERM          0x40    // enable termination in RS-485 and RS-422 modes
#define SL504_PHY_LOOP          0x80    // enable loopback in the selected mode
#define SL504_PHY_NOECHO        0x100   // suppress echo in RS-422/423 modes
```

`<SL504_PHY_TERM>` enables termination on RS-485 or RS-422 bus

`<SL504_PHY_LOOP>` set transceiver chip into loopback mode so all transmitted data is received by the receiver without a need of external dongle

`<SL504_PHY_NOECHO>` is used to disable receiver in RS-422 mode while transmitter transmits data on the bus. Since this feature is implemented in the software it only works without queuing of transmit packets, thus the depth of the transmit DMA FIFO becomes one frame.

<hdlc_flags> define how to handle HDLC protocol

```
#define SL504_HDLC_ABORT_7        (0L<<0)   // send 0x7f as an abort symbol
#define SL504_HDLC_ABORT_15       (1L<<0)   // send 0x7fff as an abort symbol
#define SL504_HDLC_FINISH_UNDER   (2L<<0)   // in underrun condition close the
                                            // frame by adding CRC to it
#define SL504_HDLC_FLAGS_UNDER    (3L<<0)   // in underrun condition start sending flags
#define SL504_HDLC_SHARED_ZEROES  (1L<<6)   // send idle flags with shared zeroes
```

<hdlc_encod> selects what type of encoding is used. The most popular type of encoding is NRZ, other types of encodings are rarely used.

```
#define SL504_HDLC_NRZ            (0L<<0)   // NRZ encoding
#define SL504_HDLC_NRZB           (1L<<0)   // inverted NRZ encoding
#define SL504_HDLC_NRZI           (2L<<0)   // NRZI encoding
#define SL504_HDLC_NRZI_MARK      (3L<<0)   // NRZI encoding, invert state for 1
#define SL504_HDLC_NRZI_SPACE     (4L<<0)   // NRZI encoding, invert state for 0
#define SL504_HDLC_BIPHASE_MARK   (5L<<0)   // biphase encoding, used with DPLL
#define SL504_HDLC_BIPHASE_SPACE  (6L<<0)   // biphase encoding, used with DPLL
#define SL504_HDLC_BIPHASE_LEVEL  (7L<<0)   // biphase encoding, used with DPLL
#define SL504_HDLC_BIPHASE_DIFF   (8L<<0)   // biphase encoding, used with DPLL
```

<hdlc_baud> used to select baud rate

<hdlc_clk_src> defines where receiver and transmitter receive clock from.

```
#define SL504_HDLC_FLAG_RXC_RXCPIN  (1L<<0)    // RxClk from RxC pin (default)
#define SL504_HDLC_FLAG_RXC_DPLL    (2L<<0)    // RxClk from DPLL
#define SL504_HDLC_FLAG_RXC_BRG     (3L<<0)    // RxClk from BRG0

#define SL504_HDLC_FLAG_TXC_BRG     (1L<<2)    // TxClk from BRG0 (default)
#define SL504_HDLC_FLAG_TXC_DPLL    (2L<<2)    // TxClk from DPLL
#define SL504_HDLC_FLAG_TXC_RXCPIN  (3L<<2)    // TxClk from RxC pin

#define SL504_HDLC_FLAG_DPLL_DIV8   (1L<<4)    // DPLL divider 8
#define SL504_HDLC_FLAG_DPLL_DIV16  (2L<<4)    // DPLL divider 16
```

There are three main sources of the clock available for both transmitter and receiver. Both of them can be programmed to use any of these clocks.

<hdlc_crc_mode> defines CRC mode used with HDLC protocol:

```
#define SL504_HDLC_CRC_NONE       (0L<<0)   // CRC is not check neither for transmit
                                            // nor receive
#define SL504_HDLC_CRC_USER       (1L<<0)   // User is responsible for inserting and
                                            // checking CRC
#define SL504_HDLC_CRC_16_CCITT   (2L<<0)   // 16-bit CCITT CRC is used
#define SL504_HDLC_CRC_16         (3L<<0)   // 16-bit polynomial CRC is used
                                            //
#define SL504_HDLC_CRC_32         (4L<<0)   // 32-bit Eth CRC is used
```

16-bit CCITT CRC uses (x^15+x^12+x^5+1) polynomial.
16-bit CRC uses (x^16+x^15+x^2+1) polynomial.
32-bit CRC uses (x^32+x^26+x^23+x^22+x^16 +x^12+x^11+x^10+x^8+x^7 +x^5+x^4+x^2+x+1) polynomial.

HDLC protocol allows filtering of the received packets by its
<hdlc_flt_mode> selects what type of filtering to use (selection of the type of filtering is application dependent)

```
#define SL504_HDLC_FLT_NONE       (0L<<0)   // No filtering
#define SL504_HDLC_FLT_A_16       (1L<<0)   // +16 bits into RxFIFO if Addr
                                            // matches or B/C as 2 bytes
#define SL504_HDLC_FLT_A_24       (2L<<1)   // +24 bits into RxFIFO if Addr
                                            // matches or B/C as 3 bytes
#define SL504_HDLC_FLT_A_32       (6L<<1)   // +32 bits into RxFIFO if Addr
                                            // matches or B/C as 4 bytes
#define SL504_HDLC_FLT_EA_LS      (7L<<1)   // Places bytes while LS==0,
                                            // then byte with LS==1 then 16 bits
                                            // as 2 bytes into RxFIFO if EA
                                            // matches or B/C
#define SL504_HDLC_FLT_EA_24      (3L<<1)   // Places 24 bits as 3 bytes
```

```
                                          // into RxFIFO if EA matches or B/C
#define SL504_HDLC_FLT_EA_MS      (11L<<1)  // Places bytes while MS==0, then byte
                                          // with MS==1 then 8 bits
                                          // as 1 byte into RxFIFO if Ext Addr
                                          // matches or B/C
#define SL504_HDLC_FLT_EA_MS16    (15L<<1)  // Places bytes while MS==0,
                                          // then byte with MS==1 then 16 bits
                                          // as 2 bytes into RxFIFO if Ext Addr
                                          // matches or B/C
```

<hdlc_filter> field contains 8-bit filter address value if filtering is allowed in <hdlc_flt_mode>.

HDLC protocol can operate with different preamble sizes defined in <hdlc_prmbl_sz> if preamble is selected in <hdlc_preamble> field.

```
#define SL504_HDLC_PRMBSZ_16      (0L<<0)   // 16-bit preamble is used
#define SL504_HDLC_PRMBSZ_32      (1L<<0)   // 32-bit preamble is used
#define SL504_HDLC_PRMBSZ_64      (2L<<0)   // 64-bit preamble is used
```

HDLC preamble pattern is programmed in <hdlc_preamble> field and can be one of the follows allowed by the protocol. In most cases no preamble is used.

```
#define SL504_HDLC_PRMB_NONE      (0L<<0)   // no preamble
#define SL504_HDLC_PRMB_ZERO      (1L<<0)   // all zeroes
#define SL504_HDLC_PRMB_ONE       (2L<<0)   // all ones
#define SL504_HDLC_PRMB_FLAG      (3L<<0)   // all flags
#define SL504_HDLC_PRMB_10        (4L<<0)   // alternating 1 and 0
#define SL504_HDLC_PRMB_01        (5L<<0)   // alternating 0 and 1
```

HDLC is different from many other protocols by the fact that it continuously transmits idle characters when there is no data to transmit. Idle character can be selected from one of the follows:

```
#define SL504_HDLC_IDLE_FLAG      (0L<<0)   // continuous flags (0x7E)
#define SL504_HDLC_IDLE_ZERO      (1L<<0)   // continuous zeroes
#define SL504_HDLC_IDLE_ONE       (2L<<0)   // continuous ones
#define SL504_HDLC_IDLE_MARK      (3L<<0)   // idle chars are marks
#define SL504_HDLC_IDLE_SPACE     (4L<<0)   // idle chars are spaces
#define SL504_HDLC_IDLE_MS        (5L<<0)   // alternating Mark and Space
#define SL504_HDLC_IDLE_01        (6L<<0)   // alternating 0 and 1
```

Most implementation use flag (0x7E) as an idle character.

When asynchronous mode is selected, HDLC related fields should be zero and vice versa.

In asynchronous use the following fields to program mode of operation:

<async_baud> - select baud rate (1200 through 230kbaud)

<async_char_sz> - select number of bits before parity bit (from 5 to 8), if 9 bit is selected it is inserted after parity bit. 9-bit asynchronous mode sometimes used in multi-drop protocols to distinguish between device address and device data.

<async_start> - number of start bits (=0 is default means 1 bit)
<async_stop> - number of stop bits (=0 is default means 1 bit)

<async_parity> - selects whether to insert a parity bit after the data bits and

```
#define DQ_SL504_PARITY_NONE    (0L<<0)   // none (parity field is not
                                        // populated)
```

```
#define DQ_SL504_PARITY_EVEN    (1L<<0)   // =1 if sum of data bits is even
#define DQ_SL504_PARITY_ODD     (2L<<0)   // =1 if sum of data bits is odd
#define DQ_SL504_PARITY_SPACE   (3L<<0)   // always space (constant 0)
#define DQ_SL504_PARITY_MARK    (4L<<0)   // always mark (constant 1)
```

Fields:
<async_msglen> - watermark level defines message length and
<async_tout> - timeout forces to transmit everything accumulated (in ms) are reserved for messaging and asynchronous event modes

**Note:**

For example to request status from channels 1 and 3 use chan_mask = (1<<1)|(1<<3)

### 4.38.4 DqAdv504SendFrame

**Syntax:**

```
int DqAdv504SendFrame(int hd, int devn, int chnl, int flags, uint8 *data,
int size, int *written, int *available)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Channel to send frame to |
| `int flags` | Reserved |
| `uint8* data` | Frame data to transmit |
| `int size` | Size of the frame |
| `int* written` | Number of characters written |
| `int* available` | Number of frame entries still available on the card |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-504 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Writes a frame of HDLC data to the card. If (size == written) then all data has been sent.

**Note:**

It might take multiple transactions on the Ethernet bus to deliver the whole frame.
By default sixteen 4096 bytes frames are allocated for each channel, for transmit and receive separately.

### 4.38.5 DqAdv504SendMultFrames

**Syntax:**

```
   int DqAdv504SendMultFrames(int hd, int devn, int chnl, int flags, int
n_frames, int *frm_size, uint8 *frm_data, int *written, int *available)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Channel to send frame to |
| `int flags` | Reserved |
| `int n_frames` | Number of frames in the buffer (16 max) |
| `int* available` | Array of frame sizes |
| `uint8 *frm_data` | Frame data to transmit |
| `int *written` | Number of bytes written or 0 if layer is busy trasnsmitting |
| `int* available` | Number of frame entries still available on the card |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-504 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Writes n_frames short frames of HDLC data to the card. The frames need to be placed one after the other in the frm_data array. The maximum number of frames is 16 and the total size of the frame data cannot exceed 4062 bytes.

**Note:**

It might take multiple transactions on the Ethernet bus to deliver the whole frame.
By default sixteen 4096 bytes frames are allocated for each channel, for transmit and receive separately.

### *4.38.6 DqAdv504RecvFrame*

**Syntax:**

```
int DqAdv504RecvFrame(int hd, int devn, int chnl, int flags, uint8 *data,
int size, int *received, int *available, int *rsb)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Channel to receive frame from |
| `int flags` | Reserved |
| `uint8* data` | Pointer to array to store received data |
| `int size` | Requested size of the frame (up to 4096). Rest is discarded. |
| `int* written` | Number of characters received |
| `int* available` | Number of frames with data available on the card |

| | |
|---|---|
| `int* rsb` | Receive status block from the interface |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-504 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_FIFO_OVERFLOW` | the receive fifo has overflowed; receiver must be restarted |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Reads a HDLC data frame from the SL-504.

RSB is a receive status block produced by Z16C32 IUSC when it stores frame into the memory. It is similar but not identical to the RCSR register available via DqAdv504GetStatus() and corresponds to this particular frame

| Bit | Name | Description |
|---|---|---|
| 11-9 | RxResidue | Number of bits in the last received byte of the frame<br>0 = 8 bits<br>1 = 1 bit<br>…<br>7 = 7 bits |
| 8 | ShortF | 1 = Received frame ended before address/control field |
| 5 | RCCF Overflow | 1 = RCC Residual value is not valid |
| 3 | CRCE | 1 = CRC Error |
| 2 | Abort | 1 = Rx Frame aborted |
| 1 | Rx Over | 1 = Rx FIFO overflow |
| 0 | RCC0 | 1 = Rx FIFO is not empty |

**Note:**

It might take multiple transactions on the Ethernet bus to deliver the whole frame.
By default sixteen 4096 bytes frames are allocated for each channel, for transmit and receive separately.

### *4.38.7 DqAdv504RecvMultFrames*

**Syntax:**

```
int DqAdv504RecvMultFrames(int hd, int devn, int chnl, int flags,
int n_frames, int frm_data_sz, uint8* frm_data, int* rcvd_frames, int* frm_size,
int* rsb_arr, int *received, int *available)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Channel to send frame to |
| `int flags` | Reserved |
| `int n_frames` | Maximum number of frames (can not be > 16) |
| `int frm_data_sz` | Size of allocated array to store data |
| `uint8* frm_data` | Array of frame data to receive |
| `int* rcvd_frames` | Number of frames received (16 max) |
| `int* frm_size` | Array of frame sizes |
| `int* rsb_arr` | Array of receive status blocks |
| `int* received` | Number of bytes received |
| `int* available` | Number of frame entries still available on the card |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-504 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Reads up to 16 frames of HDLC data from the card. The frames are placed one after the other in the frm_data array. The function will read 16 frames, n_frames, or frm_data_sz bytes; which ever comes first.

RSB is a receive status block produced by Z16C32 IUSC when it stores frame into the memory. It is similar but not identical to the RCSR register available via DqAdv504GetStatus() and corresponds to this particular frame

| Bit | Name | Description |
|---|---|---|
| 11-9 | RxResidue | Number of bits in the last received byte of the frame<br>0 = 8 bits<br>1 = 1 bit<br>…<br>7 = 7 bits |
| 8 | ShortF | 1 = Received frame ended before address/control field |
| 5 | RCCF Overflow | 1 = RCC Residual value is not valid |
| 3 | CRCE | 1 = CRC Error |
| 2 | Abort | 1 = Rx Frame aborted |
| 1 | Rx Over | 1 = Rx FIFO overflow |

| | | |
|---|---|---|
| 0 | RCC0 | 1 = Rx FIFO is not empty |

**Note:**
It might take multiple transactions on the Ethernet bus to deliver the whole frame.
By default sixteen 4096 bytes frames are allocated for each channel, for transmit and receive separately.

### 4.38.8 DqAdv504SendMessage

**Syntax:**

```
int DqAdv504SendMessage(int hd, int devn, int chnl, int flags, uint8 *data,
int size, int *written, int *available)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Channel to send frame to |
| `int flags` | Reserved |
| `uint8* data` | Characters to transmit |
| `int size` | Number of characters to send |
| `int* written` | Number of characters sent |
| `int* available` | Number free space available in transmit FIFO |

**Output:**
None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-504 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**
Sends a number of characters from the SL-504 channel in asynchronous mode

**Note:**
Currently firmware doesn't take advantage of DMA programming in asynchronous mode and the number of characters is limited by 32-bytes FIFO size

### 4.38.9 DqAdv504RecvMessage

**Syntax:**

```
int DqAdv504RecvMessage(int hd, int devn, int chnl, int flags, uint8 *data,
int size, int *received, int *available)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Channel to send frame to |
| `int flags` | Reserved |
| `uint8* data` | Array to store received data |
| `int size` | Number of characters requested to read |
| `int* received` | Number of characters returned |
| `int* available` | Number free space available in receive FIFO |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-504 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Receives a number of characters from the SL-504 channel in asynchronous mode

**Note:**

Currently firmware doesn't take advantage of DMA programming in asynchronous mode and the number of characters is limited by 32-bytes FIFO size

### *4.38.10 DqAdv504AbortTx*

**Syntax:**

```
int DAQLIB DqAdv504AbortTx(int hd, int devn, int channel, uint32* status)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | Channel to abort transmission at |
| `uint32* status` | Current value of TCSR status register |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-504 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |

| | |
|---|---|
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**
Abort transmission and return TX status. Clears all DMA buffers.

**Note:**

## 4.39 DNA-SL-514 Layer

### 4.39.1 DqAdv514Enable

**Syntax:**

```
int DqAdv514Enable(int hd, int devn, int channel_mask)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel_mask` | Bitmask of enabled channels:<br>“1” in the channel bit position to enable, “0” to disable |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating memory buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-514 |
| `DQ_ BAD_PARAMETER` | configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Enables/disables SSI master/slave channels on the SL-514. Each channel can be enabled or disabled independently with `channel_mask`.

**Notes:**

1. As an example, to enable channels 1 and 3 use: `channel_mask = (1<<1) | (1<<3);` or `channel_mask = 0xA;`
2. Master and slave ports are individually enabled or disabled using the `DqAdv514Config()` API.

### *4.39.2 DqAdv514Config*

**Syntax:**

```
int DAQLIB DqAdv514Config(int hd, int devn, int channel, pL514_CONFIG
config)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | Channel to set configuration for |
| `pL514_CONFIG config` | Pointer to the configuration structure |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating memory buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-514 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets channel configuration for the SL-514.

The `pL514_CFG` structure consists of the following elements:

```
// SSI channel configuration
typedef struct {
    uint32 ch_cfg; // channel configuration
    uint32 flags;  // <Reserved>

    uint32 clk_source; // clock source for master
    uint32 baud_rate;  // master baud rate (100Hz to 1.3 MHz)

// Master SSI setting (sends clocks, receives data)
    uint32 m_word_sz;  // master word size (3 to 32 bits)
    uint32 m_debounce; // debouncing settings (0=bypass)
    uint32 m_trigger;  // Tx trigger source
    uint32 m_Tv;       // Tv, master
    uint32 m_Tp;       // Tp, master
    uint32 m_Tm;       // Tm, master

// Slave SSI setting (sends data upon receiving clocks)
    uint32 s_word_sz;  // master clock size
    uint32 s_debounce; // debouncing settings
    uint32 s_trigger;  // Tx trigger source
    uint32 s_Tv;       // Tv, master
    uint32 s_Tp;       // Tp, master
    uint32 s_Tm;       // Tm, master
} L514_CONFIG, *pL514_CONFIG;
```

Configuration options for each `pL514_CONFIG` element are described below:

| | |
|---|---|
| `ch_cfg` | The following can be logically ORed together:<br>• L514CFG_SLAVE_EN: enable slave port for the channel<br>• L514CFG_MASTER_EN: enable master port for the channel<br>• L514CFG_MASTER_TS: add timestamp to the master data<br>• L514CFG_OUTPUTS_TERMN: enable termination on channel outputs<br>• L514CFG_INPUTS_TERMN: enable termination on channel inputs<br>• L514CFG_SLAVE_DATA_EN: enable line drive Tx<br>• L514CFG_MASTER_CLK_EN: enable line drive Rx<br>• L514CFG_SLAVE_Tprm: enable slave T parameters (0 = use defaults)<br>• L514CFG_MASTER_Tprm: program master T parameters (0 = use defaults) |
| `flags` | <Reserved> |
| `clock_source` | Sets clock source to use to derive master clock (for baud):<br>• L514CFG_CLK_BASE: use system 66MHz clock and divide it<br>• L514CFG_CLK_PLL: use on-board PLL as a clock source |
| `baud_rate` | 300 baud to 1.3 Megabaud (uint32) |
| `m_word_sz` | master 3 to 32 bits in the word (uint32) |
| `m_debounce` | master debouncing settings:<br>• 0=bypass<br>• 1 thru 15 = 4 thru 18, 15 ns clocks, (i.e., programming a “1” results in 4*15ns, or ~60 ns debouncing delay) |
| `m_trigger` | master trigger source that allows data to start transmission:<br>• L514CFG_TRIG_IMM: start immediately after enable, transmit when the data is in the buffer<br>• L514CFG_TRIG_GLOBAL: wait for the global trigger to start |
| `m_Tv` | master tv time delay (time from clock rising edge to data valid) |
| `m_Tp` | master pause time delay setting (time between the rising edge of the clock after the LSB transmission and the falling edge of the first clock of the next data word) |
| `m_Tm` | master transfer timeout/monoflop time setting (time delay between the data driving low after the LSB to driving high, signifying the port has entered the IDLE state) |
| `s_word_sz` | slave 3 to 32 bits in the word (uint32) |
| `s_debounce` | slave debouncing settings:<br>• 0=bypass<br>• 1 thru 15 = 4 thru 18, 15 ns clocks, (i.e., a “1” results in 4*15ns, or ~60 ns debouncing delay) |

| | |
|---|---|
| `s_trigger` | slave trigger source that allows data to start transmission:<br>• L514CFG_TRIG_IMM: start immediately after enable, transmit when the data is in the buffer<br>• L514CFG_TRIG_GLOBAL: wait for the global trigger to start |
| `s_Tv` | slave tv time delay (time from clock rising edge to data valid) |
| `s_Tp` | slave pause time delay setting (time between the rising edge of the clock after the LSB transmission and the falling edge of the first clock of the next data word) |
| `s_Tm` | slave transfer timeout/monoflop time setting (time delay between the data driving low after the LSB to driving high, signifying the port has entered the IDLE state) |

**Notes:**

### 4.39.3 DqAdv514Status

**Syntax:**

```
int DAQLIB DqAdv514Status(int hd, int devn, int chan_mask, pL514_STATUS
status)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel_mask` | Bitmask of channels:<br>"1" to capture status information,<br>and "0" to ignore |
| `pL514_STATUS status` | Pointer to store array of `L514_STATUS`<br>1 for each channel enabled in `channel_mask` |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating memory buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-514 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Requests status information from one or more SL-514 channels. Note that reading the status register for a channel clears the channel's sticky bits.

- `<channel_mask>` identifies which channels are read, (e.g. `channel_mask=0x9`; will read status on channel 0 and channel 3.)

`- <status>` is an array of returned status registers, (e.g. if your `channel_mask` indicates 2 channels, you will receive an array of 2.)

Status is returned as type `pL514_STATUS`. The `pL514_STATUS` structure consists of two elements:

```
// SSI channel status
typedef struct {
   uint32 ch_status; // channel status
    uint32 flags;     // <Reserved>
} L514_STATUS, *pL514_STATUS;
```

`<ch_status>` bit descriptions are provided below. Some status bits are sticky and are cleared after the read, and some are static and return the current status value. Sticky bits are indicated in the descriptions.

| | | |
|---|---|---|
| `SL514_CSTS_TXERR` | (1L<<19) | Slave TX Timing Error:<br>1= error detected, master clock arrived too early<br>(sticky bit, cleared after read) |
| `SL514_CSTS_RXERR` | (1L<<18) | Master RX Timing Error:<br>1= error detected, slave drove RX data high during monoflop time<br>(sticky bit, cleared after read) |
| `SL514_CSTS_TXFE` | (1L<<17) | Slave TX FIFO is empty:<br>1= FIFO empty<br>(sticky bit, cleared after read) |
| `SL514_CSTS_RXFF` | (1L<<16) | Master RX FIFO is full:<br>1= FIFO full<br>(sticky bit, cleared after read) |
| `SL514_CSTS_TXBSY` | (1L<<3) | Slave TX is transmitting data:<br>1= sending data |
| `SL514_CSTS_RXBSY` | (1L<<2) | Master RX is receiving data:<br>1= receiving data |
| `SL514_CSTS_TXFHF` | (1L<<1) | Slave TX is below watermark:<br>1= below watermark |
| `SL514_CSTS_RXFHF` | (1L<<0) | Master RX is above watermark:<br>1= above watermark |

### 4.39.4 DqAdv514SetPll

**Syntax:**

```
int DqAdv514SetPll(int hd, int devn, int chan, float baudrate, float
*actual_baud)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chan` | Channel to configure PLL for use with |
| `float baudrate` | Desired baud rate |
| `float *actual_baud` | Actual baud rate |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating memory buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-514 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets up a channel to use the on-board PLL to generate the master output clock baud rate instead of dividing down the on-board system clock (66 MHz). Consider using the PLL instead of the system clock for baud rates greater than 1Mbaud.

**Note:**

1. To use this function, you must use `L514CFG_CLK_PLL` as the `clock_source` configuration parameter using the `DqAdv514Config()` API.
2. To set the baud rate, the user-defined `baudrate` value is used to program the PLL. The PLL dividers may not be able to produce the exact programmed rate; in which case, the value closest to the user-programmed rate is used. Users can check the `actual_baud` parameter to know what baud rate is used.
3. When the `clock_source` is the system clock, the baud rate is programmed via the `baud_rate` parameter using the `DqAdv514Config()` API: this `DqAdv514SetPll` function is not used in that case.

### *4.39.5 DqAdv514WriteFIFO*

**Syntax:**

```
int DqAdv514WriteFIFO(int hd, int devn, int chan, int flags, int size,
uint32* buffer, int *written, int *available)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chan` | Channel to write TX FIFO |
| `int flags` | <Reserved> |
| `int size` | Size of the transmit buffer |
| `uint32* buffer` | Pointer to the array of transmit data |
| `int* written` | Number of data words stored in TX FIFO |
| `int* available` | Amount of free space in TX FIFO |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating memory buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-514 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Writes up to 1024 data words to the TX FIFO on the slave port of specified channel.

**Note:**

### *4.39.6 DqAdv514ReadFIFO*

**Syntax:**

```
int DqAdv514ReadFIFO(int hd, int devn, int chan, int flags, int size,
uint32* buffer, int *retrieved, int *available)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chan` | Channel to read RX FIFO |
| `int flags` | <Reserved> |
| `int size` | Maximum receive buffer size |
| `uint32* buffer` | Pointer to the array for receive data |
| `int* retrieved` | Number of data words retrieved from the RX FIFO |
| `int* available` | Amount of free space in RX FIFO |
| `uint32* status` | Status of channel |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating memory buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a SL-514 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Reads data received from the RX FIFO on the SSI master port of the specified channel. Each master port on the SL-514 has a 2048 FIFO for holding received data and for optionally holding timestamps associated with each piece of data.

**Note:**

- `<size>` limits the number of samples that will be copied into the receive buffer. The maximum allowed value is `DQ_PL_601_LISTSZ (350),` which is the maximum number allowed in a packet.
- `<status>` is returned and represents the status of the one channel programmed with this function (`chan`). Refer to the `DqAdv514Status()`API for bit descriptions.
- Alternatively, the `DqAdv514Status()` API reads status conditions from all channels listed in a bitmask of channels.

## 4.40 DNx-I2C-534 boards

### 4.40.1 DqAdv534SetConfig

**Syntax:**

```
int DqAdv534SetConfig(int hd, int devn, int port,
pI2C534CFG pCfg)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int port` | Port number |
| `pI2C534CFG pCfg` | Pointer to the device configuration structure |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a I2C-534 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_CRC_CHECK_FAILED` | received/transmitted parameters failed CRC secure shell check |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Set up parameters for master and slave configuration.

Configuration is set in a structure of type `I2C534CFG`:

```
// For L534_SET_CFG <flags>
typedef struct {    // all members must be of uint32 size
    uint32 flags;       // select active parameters to set/change
    uint32 clock;       // clock frequency, 100k, 400k and 1Mbit are supported; 0 = custom parameters
    float ttl_level;    // set line voltage (3.3V and 5.1V for now)
    uint32 tx_lines;    // enable termination and loopback
    MCTPARAM mctprm;    // custom timing parameters

    // Master configuration
    uint32 master_cfg;              // master configuration bitset
    uint32 master_idle_delay;       // delay in MM mode before acquiring bus
                                     // in 15ns increments
    uint32 master_byte_delay;       // delay between bytes sent by master in 1us
                                     // resolution
    uint32 master_max_sync_delay;   // maximum delay in uS slave could delay clock
    uint32 master_datasz_unfifo;    // <reserved>
    uint32 master_to_cfg;           // Maximum timeout delay in uS before releasing
                                     // bus (0 == default)
    uint32 master_wait_bm_fifo_ms;  // Maximum wait for BM FIFO to receive all
                                     // expected words, ms
    uint32 master_xdcp_device_type; // XDCP protocol device type - select upper 4 bits

    // Slave configuration
    uint32 slave_cfg;               // slave configuration bitset
    uint32 slave_addr;              // select 7/10 slave address and 10-bit address
                                      // mode
    uint32 slave_data;              // data to reply when slave Tx FIFO is empty
    uint32 slave_sync_dly;          // lenght of ACK in 15.15ns clocks (acknowledge
                                      // cycle stretch)
    uint32 slave_ack_dly;           // 12-bit how long we wait for master ACK (in clocks)
    uint32 slave_max_ack;           // Slave RX max count register (# of words per /ACK)
    uint32 slave_tx_reg_size;       // in unFIFO mode the size of bytes to transmit to
                                      // master, 1..4

} I2C534CFG, *pI2C534CFG;
```

`I2C534CFG` members configure the following:

**Port configuration:**

**`<flags>`**: Controls which set of configuration parameters can be changed.

Supported values include the following:

| | |
|---|---|
| `DQ_L534CFG_CLOCK`: | Allows baud rate to be reconfigured |
| `DQ_L534CFG_TTL_LEVEL`: | Allows voltage levels to be reconfigured |
| `DQ_L534CFG_MASTER_VALID`: | Allows master configuration to be reconfigured |
| `DQ_L534CFG_SLAVE_VALID`: | Allows slave configuration to be reconfigured |
| `DQ_L534CFG_TERM_LOOP`: | Allows termination and loopback to be reconfigured |
| `DQ_L534CFG_SDATA_ADDR`: | Allows slave address and register-based data to be reconfigured |
| `DQ_L534CFG_CLEAR` | Clears configurations stored from the previous calls |

**`<clock>`**: Controls baud rate, per-port.
To change `clocks`, `flags` must have `DQ_L534CFG_CLOCK` OR'ed in.

Supported values include the following:

| | |
|---|---|
| `DQ_L534CFG_CLOCK_100k` | Sets clock to 100 kHz |
| `DQ_L534CFG_CLOCK_400k` | Sets clock to 400 kHz (default) |
| `DQ_L534CFG_CLOCK_1M` | Sets clock to 1 MHz |
| `DQ_L534CFG_CLOCK_CUST` | Custom clock settings 2kHz…100kHz |

Note that both the slave and master of a port must run at the same rate; however, each port may run at a different rate.

**`<ttl_level>`**: Controls logic levels for the SDA and SCL pins, per-port.
To change `ttl_level`, `flags` must have `DQ_L534CFG_TTL_LEVEL` OR'ed in.

Supported values include the following:

| | |
|---|---|
| `DQ_L534CFG_TTL_LEVEL_3_3V` | Sets TTL levels to 3.3 V (default) |
| `DQ_L534CFG_TTL_LEVEL_5V` | Sets TTL levels to 5.0 V |

Note that both the slave and master of a port must use the same TTL level.

**`<tx_lines>`**: Controls muxing in termination resistors and configures master->slave loopback modes.
To change `tx_lines, flags` must have `DQ_L534CFG_TERM_LOOP` OR'ed in.

Supported values include the following:

| | |
|---|---|
| `DQ_L534CFG_RL_TERMIN` | Enables optional on-board termination.<br>Set to 0 to use default 1.5 KΩ pull-up resistor to the DC/DC output. |
| `DQ_L534CFG_RL_LOOPBK` | Enables hardware loopback. Hardware loopback using an onboard relay to create a master->slave connection after the port isolation before the DB connector.<br>**RL loopback** allows to read back the output on the I2C physical bus (required for Bus Monitor mode). |
| `DQ_L534CFG_FPGA_LOOPBK` | Enables loopback inside the FPGA.<br>**FPGA loopback** allows a user to test the master of each port without affecting the connected I2C physical bus. |

Note that both the slave and master of a port must use the same `tx_lines` programming.

<mctprm> - custom clock frequency settings. This structure should be programmed using DqAdv534CalcCustomTiming() and can be programmed in the range from 2kHz to 100kHz.

**Master-specific configuration:**

**`<master_cfg>`**: Allows users to configure master-specific configuration options.
To change `master_cfg, flags` must have `DQ_L534CFG_MASTER_VALID` OR'ed in.

Supported values include the following:

| | |
|---|---|
| `DQ_L534MCFG_SECURE_SHELL` | Enables secure shell operations. |
| `DQ_L534MCFG_MMASTER` | Enables multi-master mode. |
| `DQ_L534MCFG_CLK_SYNC` | Allows clock synchronization (stretching) by slave. The maximum delay that a slave can delay the clock is set to the value programmed into the `master_max_sync_delay` configuration parameter. |
| `DQ_L534MCFG_BYTE_DELAY` | Enables byte to byte delay (extra delay after ACK).<br>When `DQ_L534MCFG_BYTE_DELAY` is ORed into `master_cfg`, the master inserts a delay between bytes of whatever is programmed into the `master_byte_delay` configuration parameter. |
| `DQ_L534MCFG_LONG_IDLE_DLY` | Configures a programmed wait time before taking over the bus while in multi-master mode.<br>When `DQ_L534MCFG_MMASTER` and |

| | |
|---|---|
| | `DQ_L534MCFG_LONG_IDLE_DLY` are ORed into `master_cfg`, the master waits for the delay that is programmed into the `master_idle_delay` configuration parameter before acquiring the bus. |
| `DQ_L534MCFG_FIFO_EN` | Uses device-wide FIFO to write data to all ports (The port is selected in data word. Timestamps are available in this mode exclusively). |
| `DQ_L534MCFG_CRC_CHECK` | Forces CRC check in non-`DQ_L534MCFG_SECURE_SHELL` configuration. |
| `DQ_L534MCFG_XDCP` | Sets XDCP device ID. |
| `DQ_L534MCFG_RAWMODE` | Switch Master into raw mode operation |

**`<master_idle_delay>`**: Sets delay in multi-master mode before acquiring bus.
Programmed in 15 ns increments. This value is 32-bit.
To configure this delay, users must OR both
`DQ_L534MCFG_MMASTER | DQ_L534MCFG_LONG_IDLE_DLY` into `master_cfg`.

**`<master_byte_delay>`**: Sets delay between bytes sent by master.
Programmed in 1µs resolution. The value is 16 bit. Default value is no delay.
To configure this delay, users must OR `DQ_L534MCFG_BYTE_DELAY` into `master_cfg`.

**`<master_to_cfg>`**: Sets maximum timeout delay before releasing bus (0 == default).
Programmed in 1µs resolution. Default value is 500us. This value is 16-bit. If <master_to_cfg> is not zero, <master_max_sync_delay> needs to be set as well.

**`<master_max_sync_delay>`**: Sets maximum time slave could delay clock.
Programmed in 1µs resolution. This is 16-bit field. Default value is 500us.
To configure this delay, users must OR `DQ_L534MCFG_CLK_SYNC` into `master_cfg`,

**`<master_wait_bm_fifo_ms>`**: Sets maximum wait for BM FIFO to receive all expected words. Programmed in ms. Default value is 50ms. User needs to set up appropriate timeout value for the function to reply if any of the CRC-enabled functions wait for the transaction to be completed on the bus (for example, when slave inserts a delay between words by holding down a line).

**`<master_xdcp_device_type>`**: Sets XDCP protocol device type - select upper 4 bits. For example some Renesas' devices require type 0xA and Maxim's 0x5.

**Slave-specific configuration:**

**`<slave_cfg>`**: Allows users to configure slave-specific configuration options.
To change `slave_cfg, flags` must have `DQ_L534CFG_SLAVE_VALID` OR'ed in.

Supported values include the following:

| | |
|---|---|
| `DQ_L534SCFG_ENABLE_BM` | Sets slave into Bus Monitor (BM) mode.<br>Note that the slave is automatically set into BM mode when in secure shell master mode (when `DQ_L534MCFG_SECURE_SHELL` is ORed into |

| | |
|---|---|
| | `master_cfg`). |
| `DQ_L534SCFG_SCKSYN_AD` | Allows clock stretching during address ACK cycle The delay that the clock will be stretched is programmed as a 12-bit number of clocks in the `slave_sync_dly` configuration parameter. |
| `DQ_L534SCFG_SCKSYN_TX` | Allows clock stretching during a data TX ACK cycle The delay that the clock will be stretched is programmed as a 12-bit number of clocks in the `slave_sync_dly` configuration parameter. |
| `DQ_L534SCFG_SCKSYN_RX` | Allows clock stretching during a data RX ACK cycle The delay that the clock will be stretched is programmed as a 12-bit number of clocks in the `slave_sync_dly` configuration parameter. |
| `DQ_L534SCFG_ACKLEN` | Allows a value to be programmed into `slave_ack_dly` configuration parameter. Sets how long the slave will wait for master clock during an ACK. |
| `DQ_L534SCFG_ACK_BM` | Allows acknowledge (ACK) generation in BM mode |
| `DQ_L534SCFG_10BIT` | Configures the slave address as a 10-bit value (instead of the 7-bit default). |
| `DQ_L534SCFG_MAXACK` | Allows `slave_max_ack` number of words per /ACK. |
| `DQ_L534SCFG_SLAVE_UNFIFO` | Configures the slave to use a 32-bit TX register for transmit data (4 8-bit words) instead of the TX slave FIFO mode. |
| `DQ_L534SCFG_FIFO_SRXDO` | Suppress non-data bus conditions in RX FIFO.<br>When the slave stores received RX words, it also stores a 4-bit encoding of the bus condition that correspond with the transfer.<br>If this bit is set, only data with condition 6, 7, or 8 will be stored.<br>Bus conditions are described below:<br>`0 RSV:             Reserved`<br>`1 START:           I2C Start condition`<br>`2 RESTART:         I2C Restart condition`<br>`3 STOP:            I2C Stop condition`<br>`4 Address + ACK:   I2C Address`<br>`                   with ACK received`<br>`5 Address + /ACK:  I2C Address`<br>`                   with NACK received`<br>`6 Data + ACK:      I2C Data`<br>`                   with ACK received`<br>`7 Data + /ACK:     I2C Data`<br>`                   with NACK received`<br>`8 All data received + /ACK:`<br>`      I2C Last data byte + NACK received`<br>`9 Clock stretching error:`<br>`      I2C Error occurred in relation to`<br>`      clock stretching` |

**`<slave_addr>`**: Sets the 7/10-bit slave address.
To configure 10-bit addresses, users must OR `DQ_L534SCFG_10BIT` into `slave_cfg`.

**`<slave_data>`**: Sets data to reply when slave TX FIFO is empty.
Additionally, this is the data that gets sent when `DQ_L534SCFG_SLAVE_UNFIFO` is ORed into `slave_cfg` (configuring the bypass of the TX FIFO).

**`<slave_sync_dly>`**: Sets the length slave will delay clock during ACK (when acknowledge cycle stretch is configured).
Programmed in number of 15.15 ns clocks. This is a 12-bit field. The default value is 0x800
To use this delay, users must OR at least one of `DQ_L534SCFG_SCKSYN_AD`, `DQ_L534SCFG_SCKSYN_RX`, or `DQ_L534SCFG_SCKSYN_TX` into `slave_cfg`.

**`<slave_ack_delay>`**: Sets how long the slave will wait for master clock during an ACK.
Programmed in number of 15.15 ns clocks. This is a 12-bit field. The default value is 0x800
To configure this, users must OR `DQ_L534SCFG_ACKLEN` into `slave_cfg`.

**`<slave_max_ack>`**: Sets the slave RX max count register, which programs the number of words per /ACK.

**`<slave_tx_reg_size>`**: Sets the number of bytes to transmit to the master when the slave is configured with the `DQ_L534SCFG_SLAVE_UNFIFO` bit set in `slave_cfg`.
This value can be between 1 and 4.

**Note:**

Function envelopes parameters into CRC shell.

### *4.40.2 DqAdv534Flush*

**Syntax:**

```
int DqAdv534Flush(int hd, int devn, int port_mask,
int flags)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int port_mask` | Port bitmask that the flush operation is applied to |
| `int flags` | Defines operation |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not I2C-534 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_DEVICE_BUSY` | device is in use |
| `DQ_DATA_ERROR` | layer returned invalid data |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Flushes data from the selected FIFOs.

This function flushes receive and/or transmit FIFOs on slave and/or master depending on the <`flags`> for the selected bitmask of ports.

The following are supported `flags`:

| | |
|---|---|
| `DQL_IOCTL534_FLUSH_SRX`: | Flush slave RX FIFO |
| `DQL_IOCTL534_FLUSH_STX`: | Flush slave TX FIFO |
| `DQL_IOCTL534_FLUSH_MRX`: | Flush master RX FIFO |
| `DQL_IOCTL534_FLUSH_MTX`: | Flush master TX FIFO |
| `DQL_IOCTL534_FLUSH_MRST`: | Reset master module |
| `DQL_IOCTL534_FLUSH_SRST`: | Reset slave module |
| `DQL_IOCTL534_FLUSH_ALL`: | Flush all |

**Note:**

Use port mask. Port position with bit set to "1" enables port, "0" disables it.

### *4.40.3 DqAdv534Enable*

**Syntax:**

```
int DqAdv534Enable(int hd, int devn, int port_mask,
int flags)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int port_mask` | Port bitmask to enable |
| `int flags` | <reserved> |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not I2C-534 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_DEVICE_BUSY` | device is in use |
| `DQ_DATA_ERROR` | layer returned invalid data |
| `DQ_CRC_CHECK_FAILED` | received/transmitted parameters failed CRC secure shell check |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Enables or disables ports (configuration should be set prior to calling the enable using `DqAdv534SetConfig`).

**Note:**

Use port mask. Port position with bit set to "1" enables port, "0 disables it.
The stored configuration established when `DqAdv534SetConfig()` is called

Once master is enabled it asserts stop condition on the bus

## *4.40.4 DqAdv534GetStatus*

**Syntax:**

```
int DqAdv534GetStatus(int hd, int devn, int flags,
int port_mask, pI2C534STS pSts)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int flags` | <reserved> |
| `int port_mask` | Port mask to get status for |
| `pI2C534STS pSts` | Pointer to the array of I2C534STS to store status information |

**Output:**

| | |
|---|---|
| `pI2C534STS pSts` | Array of I2C534STS to store status information (allocated by the calling program) |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a I2C-534 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function requests the error and status of the interface.

```
typedef struct {
    uint32 hw_stat;     // HW status as reported by the hardware
    uint32 pin_stat;    // <reserved>
    uint32 drv_stat;    // <reserved>
} I2C534STS, *pI2C534STS;
```

**Notes:**

The `hw_status` reports the status of the I2C master and slave module. "Sticky" bits are cleared after each read.

The bitmapping for `hw_status` is as follows:

```
// Sticky bits 31..16
#define SL534_CH_STS_ACKERR   (1L<<30)   // =1 - master ack error has occurred
#define SL534_CH_STS_PIERR    (1L<<29)   // =1 - PHY idle wait timeout error
#define SL534_CH_STS_PTOERR   (1L<<28)   // =1 - PHY command wait while bus is active timeout error

#define SL534_CH_STS_CSERR    (1L<<27)   // =1 - slave clock synchronization error
#define SL534_CH_STS_MDONE    (1L<<26)   // =1 - master completes execution of the last command
#define SL534_CH_STS_MSCH     (1L<<25)   // =1 - master command or data sequence received
```

```
#define SL534_CH_STS_MCRCERR (1L<<24)   // =1 - error in CLO-->master CRC (data/command ignored)

#define SL534_CH_STS_MCLOERR (1L<<23)   // =1 - error in CLO-->master sequence
#define SL534_CH_STS_MMERR   (1L<<22)   // =1 - multi-master arbitration was lost
#define SL534_CH_STS_STXDONE (1L<<21)   // =1 - slave TX data sent (non-continuous mode)
#define SL534_CH_STS_SRXDTR  (1L<<20)   // =1 - slave RX data ready (non-continuous mode)

#define SL534_CH_STS_SRXFFS  (1L<<19)   // =1 - slave RX FIFO was full
#define SL534_CH_STS_STXFES  (1L<<18)   // =1 - slave TX FIFO was empty
#define SL534_CH_STS_RXFFS   (1L<<17)   // =1 - master RX FIFO was full
#define SL534_CH_STS_TXFES   (1L<<16)   // =1 - master TX FIFO was empty

    // Static bits 15..0
#define SL534_CH_STS_BBSY    (1L<<15)   // =1 when bus is busy
#define SL534_CH_STS_CSWAIT  (1L<<14)   // =1 indicates that master waits for the slave to release SCL
#define SL534_CH_STS_MCLORDY (1L<<13)   // =1 when port can accept writes to SL534_CH_CLOWR
#define SL534_CH_STS_MWAIT   (1L<<12)   // =1 - master waiting for command/data release (write to the
SL534_CH_RBGO)

#define SL534_CH_STS_CLOMWT  (1L<<11)   // =1 when output port list SM waits for the master to become
available
#define SL534_CH_STS_SRXFHF  (1L<<10)   // =1 - slave RX FIFO is above watermark
#define SL534_CH_STS_SRXFF   (1L<<9)    // =1 - slave RX FIFO is full
#define SL534_CH_STS_STXFHF  (1L<<8)    // =1 - slave TX FIFO is below watermark

#define SL534_CH_STS_STXFE   (1L<<7)    // =1 - slave TX FIFO is empty
#define SL534_CH_STS_SBSY    (1L<<6)    // =1 when slave state machine is busy
#define SL534_CH_STS_EMAS    (1L<<5)    // =1 when external master owns the bus
#define SL534_CH_STS_MBSY    (1L<<4)    // =1 when master state machine is busy

#define SL534_CH_STS_RXFHF   (1L<<3)    // =1 - master RX FIFO is above watermark
#define SL534_CH_STS_RXFF    (1L<<2)    // =1 - master RX FIFO is full
#define SL534_CH_STS_TXFHF   (1L<<1)    // =1 - master TX FIFO is below watermark
#define SL534_CH_STS_TXFE    (1L<<0)    // =1 - master TX FIFO is empty
```

### 4.40.5 DqAdv534BusControl

**Syntax:**

```
int DqAdv534BusControl(int hd, int devn, int port, uint32 flags,
uint32* parameters);
```

**Command:**

IOCTL

**Input:**

int hd -- Handle to the IOM
int devn -- Device number
int port -- port to control
uint32 flags -- defines operation
uint32* parameters -- array of parameters for the operation defined in flags

**Output:**

None

**Returns:**

DQ_NO_MEMORY - error allocating buffer
DQ_ILLEGAL_HANDLE - illegal IOM Descriptor or communication wasn't established

DQ_BAD_DEVN - device indicated by devn does not exist or is not a I2C-534
DQ_SEND_ERROR - unable to send the Command to IOM
DQ_TIMEOUT_ERROR - nothing is heard from the IOM for Time out duration
DQ_IOM_ERROR - error occurred at the IOM when performing this command
DQ_SUCCESS - successful completion
Other negative values - low level IOM error

**Description:**

This function controls some features of I2C port while the board is configured and application is running. Supported control features include turning on and off DC/DC per each port, selecting loopback mode and additional termination resistors

<flags> define the feature to control (one flag can be used per function call). The <parameters> used depend on the <flags> parameter, explained in the table below:

| <flags> | <parameters> |
|---|---|
| DQL_IOCTL534_BUSCON_DCDC | parameters[0] = 1; // to enable DC/DC for the port<br>parameters[0] = 0;  // to disable DC/DC for the port |
| DQL_IOCTL534_BUSCON_LBENTERM | parameters[0] = DQ_L534_BUSCON_LBEN; // enable select local // loopback relay<br>Parameters[0] = DQ_L534_BUSCON_TERMEN; // enable // additional 1.5kOhm pull-up (in parallel with 4.99kOhm) |

**Notes:**

The size of <parameters> array depends on the operation to perform (i.e. the value of <flags> parameter)

### *4.40.6 DqAdv534MasterSendCommandCRC*

**Syntax:**

```
int DqAdv534MasterSendCommandCRC(int hd, int devn, int port, uint32
flags, uint32 command, int wrrd_rx_size, int size,
char* data, int* transmitted, int* available, uint32* crc_stat)
```

**Command:**

Encoded in `command`

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM`() |
| `int devn` | Layer inside the IOM |
| `int port` | Port number to control |
| `uint32 flags` | Defines operation |
| `uint32 command` | address and I2C command |
| `int wrrd_rx_size` | Number of bytes to read for WRRD command |
| `int size` | Number of data words to send for WRITE and WRRD command or data words to receive for READ command |
| `char* data` | Data words to send |

**Output:**

| | |
|---|---|
| `int* transmitted` | Number of bytes accepted in the FIFO (I2C is a byte-oriented protocol) |
| `int* available` | Number of bytes still available to write in the transmission FIFO |
| `uint32* crc_stat` | Status flags for the port |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a I2C-534 |
| `DQ_CRC_CHECK_FAILED` | received/transmitted parameters failed CRC secure shell check |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| `DQ_CRC_CHECK_FAILED` | CRC check has failed while verifying command CRC in IOM |
| `DQ_DEVICE_BUSY` | Device is busy with the previous operation, cannot transmit or receive data now |
| `DQ_DATA_ERROR` | Data error - CRC check failed while verifying received CRC in the library |
| Other negative values | low level IOM error |

**Description:**

Sends master command enveloped in CRC safety shell.

This function sends the master data in a secure fashion. If the FPGA CRC check failed, the data is discarded. You can program this function to wait for all BM data associated with the transmission and return only after the transaction is completed on the bus.

<**`command`**>: Defines I2C master command on the bus and the address of the slave the message is directed to. Select one of the following commands:

| | |
|---|---|
| `DQ_L534_CMD_TDELAY` | Insert NOP command for delay sequence |
| `DQ_L534_CMD_STOP` | STOP - issue a stop condition once bus is available |
| `DQ_L534_CMD_ST_WRITE` | START+WRITE (including write multiple) |
| `DQ_L534_CMD_ST_READ` | START+READ (including read multiple) |
| `DQ_L534_CMD_ST_WRRD` | START+WRITE+RESTART+READ (including read multiple) |
| `DQ_L534_CMD_XDCP_READ` | START+WRITE+READ (for Renesas XDCP protocol) |

OR the command with behavior modifiers:

| | |
|---|---|
| `DQ_L534_CMD_A_BIT` | Address mode bit: 0 == 7-bit/1 == 10-bit |
| `DQ_L534_CMD_N_BIT` | Address ignore NACK bit.<br>1 == ignore /ACK (I2C violation) |
| `DQ_L534_CMD_P_BIT` | NO-STOP (do not issue STOP) |

OR the command with the address of the slave:

| | |
|---|---|
| `DQ_L534_CMD_ADDR7(N)` | 7 bit address |
| `DQ_L534_CMD_ADDR10(N)` | 10 bit address |

A typical command word looks like:

```
command =
        //DQ_L534_CMD_ST_WRITE|     // either _READ or _WRITE command should be commented out
           DQ_L534_CMD_ST_READ|
           ((if10bit)?DQ_L534_CMD_ADDR10(slave_addr):DQ_L534_CMD_ADDR7(slave_addr));
```

<**`flags`**>: This API supports the following configuration flags:

| | |
|---|---|
| `DQ_L534_MSENDCRC_WAIT_FOR_BM`: | wait until bus monitor receives full transmission |
| `DQ_L534_MSENDCRC_DOUBLECHECK_CRC`: | double-check CRC, generally this flag is not required and increases execution time |
| `DQ_L534_MSENDCRC_WAIT_FOR_RX`: | used with RX command to wait for slave to transmit data |

**Note:**

To retrieved data received on RX command use `DqAdv534MasterReceiveDataCRC()`.
Refer to `DqAdv534GetStatus()` for the definition of bit states for `crc_stat`.

This function has an additional behavior to wait for receive data using DQ_L534_MSENDCRC_WAIT_FOR_BM or DQ_L534_MSENDCRC_WAIT_FOR_RX flags. The first one calculates how many words BM needs to receive and waits for them or timeout (50 ms by default or set in configuration structure as < master_wait_bm_fifo_ms> member. After that DqAdv534ReadBMFIFOCRC() delivers all transaction data. The second one does the same for DqAdv534MasterReceiveDataCRC().

Please notice that DqAdv534ReadBMFIFOCRC() reads slave receive FIFO when slave is set into BM mode (all conditions and bytes on the bus). DqAdv534MasterReceiveDataCRC() reads master receive FIFO (received bytes only).

If you send receive command to multiple ports using DqAdv534MasterSendCommandCRC() call you may not need to set up DQ_L534_MSENDCRC_WAIT_FOR_BM flag except for the last call (or use a known delay) to receive all data in subsequent calls to DqAdv534ReadBMFIFOCRC(). This will slash time required to call DqAdv534MasterSendCommandCRC() for multiple ports.

### *4.40.7 DqAdv534MasterSendNChanCRC*

**Syntax:**

```
int DqAdv534MasterSendNChanCRC(int hd, int devn, int port_mask,
uint32 flags, pI2C534_CMD pCmd)
```

**Command:**

Encoded in `command`

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM`() |
| `int devn` | Layer inside the IOM |
| `int port_mask` | Port mask for the ports to send data |
| `uint32 flags` | Defines operation |
| `pI2C534_CMD pCmd` | A pointer to the structure that defines I2C address and command |

**Output:**

| | |
|---|---|
| `pI2C534_CMD pCmd` | Operation status for each port |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a I2C-534 |
| `DQ_CRC_CHECK_FAILED` | received/transmitted parameters failed CRC secure shell check |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| `DQ_CRC_CHECK_FAILED` | CRC check has failed while verifying command CRC in IOM |
| `DQ_DEVICE_BUSY` | Device is busy with the previous operation, cannot transmit or receive data now |
| `DQ_DATA_ERROR` | Data error - CRC check failed while verifying received CRC in the library |
| Other negative values | low level IOM error |

**Description:**

Sends master command to a set of ports simultaneously. Master commands are enveloped in CRC safety shell.

This function sends the master data in a secure fashion. If the FPGA CRC check failed, the data is discarded. You can program this function to wait for all BM data associated with the transmission and return only after the transaction is completed on the bus.

This function is similar to `DqAdv534MasterSendCommandCRC()` but passes command to each port defined in <port_mask> in an array of `I2C534_CMD` structures.

```
typedef struct {    // all members must be of uint32 size
    uint32 command;         // address and I2C command
    int data_size;          // number of data words to send for WRITE and WRRD command
                           // or data words to receive for READ command
    int wrrd_rx_size;       // number of bytes to read for WRRD command
    int fifo_sz;            // filled automaticallu - set to zero - number of data
                           // words to be written to the internal FIFO
    int transmitted;        // number of bytes accepted in the FIFO (I2C is a
                            // byte-oriented protocol)
    int available;          // number of bytes still available to write in the
                            // transmission FIFO
    int crc_stat;           // status flags of operation
    char* data8;            // pointer to the data words to send (or retrieved
                            // data on RX command)
} I2C534_CMD, *pI2C534_CMD;
```

Members of the `I2C534_CMD` struct like <transmitted>, <available> and <crc_stat> are returned from the IOM after completing the call.

**Notes:**

The function checks CRC and program operations for each port included in <port_mask>. So, if the port mask is 1010b (i.e. ports 3 and 1) caller needs to pass `I2C534_CMD` array consistent of two elements – [0] for port 1 and [1] for port 3.

The function first programs all ports included into operation and then triggers command on each valid port at the same time minimizing shift between ports.

There are certain flags that can be used with this function:

DQ_L534_MSENDCRC_WAIT_FOR_BM -- wait until bus monitor receives full transmission
DQ_L534_MSENDCRC_WAIT_FOR_RX -- with RX command wait for slave to transmit data.

If the flag is set the function will wait for up to the default 50ms for all ports to receive data.
This default value can be reprogrammed between 0 and L534_MAX_WAIT_TO_CLOWR_mS (50ms) by setting <master_wait_bm_fifo_ms> in the port configuration.

### *4.40.8 DqAdv534MasterReceiveDataCRC*

**Syntax:**

```
int DqAdv534MasterReceiveDataCRC(int hd, int devn, int port, uint32
flags, int size, uint16* data,
int* received, int* available, uint32* crc_stat)
```

**Command:**

Encoded in `command`

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM`() |
| `int devn` | Layer inside the IOM |
| `int port` | Port number to read from |
| `uint32 flags` | Defines operation |
| `int size` | Number of words to receive |
| `uint16* data` | Data retrieved on RX command (caller allocated) |

**Output:**

| | |
|---|---|
| `int* received` | Number of bytes received |

| | |
|---|---|
| `int* available` | Number of bytes still available to read in the FIFO |
| `uint32* crc_stat` | Status flags for the port |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a I2C-534 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_CRC_CHECK_FAILED` | received/transmitted parameters failed CRC secure shell check |
| `DQ_TIMEOUT_ERROR` | nothing heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| `DQ_CRC_CHECK_FAILED` | CRC check has failed while verifying command CRC in IOM |
| `DQ_DEVICE_BUSY` | Device is busy with the previous operation, cannot transmit or receive data now |
| `DQ_DATA_ERROR` | Data error - CRC check failed while verifying received CRC in the library |
| Other negative values | low level IOM error |

**Description:**

Receives data transmitted from slave to the master, enveloped in CRC.

**Note:**

CRC is calculated for function parameters before sending them to the IOM and then IOM checks CRC of received command and parameters before executing it. When replying, IOM calculates CRC of the data and the function in the library verifies it.

Refer to `DqAdv534GetStatus()` for the definition of bit states for `crc_stat.`
Set <flags> to zero – reserved for the future extension.

You can use either one of those functions depends on your requirements:

DqAdv534ReadBMFIFOCRC() is more verbose because it also includes conditions on the bus (the upper [11..8] bits).

DqAdv534MasterReceiveDataCRC() receives data only. For example, if BM receives data_in = {0x0100, 0x0441, 0x070A, 0x0300} master FIFO receives data_in = {0x0A}

### *4.40.9 DqAdv534ReadBMFIFOCRC*

**Syntax:**

```
int DqAdv534ReadBMFIFOCRC(int hd, int devn, int port,
uint32 flags, int size, uint16* data,
int* received, int* available, uint32* crc_stat)
```

**Command:**

Encoded in `command`

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM`() |
| `int devn` | Layer inside the IOM |
| `int port` | Port number to read from |
| `uint32 flags` | Defines operation |
| `int size` | Number of words to receive |
| `unit16* data` | Data retrieved on RX command |

**Output:**

| | |
|---|---|
| `int* received` | Number of bytes received |
| `int* available` | Number of bytes still available to retrieve from the FIFO |
| `uint32* crc_stat` | Port status flags |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a I2C-534 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_CRC_CHECK_FAILED` | received/transmitted parameters failed CRC secure shell check |
| `DQ_TIMEOUT_ERROR` | nothing heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| `DQ_CRC_CHECK_FAILED` | CRC check has failed while verifying command CRC in IOM |
| `DQ_DEVICE_BUSY` | Device is busy with the previous operation, cannot transmit or receive data now |
| `DQ_DATA_ERROR` | Data error - CRC check failed while verifying received CRC in the library |
| Other negative values | low level IOM error |

**Description:**

Retrieves data from bus monitor with CRC.

**Note:**

Refer to `DqAdv534GetStatus()` for the definition of bit states for `crc_stat.`
Set <flags> to zero – reserved for the future extension.

Values retrieved from this function call are defined in DNx-I2C-534 User Manual. Please notice that when master is enabled for the first time it asserts stop condition on the bus.

Bus conditions are described below (bits [11..8]):

```
0 RSV:                  Reserved
1 START:                I2C Start condition
2 RESTART:              I2C Restart condition
3 STOP:                 I2C Stop condition
4 Address + ACK:        I2C Address with ACK received
5 Address + /ACK:       I2C Address with NACK received
6 Data + ACK:           I2C Data with ACK received
```

```
7 Data + /ACK:          I2C Data with NACK received
8 All data received + /ACK:   I2C Last data byte + NACK received
9 Clock stretching error:     I2C Error occurred in relation toclock stretching
```

### *4.40.10 DqAdv534BuildCmdData*

**Syntax:**

```
int DqAdv534BuildCmdData(int hd, int devn, int port,
uint32 flags, uint32 command, int wrrd_rx_size, int size,
char* data, int* tx_size, uint32* tx_data)
```

**Command:**

Encoded in `command`

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM`() |
| `int devn` | Layer inside the IOM |
| `int port` | Port number to control |
| `uint32 flags` | Defines operation |
| `uint32 command` | Address and I2C command |
| `int wrrd_rx_size` | Number of bytes to read in WRRD command |
| `int size` | Number of data words to send for WRITE and WRRD command or data words to receive for READ command |
| `char* data` | Data words to send |

**Output:**

| | |
|---|---|
| `int* tx_size` | Actual size of the resulting array |
| `uint32* tx_data` | User-allocated array to store data<br>(must be at least of 4+`size`/2 words) |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a I2C-534 |
| `DQ_CRC_CHECK_FAILED` | received/transmitted parameters failed CRC secure shell check |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Builds master command enveloped in CRC safety shell.

This function configures the master command. If the command is a write, this API uses `data` and `size` to create the resulting command. The resulting command data is stored in `tx_data` and the size of the resulting command data in `tx_size`. Use `DqAdv534MasterWriteTxFIFO` to write the command data to the FIFO.

<**`command`**>: Defines I2C master command on the bus and the address of the slave the message is directed to. Select one of the following commands:

| | |
|---|---|
| `DQ_L534_CMD_A_BIT` | Address mode bit: 0 == 7-bit/1 == 10-bit |
| `DQ_L534_CMD_N_BIT` | Address ignore NACK bit.<br>1 == ignore /ACK (I2C violation) |
| `DQ_L534_CMD_P_BIT` | NO-STOP (do not issue STOP) |
| `DQ_L534_CMD_TDELAY` | Insert NOP command for delay sequence |
| `DQ_L534_CMD_STOP` | STOP - issue a stop condition once bus is available |
| `DQ_L534_CMD_ST_WRITE` | START+WRITE (including write multiple) |
| `DQ_L534_CMD_ST_READ:` | START+READ (including read multiple) |
| `DQ_L534_CMD_ST_WRRD:` | START+WRITE+RESTART+READ<br>(including read multiple) |
| `DQ_L534_CMD_XDCP_READ:` | START+WRITE+READ<br>(for Renesas XDCP protocol - ex. X9119) |
| `DQ_L534_CMD_XDCP_WRITE:` | START+WRITE+WRITE<br>(for Renesas XDCP protocol - ex. X9259) |

OR the command with the address of the slave:

| | |
|---|---|
| `DQ_L534_CMD_ADDR7(N)` | 7 bit address |
| `DQ_L534_CMD_ADDR10(N)` | 10 bit address |

<**`flags`**>: This API supports the following configuration flags:

| | |
|---|---|
| `DQ_L534_MSENDCRC_IGNORE_CRC` | `ignore CRC errors` |

**Note:**

### *4.40.11 DqAdv534MasterWriteTxFIFO*

**Syntax:**

```
int DqAdv534MasterWriteTxFIFO(int hd, int devn, int port,
uint32 flags, int size, uint32* data,
int* transmitted, int* available)
```

**Command:**

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM`() |
| `int devn` | Layer inside the IOM |
| `int port` | Port number to write to |
| `uint32 flags` | Behavior modifiers |
| `int size` | Number of uint32 words in <`data`> to write |
| `unit32* data` | Data array to write to the layer. The size of the command is limited to 255 words |

**Output:**

| | |
|---|---|
| `int* transmitted` | Number of words successfully written into transmit FIFO |
| `int* available` | space still available in the FIFO at the moment of return |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a I2C-534 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Writes to the master FIFO and returns number of words written and still available in the FIFO. To simplify usage `DqAdv534BuildCmdData()` creates an array of words to be written into Tx FIFO. `DqAdv534BuildCmdData()` stores an array of uint32 data to be written to the FIFO using this function.

**Note:**

Flags determine whether the write is intended for CLOWR (direct port Master FIFO).
If the write happens into CLOWR upper (port number) bits in uint32 words don't matter because CLOWR FIFO is separate for each port. CLO FIFO works across all ports.
If the write happens to be into CLO FIFO upper bits define port to write to.

### 4.40.12 DqAdv534MasterReadRxFIFO

**Syntax:**

```
int DqAdv534MasterReadRxFIFO(int hd, int devn, int port,
uint32 flags, int size, uint16* data,
int* received, int* available)
```

**Command:**

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM`() |
| `int devn` | Layer inside the IOM |
| `int port` | Port number to read from |
| `uint32 flags` | Behavior modifiers |
| `int size` | Number of uint16 words in <`data`> to read |
| `unit16* data` | Array of data read from the layer. The size of the command is limited to 255 words. A caller program allocates this array |

**Output:**

| | |
|---|---|
| `int* received` | Number of words successfully received from RX FIFO |
| `int* available` | Space still available in the FIFO at the moment of return |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a I2C-534 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Reads from the master Rx FIFO and returns number of words read and still available in the FIFO.

**Note:**

The values received from the master read FIFO contain values transmitted by slave upon read command. Notice that “STOP” bit is set at the end of each read command (1<<8). For example, a read command to read four bytes would output: 0x00dd 0x00dd 0x00dd 0x01dd, where “dd” is received byte of data.

### 4.40.13 DqAdv534SlaveWriteTxFIFO

**Syntax:**

```
int DqAdv534SlaveWriteTxFIFO(int hd, int devn, int port,
uint32 flags, int size, uint16* data,
int* transmitted, int* available)
```

**Command:**

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM`() |
| `int devn` | Layer inside the IOM |
| `int port` | Port number to write to |
| `uint32 flags` | Behavior modifiers |
| `int size` | Number of uint16 words in <`data`> to write |
| `uint16* data` | Data array to write to the layer. The size of the command is limited to 255 words |

**Output:**

| | |
|---|---|
| `int* transmitted` | Number of words successfully written into transmit FIFO |
| `int* available` | Number of words still available to write to the FIFO at the moment of return |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a I2C-534 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Writes to the slave FIFO and returns number of words written and still available in the FIFO.

**Note:**

This is the FIFO where data is taken to be transmitted when the slave receives an RX command from the master.

Slave implementation includes two modes:

1. Simple mode when for up to four bytes are written to the register. Example – in the configuration structure:

```
CfgSlv.slave_data = 0x40506070;    // something to send back since we don't fill slave Tx
FIFO
CfgSlv.slave_tx_reg_size = 4;      // number of bytes 1..4
CfgSlv.slave_cfg |= DQ_L534SCFG_SLAVE_UNFIFO; // Use <slave_data> instead of FIFO
```

Data in the slave register can be replaced anytime by calling DqAdv534SetConfig() with the flag DQ_L534CFG_SDATA_ADDR and fields <slave_data> and <slave_addr> set to the new values.

2. FIFO mode when a byte of data is taken from the FIFO each time a read command is on the bus (1L<<8) is the end of the data for read command

This is the default mode. User can optionally add `CfgSlv.flags |= DQ_L534CFG_SDATA_ADDR;` flag and fill <slave_data> field. In this case when FIFO becomes empty slave will start taking data from the register instead repeating last byte from the FIFO.

### 4.40.14 DqAdv534SlaveReadRxFIFO

**Syntax:**

```
int DqAdv534SlaveReadRxFIFO(int hd, int devn, int port,
uint32 flags, int size, uint16* data,
int* received, int* available)
```

**Command:**

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM`() |
| `int devn` | Layer inside the IOM |
| `int port` | Port number to read from |
| `uint32 flags` | Behavior modifiers |
| `int size` | Number of uint16 words in <`data`> to read |
| `unit16* data` | Array of data read from the layer. The size of the command is limited to 255 words |

**Output:**

| | |
|---|---|
| `int* received` | Number of words successfully received from RX FIFO |
| `int* available` | Number of words still available in the FIFO at the moment of return |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a I2C-534 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Reads from the slave FIFO and returns number of words read and still available in the FIFO.

**Notes:**

Slave RX FIFO contains data received from the master (including command and bus transitions). The lower [7..0] eight bits contain data or address byte and the top [11..8] bits contain bus condition:

```
// bits [11:8] in BM and slave RX FIFO data) data and bus conditions
#define L534_I2C_BM_CODE_START   1     // 1 - START
#define L534_I2C_BM_CODE_RESTART 2     // 2 - ReSTART
#define L534_I2C_BM_CODE_STOP    3     // 3 - STOP
```

```
#define L534_I2C_BM_CODE_AACK    4     // 4 - Address + ACK
#define L534_I2C_BM_CODE_ANACK   5     // 5 - Address + /ACK
#define L534_I2C_BM_CODE_DACK    6     // 6 - Data + ACK
#define L534_I2C_BM_CODE_DNACK   7     // 7 - Data + /ACK
#define L534_I2C_BM_CODE_DDONE   8     // 8 - All data received + /ACK
#define L534_I2C_BM_CODE_SCERR   9     // 9 - Clock stretching error
```

A typical received looks like:
0x100 (start)
0x4aa (address with acknowledge)
0x6dd (data with acknowledge)
0x8dd (last data word)
0x300 (stop condition)

During the configuration case user can add DQ_L534SCFG_FIFO_SRXDO bit into <slave_cfg> field to suppress storing bus conditions into the FIFO.

### 4.40.15 DqAdv534MasterWriteTxPhyFIFO

**Syntax:**

```
int DAQLIB DqAdv534MasterWriteTxPhyFIFO(int hd, int devn,
int port, uint32 flags, int size, uint16* data,
int* transmitted, int* available)
```

**Command:**

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM`() |
| `int devn` | Layer inside the IOM |
| `int port` | Port number |
| `uint32 flags` | Behavior modifiers |
| `int size` | Number of uint16 words in <`data`> to write |
| `uint16* data` | Data array to write to the layer. The size of the command is limited to 255 words |

**Output:**

| | |
|---|---|
| `int* transmitted` | Number of words successfully transmitted from FIFO |
| `int* available` | Number of words still available to write to the FIFO at the moment of return |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a I2C-534 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |

| | |
|---|---|
| Other negative values | low level IOM error |

**Description:**
Writes to the master Tx FIFO and returns number of words written and still available in the FIFO. This function is not capable of performing CRC-enabled operations.

**Notes:**

This is a low-level access to the Master, there is no control over what is transmitted.
Each byte is packed into uint16. TX FIFO should be pre-loaded before issuing an I2C write command. Bit 8 should be used to indicate last byte that should be transmitted on the bus with currently processed command. Multiple packets can be preloaded each one terminated with STOP bit (1L<<8).

This function is also used for "raw" operation access where host writes byte-by-byte instructions of what master phy should do (bits [11..8] are command and [7..0] data byte if applicable

In “raw” mode upper bits [11..8] define a command on the bus:

```
I2C_MRAW_PHY_TX1(B)      - PHY: Set SDA to 1 for one clock
I2C_MRAW_PHY_TX0(B)      - PHY: Set SDA to 0 for one clock
I2C_MRAW_PHY_RX(B)       - PHY: Set SDA to 1 for one clock, return SDA at the
falling edge of the clock
I2C_MRAW_PHY_START(B)    - PHY: START condition on the bus
I2C_MRAW_PHY_STOP(B)     - PHY: STOP condition on the bus
I2C_MRAW_PHY_RELEASE(B)  - PHY: Release bus without creating START or STOP
conditions
I2C_MRAW_DLY_2NUS(B)     - MASTER: Delay execution for 2^n uS (n is in 5 LSBs)
I2C_MRAW_DLY_NX8US(B)    - MASTER: Delay execution for n*8uS (n is in 8 LSBs)
I2C_MRAW_BYTE_SEND(B)    - MASTER: Transmit data to the I2C bus
I2C_MRAW_BYTE_RECEIVE(B) - MASTER: Read byte of data from the I2C bus and save to
the RX FIFO
I2C_MRAW_ACK_WAIT(B)     - MASTER: Wait for the ACK, NACK ends sequence
I2C_MRAW_SEQ_END(B)      - MASTER: Last command in the sequence, write starts
execution
```

For these macros pass zero as B if lower eight bits don’t need to be defined.

For example, a bus write command with two words can be programmed as follows:

```
I2C_MRAW_PHY_START(0);
I2C_MRAW_BYTE_SEND(slave_addr<<1);
I2C_MRAW_ACK_WAIT(0);
I2C_MRAW_BYTE_SEND(data8[0]);
I2C_MRAW_ACK_WAIT(0);
I2C_MRAW_BYTE_SEND(data8[1]);
I2C_MRAW_ACK_WAIT(0);
I2C_MRAW_PHY_STOP(0);
I2C_MRAW_SEQ_END(0);
```

A read counterpart can be programmed with the following sequence:

```
I2C_MRAW_PHY_START(0);
I2C_MRAW_BYTE_SEND((slave_addr<<1)|1);
```

```
I2C_MRAW_ACK_WAIT(0);
I2C_MRAW_BYTE_RECEIVE(0);
I2C_MRAW_PHY_TX0(0);
I2C_MRAW_BYTE_RECEIVE(0);
I2C_MRAW_PHY_TX1(0);
I2C_MRAW_PHY_STOP(0);
I2C_MRAW_SEQ_END(0);
```

Some devices require write-restart-read sequence which could be also defined with “raw” mode commands:

```
I2C_MRAW_PHY_START(0);      // start
I2C_MRAW_BYTE_SEND(0xA0);   // address + write
I2C_MRAW_ACK_WAIT(0);       // slave ACK
I2C_MRAW_BYTE_SEND(0xF0);   // register
I2C_MRAW_ACK_WAIT(0);       // slave ACK
I2C_MRAW_PHY_RELEASE(0);    // release + start = restart
I2C_MRAW_PHY_START(0);
I2C_MRAW_BYTE_SEND(0xA1);   // address + read
I2C_MRAW_ACK_WAIT(0);       // slave ACK
I2C_MRAW_BYTE_RECEIVE(0);   // receive
I2C_MRAW_PHY_TX1(0);        // master NACK
I2C_MRAW_PHY_STOP(0);       // stop
I2C_MRAW_SEQ_END(0);
```

Read command results can be retrieved from the master read FIFO using `DqAdv534MasterReadRxFIFO()` and if BM mode and loopback is enabled on that port `DqAdv534SlaveReadRxFIFO()` can be used to retrieve all transactions on the bus.

### *4.40.16 DqAdv534CalcCustomTiming*

**Syntax:**

```
int DAQLIB DqAdv534CalcCustomTiming(int hd, float divider, pMCTPARAM
pMct)
```

**Command:**

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM`() |
| `float divider` | Divider of 100kHz clock |

**Output:**

| | |
|---|---|
| `pMCTPARAM pMct` | Pointer to the caller allocated structure to store timing parameters |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a I2C-534 |

| | |
|---|---|
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

The function calculates timing parameters based on 100kHz base clock and the divider. As a result it fills out the following structure:

```
// custom timing parameters
typedef struct {
    uint32 mctidle;     // time interval before IDLE is detected
    uint32 mctstart;    // SDA low to SCL low for START condition
    uint32 mctstop;     // SCL high to SDA high for STOP condition
    uint32 mctdbnc;     // debounce time
    uint32 mctsudat;    // T su;dat
    uint32 mcthigh;     // T high
} MCTPARAM, *pMCTPARAM;
```

This structure can be passed as a parameter into `DqAdv534SetConfig()` as <mctprm> member of `pI2C534CFG pCfg` structure. To make use of custom timing field <clock> should be set to DQ_L534CFG_CLOCK_CUST.

**Note:**

Supported custom rates are from 2kHz to 100kHz

## 4.41 DNA-CAR-550 layer

### 4.41.1 DqAdvSetWirelessState

**Syntax:**

`DqAdvSetWirelessState(int hd, int devn, uint32 cmd, uint32 data)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 cmd` | what parameter to set |
| `uint32 data` | value depending on `cmd` parameter<br>if cmd is `DQ_CAR550_WIRELESS_EN_DIS`, `data` is 32-bit value to enable or disable wireless system. Zero to disable wireless, non-zero to enable. |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CAR-550 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function is used to enable or disable the wireless interface installed on a CAR-550.

**Note:**

None.

## 4.42 DNA-429-566/512/516 (ARINC-429) layers

DNx-429 layers provide the following transmit/receive configurations:

| | |
|---|---|
| • DNx-429-512: | 12 Receive channels |
| • DNx-429-566: | 6 Receive and 6 Transmit channels with dedicated hardware loopback on all TX channels |
| • DNx-429-516: | 16 Transmit channels with RX readback and TX disable |

### 4.42.1 DqAdv566BuildPacket/DqAdv566ParsePacket

**Syntax:**

```
uint32 DqAdv566BuildPacket(uint32 data, uint32 label, uint32 sdi, uint32
ssm, uint32 parity)

void DqAdv566ParsePacket(uint32 packet, uint32* data, uint8* label, uint8*
sdi, uint8* ssm, uint8* parity)
```

**Command:**

None

**Input:**

| | |
|---|---|
| `uint32 data` | Data (19 bit) |
| `uint32 label` | Label (8 bit) |
| `uint32 sdi` | SDI field (2 bit) |
| `uint32 ssm` | SSM field (2 bit) |
| `uint32 parity` | Parity (1 bit) |

**Output:**

Assembled packet converted to ARINC 429 Holt3282 bit order (DqAdv566BuildPacket)
Fields of parsed received packet (DqAdv566ParsePacket)

**Description:**

These paired functions assemble an ARINC message out of individual fields for transmission and split a received packet into individual fields. The function takes care of converting normal ARINC 429 data into the representation required by Holt3282 chip on DNA-429-566 layer.

**Note:**

None.

### *4.42.2 DqA429toHolt3282/ DqHolt3282toA429*

**Syntax:**

```
uint32 DAQLIB DqA429toHolt3282 (uint32 hst)
uint32 DAQLIB DqHolt3282toA429 (uint32 hdw)
```

**Command:**

None

**Input:**

| | |
|---|---|
| `uint32 hst/hdw` | ARINC word in standard/Holt3282 bit order |

**Output:**

Packet converted from standard to Holt3282 bit order (DqA429toHolt3282)
Packet converted from Holt3282 to standard bit order (DqHolt3282toA429)

**Description:**

These paired functions convert between normal ARINC 429 data and the representation required by the Holt3282 chip on DNA-429-566 layer.

**Note:**

None.

### *4.42.3 DqAdv566BuildFilterEntry*

**Syntax:**

```
uint32 DqAdv566BuildFilterEntry(uint32 label, uint32 new_data_only, uint32
trigger_scheduler)
```

**Command:**

None

**Input:**

| | |
|---|---|
| `uint32 label` | Label to accept |
| `uint32 new_data_only` | (TRUE/FALSE) Store only new (changed) data into the FIFO |
| `int trigger_scheduler` | (TRUE/FALSE) Upon receiving a label trigger (mark for execution) schedule entry with the same index in the scheduler table |

**Output:**

Returns uint32 assembled filter entry (no error codes)

**Description:**

This function assembles filter entries from the separate fields

**Note:**

None.

### 4.42.4 DqAdv566BuildSchedEntry

**Syntax:**

```
uint32 DAQLIB DqAdv566BuildSchedEntry(uint32 master_entry, uint32 periodic, uint32
prescaler, uint16 delay_counter)
```

**Command:**

None

**Input:**

| | |
|---|---|
| `uint32 master_entry` | TRUE for master entry, FALSE for slave entry. Upon execution, scheduler executes master entry and all following valid slave entries. Execution stops upon new master entry or zero entry |
| `uint32 periodic` | Set to TRUE to execute this entry on a periodic basis, set to FALSE to execute it one time |
| `uint32 prescaler` | Select one out of three prescalers:<br>`DQ_AR_SCHED_PS100us` – main 100us prescalers<br>`DQ_AR_SCHED_PSTB0` – first user-programmable timebase<br>`DQ_AR_SCHED_PSTB1` – second user-programmable timebase<br>Set to zero to disable entry<br>Use of three independent timebases allows you to create schedules driven from independent clocks |
| `uint16 delay_counter` | Number of prescaler ticks to wait before executing the entry.<br>For slave entries in Frame Clock mode when `DQ_AR_SCHED_SLAVETD` is set, the `delay_counter` is used to schedule the transmission of the slave entry every `delay_counter` number of master entry transmissions |

**Output:**

Returns uint32 assembled scheduler entry (no error codes)

**Description:**

This function assembles a scheduler control word entry from the separate fields. If programming DNx-429-516 channels using the Scheduler in major/minor frame mode, use the `DqAdv566BuildFrameEntry()` API instead.

The output of this API is used as the `uint32* flags` parameter in the `DqAdv566SetScheduler()`API, which writes assembled entries into the hardware Scheduler table.

**Note:**

The rate for prescaler timebases `DQ_AR_SCHED_PSTB0` and `DQ_AR_SCHED_PSTB1` is programmed using the `DqAdv566SetSchedTimebase()` function.

### *4.42.5 DqAdv516BuildFrameEntry*

**Syntax:**

`uint32 DAQLIB DqAdv516BuildFrameEntry(uint32 periodic, uint16 frame_mask) {`

**Command:**

None

**Input:**

| | |
|---|---|
| `uint32 periodic` | Set to TRUE to execute this entry on a periodic basis, set to FALSE to execute it one time |
| `uint16 frame_mask` | Bitmask that indicates which minor frame the frame entry belongs to, (e.g., 0x0001 sets entry to minor frame 0, 0x0010 sets entry to minor frame 4).<br>The same label can be transmitted as part of up to 16 different minor frames.<br>Set to zero to disable entry. |

**Output:**

Returns uint32 assembled frame entry (no error codes)

**Description:**

This function assembles a scheduler control word entry from the separate fields for DNx-429-516 channels using the Scheduler in major/minor frame mode only. For programming DNx-429-516 channels not using the Scheduler in major/minor frame mode or for programming the DNx-429-566/512, use the `DqAdv566BuildSchedulerEntry()` API to assemble scheduler control word entries.

The output of this API is used as the `uint32* flags` parameter in the `DqAdv566SetScheduler()`API, which writes assembled entries into the hardware Scheduler.

**Note:**

Supports DNx-429-516 boards.

Minor frames are programmed with the `DqAdv516SetMajorFrameDelay()` function, which sets the minor frame initial delay from the major frame boundary and sets a divider that defines how often the minor frame repeats.

### *4.42.6 DqAdv566SetConfig*

**Syntax:**

```
int DqAdv566SetConfig(int hd, int devn, int ss, int timeout_us, uint32
config)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ss` | Subsystem (DQ_SS0IN for Rx or DQ_SS0OUT for Tx) |
| `uint32 timeout_us` | Timeout to wait in case Tx is not ready or Rx message is not available |
| `uint32 config` | Configuration flags when ss = DQ_SS0OUT<br>Use DQ_AR_ENABLE_DIO0 to switch on DIO0<br>Use DQ_AR_ENABLE_DIO1 to switch on DIO1<br>Use DQ_AR_ENABLE_DIO2 to switch on DIO2<br>0 if DIO not used. |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a 429-566/512/516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function configures basic board-level ARINC device parameters.

**Note:**

DIO pins are not supported on the DNx-429-516 board.

### *4.42.7 DqAdv566SetMode*

**Syntax:**

```
int DqAdv566SetMode(int hd, int devn, int ss, int chnl, uint32 cmd)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ss` | Subsystem (DQ_SS0IN for Rx or DQ_SS0OUT for Tx) |
| `int chnl` | Channel:<br>429-566 – Rx: 0 to 5, Tx: 0 to 5, loopback Rx: 6 to 11<br>429-512 – Rx: 0 to 11<br>429-516 – Tx: 0 to 15 |
| `uint32 cmd` | Command – see description below |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not 429-566/512/516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function configures the mode of operation for each channel of an ARINC-429 layer
The following commands (they must be logically added) are valid:

**a. rate control (Rx/Tx)**

```
DQ_AR_RATEHIGH – set I/O rate to 100kbaud
DQ_AR_RATELOW – set I/O rate to 12.5kbaud
```

Rate can be selected on per-chip basis. It means that Tx0, Rx0, and Rx6; Tx1, Rx1, and Rx7 of DNA-429-566 and Rx0/Rx1, Rx2/Rx3 of DNA-429-512 layers must share the same bit rate.

**b. Parity control (Rx/Tx)**

```
DQ_AR_PARITYODD  - odd parity
DQ_AR_PARITYEVEN - even parity
DQ_AR_PARITYOFF  - no parity check/generation in hardware
```

**c. SDI filtering (Rx only)**

```
DQ_AR_SDI_ENABLED  - SDI filtering is enabled
DQ_AR_SDI_DISABLED - SDI filtering is disabled
```

**g. Rx Timestamp enabled (Rx only)**

DQ_AR_TIMESTAMP_ENABLED - write data packet into the receive FIFO along with the timestamp
DQ_AR_TIMESTAMP_DISABLED – no timestamp

When timestamp is enabled, each received message generates two entries in the FIFO – one is the actual message, and the second one is a timestamp count. The timestamp count is cleared when the layer is enabled for operation. The timestamp count increments every 10us by default. You can call `DqCmdResetTimestamp()` to select a different timestamp increment period.

**h. Tx Slow slew rate enabled (Tx only)**

DQ_AR_SLOWSLEW_ENABLED - enable slow slew rate
DQ_AR_SLOWSLEW_DISABLED (0)

**i. Enable on-chip SDI mask (Rx only)**

DQ_AR_SDIMASK0 – bit 0
DQ_AR_SDIMASK1 – bit 1

**j. for Rx FIFO control (Rx only)**

DQ_AR_LB_CHECK_PARITY - include parity check into loopback comparison
DQ_AR_ADD_TIMESTAMP - add timestamp into FIFO data
DQ_AR_IGNORE_BAD_DATA – ignore data with parity errors

**k. frame counter mode (Rx only)**

```
DQ_FRCNT_COUNT_ALL     - count all frames
DQ_FRCNT_COUNT_GOOD    - only correctly received frames
DQ_FRCNT_COUNT_FIFO    - only placed into the FIFO
DQ_FRCNT_COUNT_TRIGGER - count frames that triggered scheduler
DQ_FRCNT_COUNT_PAR_ERR - count frames with parity error
```

**l. control zero label and FIFO priority behavior (Tx only)**

```
DQ_AR_ALLOW_ZERO_LBL  - allow scheduler to output zero labels
DQ_AR_ALLOW_FIFO_HIGH - set FIFO priority higher than scheduler
```

**Note:**

None.

### 4.42.8 DqAdv566DOutCtrl

**Syntax:**

```
int DqAdv566DOutCtrl(int hd, int devn, int chan, int label, int dio_state,
int pulse_len)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chan` | Channel number |
| `int label` | Label and SDI and SSM fields |
| `int dio_state` | DIO line parameters |
| `Int pulse_len` | Length of the pulse in us (1 through 0xFFFF) |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not 429-566 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function selects either pulse-on-Tx-label mode or controls state of DIO line.
DIO0 is hard-coded to channel0, DIO1 - to 1, DIO2 - to 2. Pulse-on-Tx-label mode cannot be used on channels 3 through 5 on the 429-566.

The `dio_state` parameter consists of the DIO control state flags. The following constants are logically OR'ed together to set the `dio_state`:

```
DQL_L566_LBL_PULSE_EN  (1)          // enable pulse on a label
DQL_L566_LBL_PULSE_DIS (0)          // disable pulse on a label
DQL_L566_LBL_CONT      (1L<<14)     // Pulse repeatability
                                    //    (0=one-time,1=repeatable)
DQL_L566_LBL_POL       (1L<<13)     // Pulse polarity
                                    //    (0=negative, 1=positive)
```

**Notes:**

- *For 566 only*. Requires logic 12.5C or above
- Descriptions of <`dio_state`> flag options:

| | |
|---|---|
| `DQL_L566_LBL_PULSE_EN` | the 566 is running in pulse-on-Tx- label mode that causes the connected DIO line to pulse for <`pulse_len`> when the transmitter queues the identified <`label`>, either from the FIFO or scheduler. |
| `DQL_L566_LBL_PULSE_DIS` | the DIO pulse-on-Tx- label mode is disabled (same as not ORing `DQL_L566_LBL_PULSE_EN` into <`dio_state`>). |
| `DQL_L566_LBL_CONT` | the output pulses for each matching label; if the `DQL_L566_LBL_CONT` flag is not used, the pulse appears only once until this function is called again. |
| `DQL_L566_LBL_POL` | the DIO pulse (open collector output) is active low, otherwise it is active high.<br>`DQL_L566_LBL_POL` asserts the DIO output (i.e. |

drives low); no `DQL_L566_LBL_POL` flag deasserts the output (floats back via built-in 100 kOhm resistor to 5 V).

  - When defining the `<label>` field, bits [7:0] define the label, [10:9] define the SSM field, and [12:11] define the SDI field.

- The following is an example of using the DIO function:

```
DqAdv566DOutCtrl(hd0, DEVN, CHAN_TX+ch, 0xA0+ch,
DQL_L566_LBL_PULSE_EN|DQL_L566_LBL_CONT| DQL_L566_LBL_POL, 1000);
```

  - This example identifies the UEI chassis at handle hd0 and identifies the 566 card in slot DEVN. It looks for a transmit label matching “`0xA0+ch`” queued for transmission on channel `CHAN_TX+ch` (which must be channel 0, 1 or 2). We enable the DIO line that corresponds with the channel enabled; the DIO will pulse every time the label is matched, and we specify the active state as “low”. The pulse will last 1000*1us in duration.

### 4.42.9 DqAdv566SetFilter

**Syntax:**

```
int DqAdv566SetFilter(int hd, int devn, int chnl, uint32 cmd, uint32 from, uint32
size, uint32* labels)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Channel number |
| `uint32 cmd` | Command (see description) |
| `uint32 from` | Start entry |
| `uint32 size` | Number of entries to set/get |
| `uint32* labels` | Array of label entries to write/read |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not 429-566/512/516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function writes to or reads from the label filter.
The following commands are valid
DQ_AR_SETFILTER_PUT - change label entries
DQ_AR_SETFILTER_GET - retrieve stored entries
DQ_AR_SETFILTER_FILL_TABLE - fill the scheduler table with values taken from labels[0]

**Note:**

You can put and get data from the label filter at the same time by OR-ing put and get flags. In such a case, the label array is overwritten with the data retrieved from the layer.

### *4.42.10 DqAdv566SetScheduler*

**Syntax:**

```
int DqAdv566SetScheduler(int hd, int devn, int chnl, uint32 cmd, uint32 from,
uint32 size, uint32* flags, uint32* data)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Channel number |
| `uint32 cmd` | Command (see description) |
| `uint32 from` | Start entry |
| `uint32 size` | Number of entries to set/get |
| `uint32* flags` | Array for scheduler flags (control word) table |
| `uint32* data` | Array for scheduler data table |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not 429-566/516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function writes to or reads from scheduler table(s).
The following commands are supported:
DQ_AR_SETSCHED_PUT - change scheduler entries
DQ_AR_SETSCHED_GET - retrieve stored scheduler entries
DQ_AR_SETSCHED_DATA_ONLY – works with scheduler data only

DQ_AR_SETSCHED_FILL_TABLE - fill scheduler table with data provided. Data is taken from flags[0] and data[0] for corresponding table values.
You can put and get data from the scheduler tables at the same time by OR-ing put and get flags. In such cases, the label array is overwritten with the data retrieved from the layer.

**Note:**
The scheduler table consists of two 256-entry arrays. The first array contains scheduling information and control flags; the second array contains actual data to be transmitted. Each entry can be enabled and disabled separately.
You can combine DQ_AR_SETSCHED_DATA_ONLY with all three other modes.
DQ_AR_SETSCHED_FILL_TABLE cannot be combined with other modes.

### *4.42.11 DqAdv566SetSchedTimebase*

**Syntax:**

```
int DqAdv566SetSchedTimebase(int hd, int devn, int chnl, int timebase, uint32
rate_us)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Channel number |
| `int timebase` | Select timebase 0 (DQ_AR_SCHED_PSTB0) or timebase 1 (DQ_AR_SCHED_PSTB1) |
| `uint32 rate_us` | Desired rate in microseconds |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not 429-566/516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function programs one of two programmable scheduler timebases at a time.

**Note:**

None

### *4.42.12 DqAdv516SetMajorFrameDelay*

**Syntax:**

```
int DqAdv516SetMajorFrameDelay(int hd, int devn, uint32 chnl, uint32 mn_frame,
uint32 delay, uint32 divider)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 chnl` | TX channel |
| `uint32 mn_frame` | Minor frame number (0 to 15) |
| `uint32 delay` | Number of 100us prescaler ticks representing the delay offset between the start of the major frame and the start of the minor frame |
| `uint32 divider` | Number of 100us prescaler ticks representing the minor frame repeat rate |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not 429-516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function configures the timing of `mn_frame` minor frame by setting the initial `delay` from the major frame boundary for the selected minor frame and setting a `divider` that defines how often the minor frame repeats.

**Note:**

Supports DNx-429-516 only.

### *4.42.13 DqAdv516ConfigFrameClock*

**Syntax:**

```
int DqAdv516ConfigFrameClock(int hd, int devn, uint32
channel_mask, uint32 val)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

| | |
|---|---|
| `uint32 channel_mask` | 16-bit mask of TX channels to configure |
| `uint32 val` | Counter reset setting for each minor frame |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not a 429-516/429-516-024 |
| `DQ_BAD_PARAMETER` | More than one `val` bit is set for each minor frame |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

This function configures when the minor frame counter resets. Each of the 16 minor frames has two possible settings:

1. Setting bits 0…15 in `val` configures the corresponding minor frame counter to reset at the start of each major frame. This causes an extra delay upon each major frame being issued (default setting for backward compatibility purposes).
2. Setting bits 16…31 in `val` configures the minor frame counter to reset one clock cycle after the major frame. Bit 16 corresponds to frame 0, bit 17 to frame 1, etc…By resetting the counter after the major frame instead of at the start, periodic labels may be transmitted without an added delay.

Only one setting may be selected for each minor frame (i.e. bits 0 and 16 should not both be set).

**Notes:**

This API is for the 516 only. It should be called directly after configuring frame delay with `DqAdv516SetMajorFrameDelay()`.

### *4.42.14 DqAdv516SetTxPage*

**Syntax:**

```
int DqAdv516SetTxPage(int hd, int devn, uint32 action_mask, uint32
write_page_mask, uint32 tx_page_mask, uint32* cur_tx_page)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 action_mask` | Bitmask identifying when TX page write and active settings will take effect. Set per channel:<br>`DQ_AR_SETTXPAGE_IMMEDIATE` – change active scheduler page immediately<br>`DQ_AR_SETTXPAGE_ONMF` – change active scheduler page on major frame clock<br>For example, setting `action_mask` to 0x0001 sets channel 0 TX page settings to take effect on a major frame clock boundary and sets channels 1 thru 15 TX page settings to take effect immediately. |
| `uint32 write_page_mask` | Bitmask identifying which TX page to write to from host per channel<br>For example, setting `write_page_mask` to 0x0001 sets channel 0 data writes to Scheduler Page 1 and sets channels 1 thru 15 data writes to Scheduler Page 0. |
| `uint32 tx_page_mask` | Bitmask identifying which TX page to make active for transmission per channel<br>For example, setting `tx_page_mask` to 0x0001 sets TX 0 to transmit data from Scheduler Page 1 and sets TX 1 thru TX 15 to transmit data from Scheduler Page 0. |
| `uint32* cur_tx_page` | Bitmask of the current active page for each channel |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not 429-516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function programs Scheduler data page usage for each DNx-429-516 channel in major/minor frame mode. Scheduler data array paging is only available on the DNx-429-516 for channels in major/minor frame mode.

The Scheduler data arrays consist of a 256x32 Page 0 array and a 256x32 Page 1 array. TX page buffering allows the user to write data words as needed to one page and not overwrite data in the process of transmitting from the other page. Pages can be programmed to swap on a new major frame boundary or immediately after the scheduler has completed transmitting pending data.

Programming identifies which data page (Page 0 or Page 1) is available to write to from the host side and which page will be read from when transmitting ARINC data.

**Note:**

Supports DNx-429-516 only.

### *4.42.15 DqAdv566SetFifoRate*

**Syntax:**

```
int DqAdv566SetFifoRate(int hd, int devn, int ss, int chnl, int timebase, uint32
rate_us)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ss` | Subsystem (DQ_SS0IN for Rx or DQ_SS0OUT for Tx) |
| `int chnl` | Channel number |
| `int timebase` | Select TX FIFO timebase and rate:<br>`DQ_AR_FIFO_PSTB0` – FIFO is paced by TX ready<br>`DQ_AR_FIFO_PSTB1` – FIFO is paced by prescaler |
| `uint32 rate_us` | delay in us between output packets when `timebase` is set to `DQ_AR_FIFO_PSTB1`, 0 as soon as a slot becomes available |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not 429-566/516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up a timebase for writing output packets to the Tx FIFO

**Note:**

None

### *4.42.16 DqAdv566SetChannelCfg*

**Syntax:**

```
int DqAdv566SetChannelCfg(int hd, int devn, int ss, int chan, uint32 actions)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ss` | Subsystem (DQ_SS0IN for Rx or DQ_SS0OUT for Tx) |
| `int chan` | Channel |
| `int actions` | What to select :<br>`DQ_AR_ENABLE_Tx - enable transmit operations (Tx)`<br>`DQ_AR_ENABLE_Rx - enable receive operations (Rx)`<br>`DQ_AR_ENABLE_SCHEDULER - enable scheduler (Tx)`<br>`DQ_AR_ENABLE_LOOPBACK - loopback validation of transmitted packets is enabled (Tx)`<br><br>`DQ_AR_ENABLE_FILTER - enable receive operations (Rx)`<br>`DQ_AR_ENABLE_RxFIFO - enable receive FIFO (Rx)`<br>`DQ_AR_ENABLE_TxFIFO - enable transmit FIFO (Tx)`<br>`DQ_AR_LOGIC_LOOPBACK - for internal logic tests (internal to HI-3282)`<br>`DQ_AR_SCHED_FRAMECLK – set scheduler to Frame Clock mode (TX 429-566/429-516)`<br>`DQ_AR_SCHED_SLAVETD – allow scheduling slave entries using the delay_counter field when in Frame Clock mode (TX 429-566/429-516)`<br>`DQ_AR_SCHED_MJMN - set scheduler to Major/Minor Frame mode (TX 429-516)` |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not 429-566/512/516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Set up operating mode for each channel

**Notes:**

`DQ_AR_ENABLE_SCHEDULER` must be logically added to `DQ_AR_SCHED_FRAMECLK` when enabling the scheduler in Frame Clock mode, and added to `DQ_AR_SCHED_MJMN` in Major/Minor Frame mode. The DNx-429-516 is the only board version that supports Major/Minor Frame mode.

### *4.42.17 DqAdv516EnableTransmitters*

**Syntax:**

```
int DqAdv516EnableTransmitters(int hd, int devn, uint32 tx_fifo_mask,
uint32 scheduler_mask)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 tx_fifo_mask` | Bitmask enabling TX FIFO per channel<br>For example, setting `tx_fifo_mask` to 0x0005 enables TX FIFOs on channels 0 and 2. |
| `uint32 scheduler_mask` | Bitmask enabling TX schedulers per channel |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not 429-516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function enables and disables TX schedulers and FIFOs while channels are in operation.

**Notes:**

Support DNx-429-516 only. Disabling both TX FIFO and scheduler stops TX transmitter from driving the bus.

### 4.42.18 DqAdv516ChangeBaudRate

**Syntax:**

`int DqAdv516ChangeBaudRate(int hd, int devn, int channel_mask, int baud_rate)`

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel_mask` | Channel mask to apply change to ORed with:<br>For RX channels, OR with `DQ_AR_CHANGE_RX`<br>For TX channels, OR with `DQ_AR_CHANGE_TX` |
| `int baud_rate` | Baud rate 10kbaud..200kbaud |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not 429-516/429-516-024 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function changes the channel baud rate. Requires logic 11.48+

The `channel_mask` must have a TX or RX change flag ORed in with the channels.
For example,

- to change the baud rate on channel 0..3 receivers:
  `chan_mask = 0xF | DQ_AR_CHANGE_RX;`
- to change the baud rate on channel 4..7 transmitters:
  `chan_mask = 0xF0 | DQ_AR_CHANGE_TX;`
- to change the baud rate on channel 8 transmitter and receiver:
  `chan_mask = 0x100 | DQ_AR_CHANGE_TX | DQ_AR_CHANGE_RX;`

**Notes:**

This function is supported for DNx-429-516 and DNx-429-516-024 boards only.

### 4.42.19 DqAdv516ChangeWordLength

**Syntax:**

`int DqAdv516ChangeWordLength(int hd, int devn, int channel_mask, int word_len)`

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel_mask` | Channel mask to apply change to |
| `int word_len` | Number of bits in ARINC word (default is 32 = pass 0) |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not 429-516/429-516-024 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function changes the word length to support non-standard ARINC transmissions. Requires logic 14.C8+

Non-standard word lengths may range from 17-31 bits. Setting the word length to 16 or lower does nothing, and the length will default to the standard 32 bits. When transmitting words of a non-standard length, TX and RX data will not be scrambled/unscrambled as for the Holt HI-3282. Data bits will be transmitted/received LSB first to/from the ARINC bus.

**Notes:**

This function is supported for DNx-429-516 and DNx-429-516-024 boards only.

### *4.42.20 DqAdv566ChangeBaseClock*

**Syntax:**

`int DqAdv566ChangeBaseClock(int hd, int devn, int clock_div)`

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

| | |
|---|---|
| `int clock_div` | Change HI-3282 chip clock rate (changes baud rate).<br>`clock_div` +1 = desired divider (zero based). |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not 429-566/512/516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function changes the base clock used by the onboard ARINC 429 serial interface, the HI-3282 chip. Changing the base clock rate results in a change to the baud rate. Requires logic 11.48+

The base clock rate is 33 MHz, and the default `clock_div` value is 32, which maps to an actual divider of 33 and provides a 1 MHz chip clock rate. The supported divider range is 25 (1320 kHz) to 50 (660 kHz).

HI-3282 default baud rate at 1 MHz is at high speed=100 kbaud+/-1% and at low speed 12 to 14.5 kbaud. Changing the frequency changes the baud rate proportionally.

**Notes:**

This function is supported for DNx-566/512/516 layers.

DqAdv566Enable

**Syntax:**

`int DqAdv566Enable(int hd, int devn, uint32 actions)`

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int actions` | TRUE – enable device operations<br>FALSE – disable device operations |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not 429-566/512/516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function enables and disables operations on the layer.

**Notes:**

None

### *4.42.21 DqAdv566SendPacket*

**Syntax:**

```
int DqAdv566SendPacket(int hd, int devn, int chan, int priority, uint32 data)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ss` | Subsystem (DQ_SS0IN for Rx or DQ_SS0OUT for Tx) |
| `int chnl` | Channel number |
| `int priority` | `DQ_AR_TxPRIORITY_HIGH` - high priority (preempt scheduler)<br>`DQ_AR_TxPRIORITY_LOW` - low priority (send as scheduler permits)<br>`DQ_AR_TxIMMEDIATE` - immediate return from the function, do not wait until slot becomes available, otherwise wait timeout defined by `DqAdv566SetConfig` (`ss` = `DQ_SS0OUT`) |
| `uint32 data` | Properly formed message (see `DqAdv566BuildPacket`) |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not 429-566/516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sends an ARINC data word packet to the specified channel (uses the TX high priority register or low priority register as specified by the `priority` parameter).

**Note:**

The function returns after data is transmitted on the interface. If data cannot be transmitted in the timeout period, the function returns DQ_TIMEOUT_ERROR.

### *4.42.22 DqAdv566SendFifo*

**Syntax:**

```
int DqAdv566SendFifo(int hd, int devn, int chan, int packets, uint32* data,
uint32* accepted, uint32* available)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Channel number |
| `int packets` | Number of packets to send |
| `uint32* data` | Array of packets |
| `uint32* accepted` | Number of packets the function was able to write to the channel FIFO |
| `uint32* available` | Number of entries still available in the FIFO |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not 429-566/516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function puts packets of data into the Tx FIFO

**Note:**

This function returns after data is transmitted on the interface. If data cannot be transmitted in the timeout period, the function returns a `DQ_TIMEOUT_ERROR`.

The data words are expected to be in Holt3282 bit order. The `DqAdv566BuildPacket` function will assemble data packets into the correct bit order. If the data words are constructed manually into standard bit order, the `DqA429toHolt3282` function must be used to convert them.

### *4.42.23 DqAdv566RecvPacket*

**Syntax:**

```
int DqAdv566RecvPacket(int hd, int devn, int chan, int where, uint32* data)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Channel number |
| `int where` | Mode of operation:<br>`DQ_AR_Rx_LATEST` – get latest data from the interface<br>`DQ_AR_Rx_FIFO` – get a sample from the FIFO, queued<br>`DQ_AR_RxIMMEDIATE` - immediate return if no data<br>Immediate flag can be or-ed with the first two flags |
| `uint32* data` | Received packet (raw) |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a 429-566/512/516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function receives a packet of data from the Rx interface.

**Note:**

A DNA-429-566 uses channels Rx6 to Rx11 as loopback channels for Tx0 to Tx5.

### *4.42.24 DqAdv566RecvFifo*

**Syntax:**

```
int DqAdv566RecvFifo(int hd, int devn, int chan, int maxsz, uint32* data,
uint32* copied, uint32* remains)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Channel number |
| `int maxsz` | Maximum number of messages to retrieve |
| `uint32* data` | Array for received messages |
| `uint32* copied` | Number of packets the function was able to retrieve from the channel FIFO |
| `uint32* remains` | Number of messages remains in the FIFO |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a 429-566/512/516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function retrieves messages from the Rx FIFO.

**Note:**

The function returns after data is received on the interface. If a timestamp was enabled for that channel, the function returns two uint32s per packet, one for the packet and one for the timestamp.

The data words are returned in Holt3282 bit order. The `DqAdv566ParsePacket` function will disassemble data packets from this bit order. If you wish to utilize the raw values in standard bit order, use the `DqHolt3282toA429` function to convert them.

### *4.42.25 DqAdv566ReadWriteFifo*

**Syntax:**

```
     int DqAdv566ReadWriteFifo(int hd, int devn, int chanwr, int chanrd, int
writesz, uint32* wrdata, int* written, int* available, int readsz, uint32* rddata,
int* read, int* remains)
```

**Command:**

DQCMD_WRRDFIFO

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chanwr` | Channel to write to |
| `int chanrd` | Channel to read from |
| `int writesz` | Number of messages to write |
| `uint32* wrdata` | Array of messages to write |
| `int* written` | Number of messages written to the channel FIFO |
| `int* available` | Number of entries still available in the FIFO |
| `int rdsize` | How many messages try to read |
| `uint32* rddata` | Array to store read messages |
| `int* read` | Number of messages read |
| `int* remains` | Number of messages remains in the FIFO |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a 429-566/516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function writes and receives packets from the specified channel FIFO.

**Note:**

The function returns after data is received on the interface. If a timestamp was enabled for that channel, the function returns two uint32s per packet, one for the packet and one for the timestamp.

### *4.42.26 DqAdv566ReadWriteAll*

**Syntax:**

```
int DqAdv566ReadWriteAll(int hd, int devn, uint32 tx_ch, uint32 rx_ch,
uint32* tx_chan_arr, uint32* tx_data_arr[], uint32* tx_size, uint32* tx_wrt,
uint32* tx_avl, uint32* rx_chan_arr, uint32* rx_data_arr[], uint32* rx_size,
uint32* rx_read, uint32* rx_rmns)
```

**Command:**

DQCMD_WRRDFIFO

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 tx_ch` | Number of channels in transmit channel list |
| `uint32 rx_ch` | Number of channels in receive channel list |
| `uint32* tx_chan_arr` | Tx channel list |
| `uint32* tx_data_arr[]` | Array of pointers to the arrays of the transmit data |
| `uint32* tx_size` | Array of number of messages to be transmitted for each channel in Tx channel list |
| `uint32* tx_wrt` | Array in which the function returns the actual number of messages written to each channel |
| `uint32* tx_avl` | Array in which the function returns the number of entries available in the FIFO for each channel |
| `uint32* rx_chan_arr` | Rx channel list |
| `uint32* rx_data_arr[]` | Array of pointers to the arrays for storing received data |
| `uint32* rx_size` | Array of number of messages to be received for each channel in the Tx channel list |
| `uint32* rx_read` | Array of number of messages actually received for each channel |
| `uint32* rx_rmns` | Array of number of messages remaining in the receive FIFO for each channel |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a 429-566/512/516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function writes and receives packets from the specified channel FIFOs.

Special channel numbers are defined for the purpose of accessing special data areas (for write only):

`DQ_AR566_CH_SCHED` - scheduler table
`DQ_AR566_CH_SCHDATA` - scheduler data
`DQ_AR566_CH_FILTER` - filter table
"From" parameter should be first entry in the channel data for these special channels.

This function allows writing scheduler data and table in a single write. To do that, the maximum size of the Tx channel list is extended to twelve channels – six for scheduler table for each channel and six for scheduler data. For example, to write to the scheduler table for channel 3, use (DQ_AR566_CH_SCHED|3) and (DQ_AR566_CH_SCHDATA|3) to write scheduler data for the same channel. The first entry in the scheduler (or filter) entry or data must be the <from> parameter which tells from which entry to write following scheduling data. For example, if one would like to write two entries into a table from index 5 in the scheduler table, it must prepare a `tx_data_arr[]` data array consisting of three values: [5, entry1, entry2], where each entry is a 32-bit word to be written into scheduler table. Please note that the size must include <from> into count, thus `tx_size = 3;`

**Note:**

This function allows sending and receiving data for the whole device in one packet. Currently, the function is limited to sending and receiving one packet of 1518 bytes each. This allows you to transmit and receive 58 messages per packet for six channels maximum. This is not a significant limitation: ARINC-429 interface allows you to send 2777 messages a second. Thus, the function should be called 2777/60 = 46 times a second or every 22ms.

For realtime operation, one should use VMap mode.

### *4.42.27 DqAdv566GetStatus*

**Syntax:**

```
int DqAdv566GetStatus(int hd, int devn, int ss, int chan, uint32 request,
uint32* errors, uint32* count)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ss` | Subsystem (use DQ_SS0IN for Rx) |
| `int chnl` | Channel number |
| `int request` | What status to return (flags can be ORed):<br>• `DQ_AR_STATUS_CLEAR_ERROR - clear sticky error flags (all channels together)`<br>• `DQ_AR_STATUS_GET_TOTAL - get total number of packets received (for the channel)`<br>• `(12) DQ_AR_STATUS_GET_FRM_CTR - get frame counter for Rx0..Rx11`<br>• `(12) DQ_AR_STATUS_GET_FRM_ERR - get number of loopback errors for Rx0..Rx11`<br>• `(12) DQ_AR_STATUS_GET_FRM_MIS - get number of missed frames for Rx0..Rx11`<br>• `(12) DQ_AR_STATUS_GET_FIFO_CNT - get current levels in receive FIFOs Rx0..Rx11` |

**Output:**

| | |
|---|---|
| `uint32* errors` | Array of requested status information. See Notes section below. |
| `uint32* count` | Total number of uint32 words returned in the `errors` array |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a 429-566/512/516 |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function requests the error and status of the interface.

**Notes:**

Output array values are as follows:

- `errors[0]` is the board status, which is always returned with this API and provides loopback error bits, scheduler overrun bits, and parity error bits:
```
// +0 – Board status (EISR) bits:
//          [11..0]  - parity error for Rx11..Rx0
//          [17..12] - scheduler overrun error for Tx5..Tx0 (cannot output one or more
//                     entries on time)
//          [29..18] - loopback error for Rx11..Rx0 (loopback should be enabled)
```
- Subsequent `errors` status information depends on which flags are ORed into `int request` and subsequent status words are always returned in the following order:
  - `DQ_AR_STATUS_GET_TOTAL -` get total number of packets received (for the channel specified as `chnl`); returns (1) 32-bit `errors` words.
    ```
    // + if DQ_AR_STATUS_GET_TOTAL
    //   4 bytes - frame count for requested channel
    ```
  - `(12) DQ_AR_STATUS_GET_FRM_CTR -` get frame counter for Rx0..Rx11; returns (12) 32-bit `errors` words.
    ```
    // + if DQ_AR_STATUS_GET_FRM_CTR:
    //   48 bytes (12*4) frame counts for Rx0..Rx11
    ```
  - `(12) DQ_AR_STATUS_GET_FRM_ERR -` get number of loopback errors for Rx0..Rx11; returns (12) 32-bit `errors` words.
    ```
    // + if DQ_AR_STATUS_GET_FRM_ERR:
    //   48 bytes (12*4) number of loopback errors for Rx0..Rx11
    ```
  - `(12) DQ_AR_STATUS_GET_FRM_MIS -` get number of missed frames for Rx0..Rx11; returns (12) 32-bit `errors` words.
    ```
    // + if DQ_AR_STATUS_GET_FRM_MIS:
    //   48 bytes (12*4) number of missed frames for Rx0..Rx11
    ```
  - `(12) DQ_AR_STATUS_GET_FIFO_CNT -` get current levels in receive FIFOs Rx0..Rx11; returns (12) 32-bit `errors` words.

As an example: if `request= DQ_AR_STATUS_GET_TOTAL| DQ_AR_STATUS_GET_FRM_CTR`, you'll read the board status bits that get returned in `errors[0]; DQ_AR_STATUS_GET_TOTAL` returns (1) uint32 word (`errors[1]);` and `DQ_AR_STATUS_GET_FRM_CTR` returns (12) uint32 words (`errors[2..13]`).

### *4.42.28 DqAdv566EnableByChip*

**Syntax:**

```
int DqAdv566EnableByChip(int hd, int devn, int chipmask, uint32 actions)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chipmask` | 0 – all channels<br>(1<<N) – Nth channel |
| `int actions` | TRUE – enable device operations<br>FALSE – disable device operations |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a 429-566/512/516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function enables and disables operations on the layer on IC per IC basis.

**Notes:**

Recommended for debug purposes only.

### 4.42.29 DqAdv566SetChannelList

**Syntax:**

```
int DqAdv566SetChannelList(int hd, int devn, int chan, uint32 ss, int32 size,
uint32* entries)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ss` | Subsystem (DQ_SS0IN for Rx or DQ_SS0OUT for Tx) |
| `int chan` | Channel number |
| `int32 size` | Size of the channel list |
| `uint32* entries` | Channel list entries |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a 429-566/512/516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up the channel list for ARINC 566 operations in messaging mode.

**Notes:**

This function sets up a channel list separately for transmitters and receivers. The channel list can have a timestamp flag to handle ARINC packets with the timestamp information (Rx only).

Use this function before calling `DqMsgInitOps`()

### *4.42.30 DqAdv566ClearFifo*

**Syntax:**

`int DqAdv566ClearFifo(int hd, int devn, uint32 ch_mask, int action)`

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 ch_mask` | Bit mask describing the channels to clear |
| `int action` | DQL_IOCTL566_CHANGE_FINCLEAR to clear input FIFO(s)<br>DQL_IOCTL566_CHANGE_FOUTCLEAR to clear output FIFO(s) |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a 429-566/512/516 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Clears input and/or output FIFOs on 566/512/516 layers.

**Note:**

Function can be applied to more than one channel on the layer by OR-ing channel mask with (1L<<channel_number).

You don't need to call this function initially because `DqAdv566Enable()` automatically cleans the FIFOs.

This function is only useful once the layer is started.

### *4.42.31 DqAdv566PauseAndResume*

**Syntax:**

```
int DqAdv566PauseAndResume(int hd, int devn, uint32 ch_mask, int
todo)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 ch_mask` | Bitmask of channels to configure (0…23) |
| `int todo` | One of the following actions: |

```
DQL_IOCTL566_CHANGE_PAUSE     //pause receiver and clear
                              RX FIFO
DQL_IOCTL566_CHANGE_RESUME    //resume receiver
DQL_IOCTL566_CHANGE_PAUSE_TX  //pause transmitter and
                              clear TX FIFO
DQL_IOCTL566_CHANGE_RESUME_TX //resume transmitter
```

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | Illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | Device indicated by `devn` does not exist or is not a 429-566/512/516 |
| `DQ_BAD_PARAMETER` | `todo` is not one of the defined constants |
| `DQ_SEND_ERROR` | Unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | Nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | Error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | Successful completion |
| Other negative values | Low level IOM error |

**Description:**

Pause or Resume the specified channel(s). The receive or transmit FIFO is cleared when data transfer is paused.

**Notes:**

For 566/512/516. This function is only useful once the layer is started.

## *4.43 DNA-1553-553 layer*

The DNA-1553-553 layer is designed to support MIL-STD-1553A and MIL-STD-1553B interfaces. Two dual redundant independent channels are available. The layer can support up to 32 remote terminals (RTs), one Bus Monitor (BM), and one Bus Controller (BC).

The layer has two independent channels and each channel incorporates all that is needed to communicate with a dual redundant 1553 bus. Bus coupling (transformer/direct) is software-selectable.

As shown on the picture, each channel has dual 1553 decoders that are capable of decoding independent streams of 1553 Manchester words and passing them to upper-level subsystems. Decoders can detect various timing errors on the bus and also keep track of the gap interval between messages as well as data parity errors and type of received data words (command-status or data).

When data is stored into the 1024 32-bit word FIFO (BM mode), it is stored from both A and B bases and each control/status word may be optionally time-stamped. In RT mode, the 1553 protocol is parsed and processed by the RT state machine which inherently supports up to 32 RTs simulated by the single channel. Each RT may be individually enabled and each sub-address may be also individually configured for RX/TX or both, with maximum and minimum number of allowable data words per subsystem. Mode commands may be enabled and disabled as well. Also each sub-address and each mode command have a flag that controls interrupt generation upon receiving a corresponding RX /TX or mode command. Switching between A and B redundant buses takes place automatically upon receiving a command on the bus, and the host may receive an interrupt once it happens.

The Manchester encoder allows data output on A and/or B buses. However, it always accepts data from the same source; i.e., it is impossible to send different data to A and B buses at the same time. The encoder has a two-256x32 word FIFO that accepts 1553 data in a format that includes bus inactivity gap time delay, word type, and parity information. These FIFOs are called high- and low- priority FIFOs and are used for both RT and BC modes. Currently, a BC only has access to the low priority FIFO. Data is outputted from the FIFOs as a complete message and a low-priority FIFO waits until all messages from the high-priority FIFOs are gone.

A dedicated memory controller interfaces with 16Mbytes of fast burst PSRAM. It keeps up with data requests from both channels for the TX data and accepts RX messages and status information as well. Once a message is received or transmitted for the particular subsystem, special flags are set and once new data is placed in the TX buffer or data is read from the RX buffer, those flags are cleared. DNA may write or read data in blocks as large as 1024 16-bit data words at a time. A read or write is executed as an atomic transaction, i.e., no data change allowed during a read or write to/from the DNA bus.

This document describes the initial function set for BM and RT modes of operation.

### *4.43.1 DqAdv553SetMode*

**Syntax:**

```
int DqAdv553SetMode(int hd, int devn, int channel, uint32 mode, uint32 flags)
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 mode | Mode of operation |
| uint32 flags | Additional initialization flags |
| **Output** | |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description**

<mode> - mode of operation

```
#define DQ_L553_MODE_BM    (1L<<0)   // bus monitor mode
#define DQ_L553_MODE_RT    (1L<<1)   // remote terminal mode
#define DQ_L553_MODE_BC    (1L<<2)   // bus controller mode
```

The following modes of operation can be used simultaneously:

- Remote Terminal and Bus Monitor
- Bus Controller and Bus Monitor

Bus monitor mode can be used in conjunction with remote terminal mode. Bus controller mode can only be used by itself.

<flags> - additional initialization flags

```
#define DQ_L553_DISCONNECT       (0L<<0) // disconnect from the bus (default)

#define DQ_L553_TRANSFORMER      (2L<<0) // transformer-coupled

#define DQ_L553_COUPLE_DIRECTLY  (3L<<0) // direct-coupling (potential isolation issues)

#define DQ_L553_FORCE_A          (1L<<4) // make I/O compliant with MIL-1553A standard
(default is MIL-1553B)
```

### 4.43.2 DqAdv553BITTest

**Syntax:**

```
int DqAdv553BITTest(int hd, int devn, int channel, uint32 testmode, uint32
iterations, uint32* bus_errors)
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 testmode | What to test |
| uint32 iterations | How many iterations to perform |
| **Output** | |
| uint32* bus_errors | Cumulative errors detected on the bus |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function initiates built-in tests and returns its results.
Following built-in tests are defined:

```
#define DQ_L553_BIT_CABLEWRAP  (1L<<0)   // test when A and B are on the same trunk
#define DQ_L553_INTERNAL       (1L<<1)   // internal bit test
#define DQ_L553_TWOCHANNELS    (1L<<2)   // tests bus A of channel 0 against
                                         // bus A of channel 1. Ditto bus B
```

Possible <bus_errors>:

```
SL553_PORT_IR_ITA    (1L<<26) // Bus: One of the following errors was detected on bus A/B :
invalid bit value

SL553_PORT_IR_ITB    (1L<<25) // Bus: (invalid Manchester code during SYNC or data bit time)
or parity error

SL553_PORT_IR_BEA    (1L<<24) // Bus: Bit timing error was detected on bus A/B, one of the
following errors: invalid combination on the input,

SL553_PORT_IR_BEB    (1L<<23) // Bus: rising/falling edge timing is invalid, bit timing
timeout detected

SL553_PORT_IR_MED    (1L<<22) // Bus: Message error detected - data error, suppress status

SL553_PORT_IR_TMW    (1L<<21) // Bus: Too many words were received within RX message,
suppress status

SL553_PORT_IR_TFW    (1L<<20) // Bus: Too few words were received within RX message,
suppress status
```

```
SL553_PORT_IR_ICD    (1L<<19) // Bus: Message error detected - illigal/illogical command
error, send status
SL553_PORT_IR_BCH    (1L<<18) // Bus: Bus was changed (command arrived on different bus)
SL553_PORT_IR_RTD    (1L<<13) // Bus: RTRT Timeout detected
SL553_PORT_IR_RVF    (1L<<12) // Bus: RTRT Validation failed
SL553_PORT_IR_DWG    (1L<<11) // Bus: Gap within data word detected (use only when
xxRTR_DGP>0)
SL553_PORT_STS_BEB    (1L<<9) // BUS: Bit timing error on bus B
SL553_PORT_STS_BEA    (1L<<8) // BUS: Bit timing error on bus A
SL553_PORT_STS_PEB    (1L<<7) // BUS: Parity error on bus B
SL553_PORT_STS_PEA    (1L<<6) // BUS: Parity error on bus A
SL553_PORT_STS_ILC    (1L<<5) // RT: Illegal command
SL553_PORT_STS_MER    (1L<<4) // RT: Message error
SL553_PORT_STS_TMW    (1L<<3) // RT: Too many words were detected in RX message
SL553_PORT_STS_TFW    (1L<<2) // RT: Too few words were detected inm RX message
```

### 4.43.3 *DqAdv553Control*

**Syntax:**

```
int DAQLIB DqAdv553Control(int hd, int devn, int channel, uint32 flags,
pDQ553Control params)
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 flags | What parameter to set up |
| pDQ553Control params | Parameters |
| **Output** | |
| uint32* bus_errors | Cumulative errors detected on the bus |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function modifies multiple parameters on the fly (i.e., while 553 is enabled)
It can change TX block, bus to use, etc. <flags> specifies parameters to control.

Currently two parameters can be controlled:

1. Select with block to use for Tx - 0 or 1 per RT. To set block of memory for transmission user needs to specify DQ_L553_SET_TX_BLOCK in <flags> field and block number in the member `<rx_block>` of DQ553Control structure.
2. Enable and disable remote terminals while application is running. To perform an enable/disable remote terminal function, a user needs to specify DQ_L553_SET_RT_ENABLE in <flags> and set "1" in the bit field `<rt_enabled>` of DQ553Control structure.

### 4.43.4 DqAdv553ConfigBM

**Syntax:**

```
int DAQLIB DqAdv553ConfigBM(int hd, int devn, int channel, uint32 busmode, uint32
rt_size, uint32* rtsa_list, uint32* trig_list);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 busmode | What to monitor: bus A, bus B or both buses |
| uint32 rt_size | Size of the following RT/SA and trigger lists |
| uint32* rtsa_list | Array of RT/SA to monitor, NULL to monitor all RTs |
| uint32* trig_list | List of trigger conditions to match rts_list (or NULL) |
| **Output** | |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**

This function defines what a Bus Monitor should actually monitor
<busmode> defines what bus to listen at:

```
#define DQ_L553_BM_LSTN_A    (1L<<0) // listen bus A
#define DQ_L553_BM_LSTN_B    (1L<<1) // listen bus B
```

```
#define DQ_L553_BM_TX_A      (1L<<2) // enable transmission on bus A
#define DQ_L553_BM_TX_B      (1L<<3) // enable transmission on bus B
#define DQ_L553_STORE_TS     (1L<<17) // Store timestamp information
#define DQ_L553_STORE_FLAGS  (1L<<18) // Store bus flags/status information
#define DQ_L553_BM_LIST_ADD  (1L<<31) // add to RTSA list
```

<busmode> allows you to monitor either bus A or B or both. This field also allows adding to the current RTSA lists in multiple calls using a `DQ_L553_BM_LIST_ADD` flag. To clear the list, a user should call this function with NULL as a pointer to the list.

Flags `DQ_L553_BM_TX_A` and `DQ_L553_BM_TX_B` enable transmission on the bus. Only one flag can be selected at a time. These flags are required in situation in which a <channel> is in the bus monitor mode. However, it is also used to transmit simple commands using a transmission FIFO. In this way a user can emulate a simple bus controller and bus monitor on the same channel. See `DqAdv553WriteTxFifo()` for details.

By supplying rt_size = 0 and rts_list = NULL, the DNA-1553-553 is instructed to monitor all activity on the bus for all terminals and all sub-addresses.

One can limit the scope of monitored activity by providing an actual channel list <rts_list>:

```
#define DQ_L553_BM_SADDR(N)    (((N)&0x1f)<<0) // sub-address <reserved>
#define DQ_L553_BM_SADDR_SEL   (1L<<5)    // Sub-address field is valid <reserved>
#define DQ_L553_BM_RT(N)       (((N)&0x1f)<<6) // RT number
#define DQ_L553_BM_RT_SEL      (1L<<11)   // RT field is valid
#define DQ_L553_INH_RX         (1L<<12)   // Inhibit Rx packets <reserved>
#define DQ_L553_INH_TX         (1L<<13)   // Inhibit Tx packets <reserved>
#define DQ_L553_INH_MODE       (1L<<14)   // Inhibit Mode commands <reserved>
#define DQ_L553_INH_ERROR      (1L<<15)   // Inhibit storing bus errors <reserved>
```

The definitions for RT and SA are encapsulated in `DQ_L553_RT_SA(RT,SA)` macro.

<trig_list> is TBD

### 4.43.5 DqAdv553ConfigBMSetFilter

**Syntax:**

```
int DAQLIB DqAdv553ConfigBMSetFilter(int hd, int devn, int channel, uint32
busmode, uint32 f_size, pDQBM553Filter pFilter);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 busmode | What to monitor: bus A, bus B or both buses |
| uint32 f_size | Size of the following filter list |
| pDQBM553Filter pFilter | Array of `DQBM553Filter` structures that defines filter for BM mode |

| **Output** | |
|---|---|
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**
This function sets up a filter for Bus Monitor mode. The filter allows limiting the scope of data passing to the user by imposing conditions on what kind of data is of interest.

**Note**
*Not supported in the current revision*

### *4.43.6 DqAdv553ConfigBMSetTrigger*

**Syntax:**
```
int DqAdv553ConfigBMSetTrigger(int hd, int devn, int channel, uint32 busmode,
uint32 t_size, pDQBM553Filter pTrigger);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 busmode | What to monitor: bus A, bus B or both buses |
| uint32 t_size | Size of the following trigger list |
| pDQBM553Filter pFilter | Array of `DQBM553Trigger` structures that defines filter for BM mode |
| **Output** | |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |

| | |
|---|---|
| Other negative values | low level IOM error |
| | |

**Description**
This function sets up a start/stop trigger for BM mode. The trigger can include various conditions: Rx/Tx/Mode/Errors, etc. It also allows setting up a decimation ratio for an RT/SA pair, to decrease the amount of data transferred.

**Note**
*Not supported in the current revision*

### 4.43.7 DqAdv553RecvBMMessages

**Syntax:**

```
int DAQLIB DqAdv553RecvBMMessages(int hd, int devn, int channel, int rq_size,
pDQBM553Message pMsg, uint32 *messages);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32* rq_size | Size of the allocated memory for pMsg array, returns actual number of bytes stored in the buffer |
| `pDQBM553Message pMsg` | Pointer to where to store captured data |
| uint32* messages | Actual number of messages received |
| **Output** | None |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**

This function retrieves messages stored in BM FIFO.

```
typedef struct {
      uint8 channel;    // channel
      uint8 stat;       // status
      uint8 size;       // size of the following data, bytes
      uint32* data[];   // variable size data
} DQBM553Message, *pDQBM553Message;
```

Depending on how a BM was configured, the data can be delivered with or without additional information like timestamps and flags. The data has the following format:

| | 31 | 30 | 29 | | 16 | 15 | | 0 |
|---|---|---|---|---|---|---|---|---|
| Command/Status | PAR | 1 | GAP1553 | | | DATA1553 | | |
| Timestamp (opt.) | 0 | | 30-bit timestamp, 15.15ns resolution | | | | | |
| Flags (opt.) | 0 | | | | | | | BUS |
| Data 0 | PAR | 0 | GAP1553 | | | DATA1553 | | |
| … | PAR | 0 | GAP1553 | | | DATA1553 | | |
| Data N | PAR | 0 | GAP1553 | | | DATA1553 | | |

Timestamp – optional, time when command or status word is received (optional)
bit 31 – If Command/Status this bit is set to 1
BUS – the bus from which data was received (if both buses A and B are enabled, optional)
PAR – parity bit
GAP1553 – time interval from the previous packet received, in 15.15ns resolution, 248us maximum. If delay is longer than that, the GAP1553 counter stays at 0x3fff.
DATA1553 – actual data received on the bus

Zero status is considered to be normal.
Error status is:

```
#define DQ_L553_BMSTATUS_OVER   (0x80)        // BM FIFO overrun
```

**Note**

Add `DQ_L553_READFIFO_ALL` to the channel number to receive data from the FIFO in one message instead of breaking it into separate messages. Each transmission on a 1553 bus always starts with aCommand or Status word followed by timestamp and status flags (if enabled). The bus monitor then stores a variable number of received data words (one to thirty two).

### *4.43.8 DqAdv553ConfigRT*

**Syntax:**

```
int DAQLIB DqAdv553ConfigRT(int hd, int devn, int channel, uint32 rt_mode, uint32
rt_size, uint32* rtsa_list, uint32* valid_list);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 rt_mode | Remote terminal mode of operation |

| | |
|---|---|
| uint32 rt_size | Size of the following RT/SA and data validation lists |
| uint32* rtsa_list | Array of RT/SA entries |
| uint32* valid_list | List of validation conditions to match rtsa_list (or NULL) |
| **Output** | |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

### Description

This function defines parameters of operation for remote terminal mode.
<rt_mode> is a combination of flags:

```
#define DQ_L553_RT_LSTN_A    (1L<<0)  // listen bus A
#define DQ_L553_RT_LSTN_B    (1L<<1)  // listen bus B
#define DQ_L553_RT_TX_A      (1L<<2)  // enable transmission on bus A
#define DQ_L553_RT_TX_B      (1L<<3)  // enable transmission on bus B
#define DQ_L553_RT_RX_CMD    (1L<<29) // return CMD and STS before the data
#define DQ_L553_RT_DEFER_EN  (1L<<30) // defer enabling of RT upon realtime
```

In most circumstances, both buses should be enabled for both transmitting and listening. An RT replies on the same bus a command came from. A Bus Controller can force a disable of one of the buses if it keeps transmitting garbled information.

If enabling of RTs is deferred upon entering realtime mode using `DQ_L553_RT_DEFER_EN` flag (i.e., upon VMaps are operational), use a `DQ_L553_RTS_RTEN` flag to write a mask of enabled RTs for each channel.

If the `DQ_L553_RT_RX_CMD` flag is selected, VMap operation adds command and status information associated with an Rx command before retrieving actual data. This information occupies four uint16 words and contains the following: command, status, command if RT-RT command was executed, RT-RT status. For communication from a BC to an RT, the second two words are zero. This way of retrieving received data is safer than using `DQ_L553_RT_STS0` (last command (higher 16 bits) and status (lower 16 bits) word) because there is no risk that the data has been updated between reading status information and actual data.

To enable an RT/SA address to become active, one needs to supply an <rtsa_list> list defining active terminals and sub-addresses:

```
DQ_L553_BM_SADDR(N)     (((N)&0x1f)<<0) // sub-address <reserved>
DQ_L553_BM_SADDR_SEL    (1L<<5)    // Sub-address field is valid <reserved>
DQ_L553_BM_RT(N)        (((N)&0x1f)<<6) // RT number
DQ_L553_BM_RT_SEL       (1L<<11)   // RT field is valid
```

These definitions are encapsulated in `DQ_L553_RT_SA(RT,SA)` macro.

If NULL is supplied as an <rtsa_list>, all RTs and all SAs are active and can be used to emulate a 1553 network during initial prototyping and BC debugging process.

<valid_list>
A validation list allows checking messages against specified parameters. Each word in the validation list is applied against RT/SA list with the same index.

```
SL553_MEMVAL_MD_TX_DW (1L<<28) // 1 when mode data word is expected and T/R_N bit set to 1
SL553_MEMVAL_MD_RX_DW (1L<<27) // 1 when mode data word is expected and T/R_N bit set to 0
SL553_MEMVAL_MD_TX_EN (1L<<25) // Enable corresponding mode command with T/R_N bit set to 1
SL553_MEMVAL_MD_RX_EN (1L<<24) // Enable corresponding mode command with T/R_N bit set to 0
SL553_MEMVAL_TX_IRQ_EN (1L<<23) // Enable TX IRQ on the selected sub-address
SL553_MEMVAL_RX_IRQ_EN (1L<<22) // Enable RX IRQ on the selected sub-address
SL553_MEMVAL_TX_EN (1L<<21) // Enable TX on the selected sub-address
SL553_MEMVAL_RX_EN (1L<<20) // Enable RX on the selected sub-address
SL553_MEMVAL_TX_WCMAX_MSB (1L<<19) // Maximum word count for the TX
SL553_MEMVAL_TX_WCMAX_LSB (1L<<15) // 0 = 32; 1-31 = 1-31
SL553_MEMVAL_TX_WCMIN_MSB (1L<<14) // Minimum word count for the TX
SL553_MEMVAL_TX_WCMIN_LSB (1L<<10) // 0 = 32; 1-31 = 1-31
SL553_MEMVAL_RX_WCMAX_MSB (1L<<9) // Maximum word count for the RX
SL553_MEMVAL_RX_WCMAX_LSB (1L<<5) // 0 = 32; 1-31 = 1-31
SL553_MEMVAL_RX_WCMIN_MSB (1L<<4) // Minimum word count for the RX
SL553_MEMVAL_RX_WCMIN_LSB (1L<<0) // 0 = 32; 1-31 = 1-31
```

If a request to receive or to transmit data cannot be validated against conditions stored in the validation list, an error and a status condition will be set and the command will be rejected.

### 4.43.9 DqAdv553SetRTWatchdog

**Syntax:**

```
int DAQLIB DqAdv553SetRTWatchdog(int hd, int devn, int channel, double
inactivity_s);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| double inactivity_s | Set up inactivity period on the bus |
| **Output** | |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |

| | |
|---|---|
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**

This function sets up an inactivity period on the bus for the watchdog. This feature is used to alert a user if the bus was idle for a longer period of time than expected. The watchdog flag is returned in the SL553_PORT_STS_WDR bit.

### *4.43.10 DqAdv553ConfigBufferRT*

**Syntax:**

```
int DAQLIB DqAdv553ConfigBufferRT(int hd, int devn, int channel, uint32 rt_sa,
uint32 rx_buf_size, uint32 rx_scan_size, uint32 tx_buf_size, uint32 tx_scan_size);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 rt_sa | RT and SA numbers (like in the RT/SA list) |
| uint32 rx_buf_size | Size of the receive buffer for this RT/SA |
| uint32 rx_scan_size | Scan size of the receive buffer for this RT/SA |
| uint32 tx_buf_size | Size of the transmit buffer for this RT/SA |
| uint32 tx_scan_size | Scan size of the transmit buffer for this RT/SA |
| **Output** | None |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description**

This function allocates contiguous buffers for the specified RT and SA (separate for RX and TX commands). This allows transfers of contiguous data arrays to and from the specified RT and SA. When buffers are allocated for either a receive or transmit operation, the interrupt upon RX or TX on this RT/SA is enabled. After the operation is over, ISR either retrieves or stores a new set of data of scan size from or into the allocated buffer.

`<rx_buf_size>` receive buffer size in uint16 words

`<rx_scan_size>` the size of one scan in the RX buffer – this is the number of words to be stored from the SA data area to the buffer upon an RX interrupt on this RT/SA combination.

`<tx_buf_size>` transmit buffer size in uint16 words

`<tx_scan_size>` the size of one scan in the TX buffer – this is the number of words to be written from the buffer to the SA data area to the buffer upon an RX interrupt.

If `<rx_buf_size>` is set to zero, the buffering function is disabled.

The data in the buffer can be presented in two ways:

1. If `<xx_scan_size>` is greater than zero, the size of SA data in the buffer is fixed.
2. If `<xx_scan_size>` is equal to zero, the first uint16 word defines the size of the data [1..32] to be placed into SA memory.

### *4.43.11 DqAdv553WriteRT*

**Syntax:**

```
int DAQLIB DqAdv553WriteRT(int hd, int devn, int channel, uint32 rt_size, uint32*
rtsa_list, uint16** data);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 rt_size | Size of the following RT/SA list |
| uint32* rtsa_list | Array of RT/SA to write to (see NOTE below for array size constraints) |
| uint16** data | Pointer to the array of pointers to the data for each entry in rtsa_list |
| **Output** | |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_NOT_ENOUGH_ROOM` | not enough room in the packet/structure |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**

This function writes data to the RT/SA memory, which is used to send out data upon receiving a Transmit command from a BC.

<`rt_size`> is a number of entries in <`rtsa_list`>. Each entry in this list is defined as follows:

```
DQ_L553_RT_SADDR(N)     (((N)&0x1f)<<0) // sub-address
DQ_L553_RT_SADDR_SEL    (1L<<5)         // Sub-address field is valid
DQ_L553_RT_RT(N)        (((N)&0x1f)<<6) // RT number
DQ_L553_RT_RT_SEL       (1L<<11)        // RT field is valid
DQ_L553_RT_SET_STATUS   (1L<<12)        // Set status bits
DQ_L553_RT_SET_BLK1     (1L<<13)        // Use Block 1 to write data
DQ_L553_RT_TX_SIZE(N)   (((N)&0x1f)<<16)   // Data size
```

Data size is defined in 16-bit words. A five-bit field `DQ_L553_RT_TX_SIZE(N)`defines the size of requested transfer. If it equals zero, all 32 words will be written into the corresponding memory area of that entry.

**NOTE:** The maximum <`rtsa_list`> array is limited by the Ethernet packet size. When using `DqAdv553WriteRT()` with the maximum size of data (32 16-bit entries per RT), each call to `DqAdv553WriteRT()` can access a maximum of 20 RTs.

If `DQ_L553_RT_SET_STATUS` flag is set, three special functions can be performed:

1. If SA is 0, 32-bit data (two 16-bit words) contains settings for bits in the RT status register.
2. If SA is 1, 32-bit data (two 16-bit words) contains a vector word (low is BIT word and high is Vector word) for RT vector register.

Instead of OR-ing bits to create an entry, we recommend that you use the macros described below.

`DQ_L553_RT_TX(RT,SA,SIZE)` – specify RT, SA and data size
`DQ_L553_RT_TX_BLK(RT,SA,SIZE,BLK)` – specify RT, SA, data size and block to write data to
`DQ_L553_RT_TX_STS(RT)` – specify RT to write status to and size
`DQ_L553_RT_TX_VECTOR(RT)` – specify RT to update vector word information for this RT

If a user would like to set status reply bits for the terminal, he should set flag `DQ_L553_RT_SET_STATUS` (or use `DQ_L553_RT_TX_STS(RT)`) and use the following bits:

```
SL553_RT_CFG0_TF     (1L<<5)   // =1 if terminal should always set TF bit in status word
SL553_RT_CFG0_SF     (1L<<4)   // =1 if terminal should always set SF bit in status word
SL553_RT_CFG0_BSY    (1L<<3)   // =1 if terminal should always set BUSY bit in status word
SL553_RT_CFG0_BSRE   (1L<<2)   // =1 if broadcast RX commands are accepted
SL553_RT_CFG0_SR     (1L<<1)   // =1 if SR bit should be set in the status word
```

<`data`> is an array of the pointers to 16-bit arrays of data allocated for each subsystem. This approach allows the user to allocate memory for each sub-address in use and then change data in the memory instead of re-assembling data every time he needs to update sub-address data.

### *4.43.12 DqAdv553ReadRT*

**Syntax:**

```
int DAQLIB DqAdv553ReadRT(int hd, int devn, int channel, uint32 rt_size, uint32*
rtsa_list, uint16** data);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 rt_size | Size of the following RT/SA list |
| uint32* rtsa_list | Array of RT/SA to read from (see NOTE below for array size constraints) |
| uint16** data | Pointer to arrays to store data from the specified sub-addresses |

| **Output** | |
|---|---|
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_NOT_ENOUGH_ROOM` | not enough room in the packet/structure |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**

This function reads data from the RT/SA memory, which is updated upon a Receive command from a BC.

<`rt_size`> is a number of entries in <`rtsa_list`>. Each entry in this list is defined as follows:

```
DQ_L553_RT_SADDR(N)    (((N)&0x1f)<<0) // sub-address
DQ_L553_RT_SADDR_SEL   (1L<<5)         // Sub-address field is valid
DQ_L553_RT_RT(N)       (((N)&0x1f)<<6) // RT number
DQ_L553_RT_RT_SEL      (1L<<11)        // RT field is valid
DQ_L553_RT_SET_STATUS  (1L<<12)        // Set status bits
DQ_L553_RT_SET_BLK1    (1L<<13)        // Use Block 1 to write data
DQ_L553_RT_TX_SIZE(N)  (((N)&0x1f)<<16)   // Data size
```

Data size is defined in 16-bit words. A five-bit field `DQ_L553_RT_TX_SIZE(N)`defines the size of the requested transfer. If it equals zero, all 32 words will be written into the corresponding memory area of that entry.

**NOTE:** The maximum <`rtsa_list`> array is limited by the Ethernet packet size. When using `DqAdv553ReadRT()` with the maximum size of data (32 16-bit entries per RT), each call to `DqAdv553ReadRT()` can access a maximum of 20 RTs.

We suggest using the following macros to form entries in the <`rtsa_list`> list:

`DQ_L553_RT_RX(RT,SA,SIZE)` – specify RT, SA and data size
`DQ_L553_RT_RX_BLK(RT,SA,SIZE,BLK)` – specify RT, SA, data size and block to read data from

DqAdv553ReadRT() has been updated to return data as well as status to avoid having to call DqAdv553ReadRT() and then DqAdv553ReadStautsRT().

We suggest using the following macros to form entries in the <`rtsa_list`> list if you would like to enable this functionality:

`DQ_L553_RT_RX_DATA_RDY(RT, BLK)` – retrieve data ready (i.e., data received by RT) word from the defined remote terminal and block. Retrieved data is a32-bit word that defines what subadresses received data. A read clears status bits.

`DQ_L553_RT_RX_DATA_SENT(RT, BLK)` – retrieve data sent (i.e., data transmitted by RT) word from the defined remote terminal and block. The retrieved data is a 32-bit word that defines what subadresses transmitted data. A read clears status bits.

`DQ_L553_RT_RX_PORT_STS(RT)` – retrieve most important status information (32-bit):

```
#define SL553_PORT_STS_BSF    (1L<<31)   // RT: Current bus that driving BM FIFO (1=BUS A)
#define SL553_PORT_STS_BSR    (1L<<30)   // RT: Current bus that driving RT (1=BUS A)
#define SL553_PORT_STS_ENB    (1L<<29)   // RT: Bus B Encoder is enabled in RT Engine
#define SL553_PORT_STS_ENA    (1L<<28)   // RT: Bus A Encoder is enabled in RT Engine
#define SL553_PORT_STS_DRB    (1L<<27)   // If set - data is ready at the decoder B
#define SL553_PORT_STS_DRA    (1L<<26)   // If set - data is ready at the decoder A
#define SL553_PORT_STS_WDR    (1L<<10)   // Watchdog request from RT: indicated no activity from the BC
within specified period
#define SL553_PORT_STS_BEB    (1L<<9)    // BUS: Bit timing error on bus B (sticky, auto-cleared after
read)
#define SL553_PORT_STS_BEA    (1L<<8)    // BUS: Bit timing error on bus A (sticky, auto-cleared after
read)
#define SL553_PORT_STS_PEB    (1L<<7)    // BUS: Parity error on bus B (sticky, auto-cleared after
read)
#define SL553_PORT_STS_PEA    (1L<<6)    // BUS: Parity error on bus A (sticky, auto-cleared after
read)
#define SL553_PORT_STS_ILC    (1L<<5)    // RT: Illegal command (sticky, auto-cleared after read)
#define SL553_PORT_STS_MER    (1L<<4)    // RT: Message error (sticky, auto-cleared after read)
#define SL553_PORT_STS_TMW    (1L<<3)    // RT: Too many words were detected in RX message (sticky,
auto-cleared after read)
#define SL553_PORT_STS_TFW    (1L<<2)    // RT: Too few words were detected inm RX message (sticky,
auto-cleared after read)
#define SL553_PORT_STS_ERB    (1L<<1)    // If set - data overflow was detected at the decoder B
(sticky, auto-cleared after read)
#define SL553_PORT_STS_ERA    (1L<<0)    // If set - data overfolw was detected at the decoder A
(sticky, auto-cleared after read)
```

`DQ_L553_RT_RX_SYNC(RT)` – retrieve last SYNC command (upper 16 bits of 32-bit word) and last transmitter shutdown/override command (lower 16-bits).

`DQ_L553_RT_RX_MODE(RT)` – retrieve last MODE command (lower 16-bits).

`<data>` is an array of the pointers to 16-bit arrays of data allocated for each subsystem. This approach allows a user to allocate memory for each sub-address in use and then change data in the memory instead of re-assembling data every time he needs to update sub-address data.

### *4.43.13 DqAdv553ReadStatusRT*

**Syntax:**

```
int DAQLIB DqAdv553ReadStatusRT(int hd, int devn, int channel, uint32 rt_size,
uint32* rtsa_list, uint32* data);
```

| **Input** | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number |
| `int channel` | Channel (0 or 1) |
| `uint32 rt_size` | Size of the following RT/SA list |
| `uint32* rtsa_list` | Array of RT/SA to read from and what to read |
| `uint32* data` | Pointer to status data received from RT |
| **Output** | |
| `uint32* data` | Status data received from RT, size = `rt_size` |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**

This function reads data from the RT/SA status memory, which holds RT/SA and channel bus status.

`<data>` is an array of 32-bit words, which is allocated to accommodate all data requested in provided RTSA list (`<rtsa_list>`).

`<rt_size>` is a number of entries in `<rtsa_list>`. Each entry in this list is defined as follows:

```
DQ_L553_RT_RT(N)     (((N)&0x1f)<<6) // RT number
DQ_L553_RT_STS0      (1L<<12)   // Last command (hi) and status (low) word
DQ_L553_RT_STS1      (2L<<12)   // Last SYNC (hi) and Transmitter (low) shutdown
DQ_L553_RT_STS2      (3L<<12)   // 32-bit timestamp when SYNC or SYNC+DW are received
DQ_L553_RT_CHSTAT    (4L<<12)   // Channel status, RT/SA ignored
DQ_L553_RT_BERRORS   (5L<<12)   // Bus errors, RT/SA ignored
DQ_L553_RT_DATA_RDY  (6L<<12)   // RT status that data has been received (BC->RT)
DQ_L553_RT_DATA_SENT (7L<<12)   // RT status that data has been send (BC<-RT)
```

`DQ_L553_CHSTAT` has the following bits available:

```
#define SL553_PORT_STS_BSF    (1L<<31)   // RT: Current bus that driving BM FIFO (1=BUS A)
#define SL553_PORT_STS_BSR    (1L<<30)   // RT: Current bus that driving RT (1=BUS A)
#define SL553_PORT_STS_ENB    (1L<<29)   // RT: Bus B Encoder is enabled in RT Engine
#define SL553_PORT_STS_ENA    (1L<<28)   // RT: Bus A Encoder is enabled in RT Engine
#define SL553_PORT_STS_DRB    (1L<<27)   // If set - data is ready at the decoder B
#define SL553_PORT_STS_DRA    (1L<<26)   // If set - data is ready at the decoder A
#define SL553_PORT_STS_WDR    (1L<<10)   // Watchdog request from RT: indicated no activity from the BC
within specified period
#define SL553_PORT_STS_BEB    (1L<<9)    // BUS: Bit timing error on bus B (sticky, auto-cleared after
read)
#define SL553_PORT_STS_BEA    (1L<<8)    // BUS: Bit timing error on bus A (sticky, auto-cleared after
read)
#define SL553_PORT_STS_PEB    (1L<<7)    // BUS: Parity error on bus B (sticky, auto-cleared after
read)
#define SL553_PORT_STS_PEA    (1L<<6)    // BUS: Parity error on bus A (sticky, auto-cleared after
read)
#define SL553_PORT_STS_ILC    (1L<<5)    // RT: Illegal command (sticky, auto-cleared after read)
#define SL553_PORT_STS_MER    (1L<<4)    // RT: Message error (sticky, auto-cleared after read)
#define SL553_PORT_STS_TMW    (1L<<3)    // RT: Too many words were detected in RX message (sticky,
auto-cleared after read)
#define SL553_PORT_STS_TFW    (1L<<2)    // RT: Too few words were detected in RX message (sticky,
auto-cleared after read)
#define SL553_PORT_STS_ERB    (1L<<1)    // If set - data overflow was detected at the decoder B
(sticky, auto-cleared after read)
#define SL553_PORT_STS_ERA    (1L<<0)    // If set - data overflow was detected at the decoder A
(sticky, auto-cleared after read)
```

`DQ_L553_BERRORS` bits (possible bus errors) are defined as follows:

```
#define SL553_PORT_IR_RWD    (1L<<31) // RT: watchdog IRQ upon inactivity on the bus for the specified
period
#define SL553_PORT_IR_OFH    (1L<<30) // FIFO: Output FIFO half-full, (below watermark position)
#define SL553_PORT_IR_OFE    (1L<<29) // FIFO: Output FIFO Empty
#define SL553_PORT_IR_IFH    (1L<<28) // FIFO: Input FIFO half-full, (above watermark position)
#define SL553_PORT_IR_IFF    (1L<<27) // FIFO: Input FIFO full, possibly overwritten
#define SL553_PORT_IR_ITA    (1L<<26) // Bus: One of the following errors was detected on bus A/B :
invalid bit value
#define SL553_PORT_IR_ITB    (1L<<25) // Bus: (invalid Manchester code during SYNC or data bit time) or
parity error
#define SL553_PORT_IR_BEA    (1L<<24) // Bus: Bit timing error was detected on bus A/B, one of the
following errors: invalid combination on the input,
#define SL553_PORT_IR_BEB    (1L<<23) // Bus: rising/falling edge timing is invalid, bit timing timeout
detected
#define SL553_PORT_IR_MED    (1L<<22) // Bus: Message error detected - data error, suppress status
#define SL553_PORT_IR_TMW    (1L<<21) // Bus: Too many words were received within RX message, suppress
status
#define SL553_PORT_IR_TFW    (1L<<20) // Bus: Too few words were received within RX message, suppress
status
#define SL553_PORT_IR_ICD    (1L<<19) // Bus: Message error detected - illegal/illogical command error,
send status
#define SL553_PORT_IR_BCH    (1L<<18) // Bus: Bus was changed (command arrived on different bus)
#define SL553_PORT_IR_CDA    (1L<<17) // Bus: Command/data received on A bus
#define SL553_PORT_IR_CDB    (1L<<16) // Bus: Command/data received on B bus
#define SL553_PORT_IR_BCR    (1L<<15) // Bus: Broadcast command received
#define SL553_PORT_IR_BDR    (1L<<14) // Bus: Broadcast data received
#define SL553_PORT_IR_RTD    (1L<<13) // Bus: RTRT Timeout detected
#define SL553_PORT_IR_RVF    (1L<<12) // Bus: RTRT Validation failed
#define SL553_PORT_IR_DWG    (1L<<11) // Bus: Gap within data word detected (use only when xxRTR_DGP>0)
#define SL553_PORT_IR_DOA    (1L<<10) // Logic: 1553 bus A/B data was overwritten -  critical IRQ
indicates that main state machine
#define SL553_PORT_IR_DOB    (1L<<9)  // Logic: stuck somewhere and new 1553 command/data arrived
before it previous processed
#define SL553_PORT_IR_EMT    (1L<<8)  // Logic: External memory access timeout
#define SL553_PORT_IR_SDW    (1L<<7)  // Mode command: Synchronize with data word received on one of
the terminals IRQ request
```

```
#define SL553_PORT_IR_STS    (1L<<6)  // Mode command: Selected transmitter shutdown received with data
word IRQ request
#define SL553_PORT_IR_OTS    (1L<<5)  // Mode command: Override selected transmitter shutdown received
with data word IRQ request
#define SL553_PORT_IR_DBC    (1L<<4)  // Mode command: Dynamic bus change request received in mode
command
#define SL553_PORT_IR_DBA    (1L<<3)  // Mode command: Dynamic bus change request accepted
#define SL553_PORT_IR_MDS    (1L<<2)  // Mode command: SYNCHRONIZE request received in mode command
#define SL553_PORT_IR_ITF    (1L<<1)  // Mode command: Inhibit terminal flag
#define SL553_PORT_IR_RRT    (1L<<0)  // Mode command: Reset remote terminal by mode command (all RTs
are reset)
```

`DQ_L553_RT_DATA_RDY` and `DQ_L553_RT_DATA_SENT` return `<data>` as a bit mask that indicates which SA has data (`DQ_L553_RT_DATA_RDY`) or sent data (`DQ_L553_RT_DATA_SENT`).

- The first bit set to 1 indicates that SA 0 received or sent data.
- The second bit set to 1 indicates that SA 1 received or sent data
- etc.

### *4.43.14 DqAdv553ConfigBC*

**Syntax:**

```
int DAQLIB DqAdv553ConfigBC(int hd, int devn, int channel, uint32 bc_mode, uint32
option_flags, double MJ_clock, double MN_clock);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 bc_mode | What bus to use: bus A, bus B or both buses |
| uint32 option_flags | BC configuration flags |
| double MJ_clock | Major clock, Hz (in 10 µHz increments up to 42950 Hz) |
| double MN_clock | Minor clock, Hz (in 10 µHz increments up to 42950 Hz) |
| **Output** | |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**

This function defines how to configure a Bus Controller:

<bc_mode> defines what bus to listen to:

```
#define DQ_L553_BC_USE_A    (1L<<0) // enable bus A
#define DQ_L553_BC_USE_B    (1L<<1) // enable bus B
```

<bc_mode> allows you to enable either bus A or B or both.

<option_flags> defines additional operational flags for the Bus Controller:

Minimum bus idle delay prior to sending data to the bus in us; default delay is 50us if this field is left empty.

```
#define SL553_PORT_BCCFG_BCIDLE(N)   (((N)&0xff)<<16)
```

Disables wait for the bus to "clear" after error, if set — ignores BCMAX delay and will only wait until BIT delay expires, this bit only affects BC behavior after the error, the SL553_PORT_BCMAX register should still be programmed to the expected time of longest single transaction on the 1553 bus in the current configuration.

```
#define SL553_PORT_BCCFG_DISWT (1L<<15)
```

Use CLI CL clock as the major clock; BCMJD will be disabled. This flag is required if the major frame clock is derived from the external source over SYNCx lines.

```
#define SL553_PORT_BCCFG_EMJC (1L<<14)
```

Use CLI CV clock as the minor clock; BCMRD will be disabled.  This flag is required if the major frame clock is derived from the external source over SYNCx lines.

```
#define SL553_PORT_BCCFG_EMNC (1L<<13)
```

Bus Controller verification parameter; late response limit in uS; default is 20us if the field is left empty.

```
#define SL553_PORT_BCCFG_BCLATE(N)     (((N)&0x1f)<<8)
```

Enable debug FIFO to monitor all commands sent on the bus by the Bus Controller.

```
#define SL553_PORT_BCCFG_BCRTE (1L<<5)
```

Define early response limit in uS

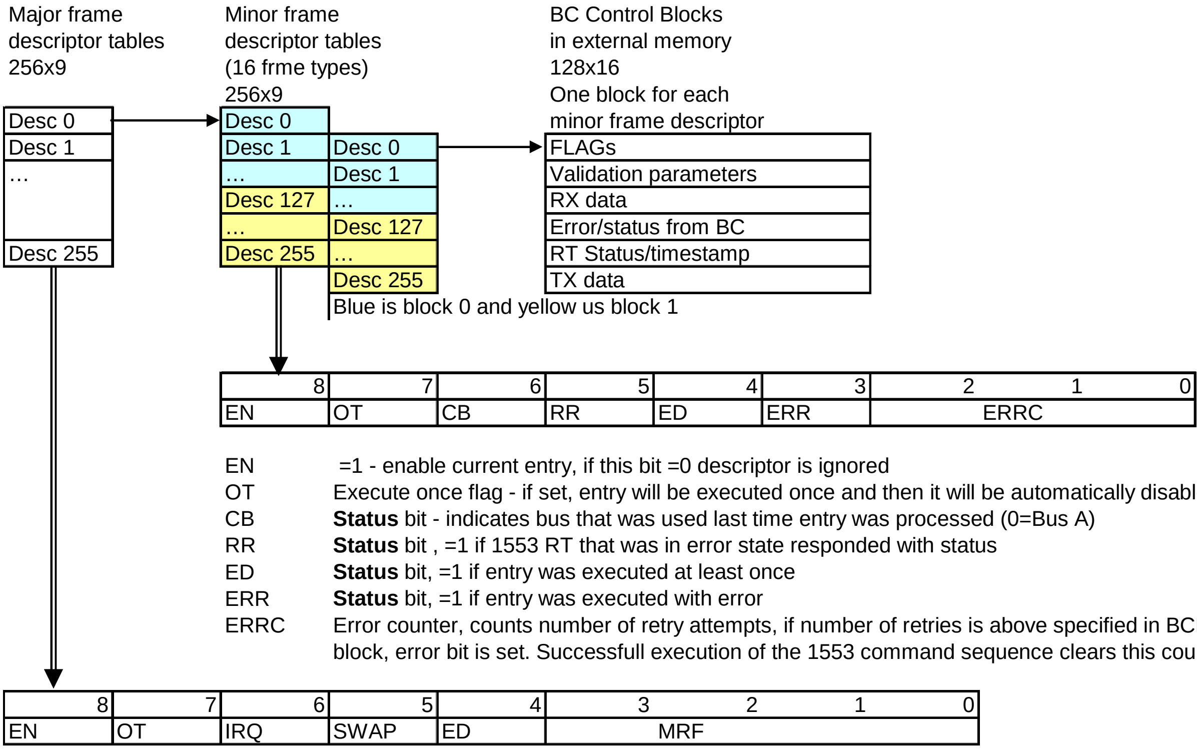
```
#define SL553_PORT_BCCFG_BCEARLY(N)     (((N)&0x1f))
```

<MJ_clock> is the frequency of the major frame clock in Hz. Frequency can be between 10 µHz and 42950 Hz in 10 µHz increments.
<MN_clock> is the frequency of the minor frame clock in Hz. Frequency can be between 10 µHz and 42950 Hz in 10 µHz increments.

A Bus Controller organizes data exchange into major and minor frames.
Each major frame can consist of 256 minor frames inserted in an arbitrary order. However, there are only sixteen types of minor frames. This organization is typical for MIL-1553 bus operation when minor frames are executed within major frames at different update frequencies. To perform an update every 50ms (frame 1), 25ms (frame 2) and 12.5ms (frame 3), only three types of frames are required arranged in the sequence {1, 3, 2, 3, empty, 3, 2, 3}. In this arrangement, a major frame clock should be set to 20Hz (50ms) and a minor clock set to 160Hz (6.25ms).

Each minor frame contains up to 256 bus controller command blocks (BCCBs), which are divided into upper and lower parts and used as a double-buffered buffer to ensure that data has not been changed by the host software when a bus controller executes the frame.

The frame structure can be represented as follows:



| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| EN | OT | CB | RR | ED | ERR | ERRC | | |

| | |
|---|---|
| EN | =1 - enable current entry, if this bit =0 descriptor is ignored |
| OT | Execute once flag - if set, entry will be executed once and then it will be automatically disabled. |
| CB | **Status** bit - indicates bus that was used last time entry was processed (0=Bus A) |
| RR | **Status** bit , =1 if 1553 RT that was in error state responded with status |
| ED | **Status** bit, =1 if entry was executed at least once |
| ERR | **Status** bit, =1 if entry was executed with error |
| ERRC | Error counter, counts number of retry attempts, if number of retries is above specified in BCB block, error bit is set. Successfull execution of the 1553 command sequence clears this counter |

| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| EN | OT | IRQ | SWAP | ED | MRF | | | |

| | |
|---|---|
| EN | Enable descriptor, if this bit =0 descriptor is ignored and next entry is processed, until end of the descriptor table.<br>Descriptor may be temporary disabled in order to replace data inside of the minor frame and/or minor frame descriptors |
| OT | Execute once flag - if set, entry will be executed once and then it will be automatically disabled.<br>May be used for the minor frame filled with aperiodic messages |
| IRQ | Issue IRQ upon completion of the minor frame execution (SL553_PORT_BCI_UIRQ) |
| SWAP | Enable entry if disabled or disable if enabled upon SWAP request; SWAP request allows simultaneous change of the sequence in the major frame descriptor table at the beginning of the major frame cycle<br>Write to SL553_PORT_BCSWP (0x21A4/0x28A4) initiates SWAP cycle upon next major frame clock |
| ED | **Status** bit, =1 if entry was executed at least once |
| MRF | Minor frame ID (0-15) |

### 4.43.15 DqAdv553WriteMJDescriptors

**Syntax:**

```
int DAQLIB DqAdv553WriteMJDescriptors(int hd, int devn, int channel, uint32 index,
int size, uint32* descriptors);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 index | Descriptor index in the major frame table |
| int size | Number of descriptors to write |
| uint32* descriptor | Array of descriptors |
| **Output** | |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**

This function programs a major frame descriptor table of <size> from the data array supplied starting from the <index> supplied. Major frame descriptor bits are defined as follows:

Enable current frame. If this bit =0, the descriptor is ignored and next entry is processed, until the end of the descriptor table is reached. The descriptor may be temporarily disabled in order to replace data inside minor frame and/or minor frame descriptors

```
#define SL553_MJF_DESC_EN   (1L<<8)
```

Execute once flag.  If set, the entry will be executed once and then it will be automatically disabled. May be used for the minor frame filled with aperiodic messages.

```
#define SL553_MJF_DESC_OT   (1L<<7)
```

Issue IRQ upon completion of the minor frame execution (this bit is overwritten by the firmware).

```
#define SL553_MJF_DESC_IRQ  (1L<<6)
```

Enable entry if disabled or disable if enabled upon SWAP request.A  SWAP request allows simultaneous change of the sequence in the major frame descriptor table at the beginning of the major frame cycle.

```
#define SL553_MJF_DESC_EN_SWAP  (1L<<5)
```

Status bit =1 if entry was executed at least once (status only, write zero to clear)

```
#define SL553_MJF_DESC_ED   (1L<<4)
```

Minor frame ID to execute [0..15]

```
#define SL553_MJF_DESC_MRF(N)   ((N)&0xf)
```

### 4.43.16 DqAdv553WriteMNDescriptors

**Syntax:**

```
int DAQLIB DqAdv553WriteMNDescriptors(int hd, int devn, int channel, uint32
mn_frame, uint32 mn_block, uint32 index, int size, uint32* descriptors);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 mn_frame | Minor frame number [0..15] |
| uint32 mn_block | Minor frame block number (0 or 1 or both) to store data |
| uint32 index | Descriptor index in the minor frame table |
| int size | Number of descriptors to write |
| uint32* descriptor | Array of descriptors |
| **Output** | |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**

This function stores a minor frame descriptor table of <size> from the data array supplied starting from the <index> supplied. Minor frame descriptor bits are defined as follows:

Enable current entry. If this bit =0, the descriptor is ignored and next entry is processed, until the end of the descriptor table is reached.

```
#define SL553_MNF_DESC_EN    (1L<<8)
```

Execute once flag. If set, the entry will be executed once and then it will be automatically disabled. This flag can be used for the aperiodic messages; the enable flag is not cleared if sn error was detected during the execution of the entry.

```
#define SL553_MNF_DESC_OT    (1L<<7)
```

Status bit, report currently active 1553 bus for this entry, zero if side A is used (status only, write zero to clear)

```
#define SL553_MNF_DESC_CB    (1L<<6)
```

Status bit, =1 if 1553 RT that was in error state responded with status (status only, write zero to clear)

```
#define SL553_MNF_DESC_RR    (1L<<5)
```

Status bit, =1 if entry was executed at least once (status only, write zero to clear)

```
#define SL553_MNF_DESC_ED    (1L<<4)
```

Status bit, =1 if entry was executed with error (status only, write zero to clear)

```
#define SL553_MNF_DESC_ERR   (1L<<3)
```

Bits [2..0] are an error counter that counts number of retry attempts; if number of retries is above th value specified in the control block, an error bit is set. Successful execution of the 1553 command sequence clears this counter.

<mn_frame> - minor frame number from 0 to 15.

<mn_block> - minor frame block number. Each minor frame has two parts containing 128 descriptors each. This organization was selected for the ability to read or replace data in the currently inactive part of each frame and then swap them simultaneously. You can select to write to both blocks simultaneously. When BC operation becomes enabled, it uses part zero of each frame until told to swap it.

```
#define SL553_MNF_BLOCK1   (1L<<1)   // block 1 of a minor frame
#define SL553_MNF_BLOCK0   (1L<<0)   // ditto block 0
```

<index> - index of the minor frame descriptor to start writing a <size> of descriptors in the supplied array.

## 4.43.17 DqAdv553ReadMNDescriptors

**Syntax:**

```
int DAQLIB DqAdv553ReadMNDescriptors(int hd, int devn, int channel, uint32
mn_frame, uint32 mn_block, uint32 index, int size, uint32* descriptors, uint32*
mj_position, uint32* mn_position);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 mn_frame | Frame number [0..15] |
| uint32 mn_block | Block number (0 or 1 or both) to store data |
| uint32 index | Descriptor index in the minor frame table |
| int size | Number of descriptors to read |
| **Output** | |
| uint32* descriptor | Array to store descriptors |
| uint32* mj_position | Currently executed descriptor in the major frame |
| uint32* mn_position | Currently executed descriptor in the minor frame |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**

This function allows you to read back minor frame descriptors. This function returns descriptors that contain execution status flags as well as currently executed descriptor indices in the major and minor frames.

There are a few status bits stored in the minor frame descriptor after it is processed:
a status bit reports the currently active 1553 bus for this entry, zero if side A is used (status only, write zero to clear)

```
#define SL553_MNF_DESC_CB    (1L<<6)
```

Status bit, =1 if the 1553 RT that was in error state responded with status (status only, write zero to clear)

```
#define SL553_MNF_DESC_RR    (1L<<5)
```

Status bit, =1 if entry was executed at least once (status only, write zero to clear)

```
#define SL553_MNF_DESC_ED    (1L<<4)
```

Status bit, =1 if entry was executed with error (status only, write zero to clear)

```
#define SL553_MNF_DESC_ERR   (1L<<3)
```

Bits [2..0] are an error counter that counts the number of retry attempts. If the number of retries is above the value specified in the control block, an error bit is set. Successful execution of the 1553 command sequence clears this counter.

<mn_frame> - minor frame number from 0 to 15.

<mn_block> - minor frame block number. Each minor frame has two parts containing 128 descriptors each. Only one block can be read at a time.

```
#define SL553_MNF_BLOCK1   (1L<<1)   // block 1 of a minor frame
#define SL553_MNF_BLOCK0   (1L<<0)   // ditto block 0
```

<index> - index of the minor frame descriptor to start writing a <size> of descriptors in the supplied array.

### *4.43.18 DqAdv553WriteBCCB*

**Syntax:**

```
int DAQLIB DqAdv553WriteBCCB(int hd, int devn, int channel, uint32 mn_frame,
uint32 mn_block, uint32 index, uint32 size, pBCCB_Control descriptor);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 mn_frame | Minor frame number [0..15] |
| uint32 mn_block | Minor frame block number (0 or 1 or both) to store data |
| uint32 index | Descriptor index in the minor frame table |
| uint32 size | Number of descriptors to write |
| pBCCB_Control descriptor | Array of BCCB descriptors |
| **Output** | |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description**

This function stores BCCB descriptors for the minor frame and slot selected.
A BCCB descriptor has the following structure:

```
typedef struct {
    uint16 flags0;    // Address of the FLAGS0 word in the BC control block
    uint16 flags1;    // Address of the FLAGS1 word in the BC control block
    uint16 rsv2;
    uint16 rsv3;
    uint16 cmd1;      // First 1553 command word
    uint16 cmd2;      // Second 1553 command word
    uint16 sts1_or;   // "OR" mask for the first status word
    uint16 sts1_and;  // "AND" mask for the first status word
    uint16 sts1_val;  // Compare "VALUE" for the first status word
    uint16 sts2_or;   // "OR" mask for the second status word
    uint16 sts2_and;  // "AND" mask for the second status word
    uint16 sts2_val;  // Compare "VALUE" for the second status word
    uint16 rsv12;
    uint16 rsv13;
    uint16 rsv14;
    uint16 rsv15;
    uint16 rx_data_tmin[32];  // RX data or minimum compare values for TX
    uint16 tmax[32];        // maximum compare values for TX
    // 80 uint16s total
} BCCB_Control, *pBCCB_Control;
```

<rx_data_tmin> contains actual data to be sent by the bus controller. Use DqAdv553BCWriteRXData() if you would like to store this data without the need to transfer the whole BCCB_Control structure.

Fields <flags0> and <flags1> define how to execute a 1553 command:
<flags0>:

```
#define BC1553_BCB_FLAGS0_IRQ   (1L<<15) // =1 if interrupt should be requested immediately
                                         // upon completion of the execution
#define BC1553_BCB_FLAGS0_BEN   (1L<<14) // =1 - enable communication with RTs on Bus B
#define BC1553_BCB_FLAGS0_AEN   (1L<<13) // =1 - enable communication with RTs on Bus A
#define BC1553_BCB_FLAGS0_EXW   (1L<<12) // =1 - allow extended wait for response from RT
                                         // (double maximum response time)
#define BC1553_BCB_FLAGS0_RSV11 (1L<<11) // Reserved
#define BC1553_BCB_FLAGS0_RSV10 (1L<<10) // Reserved
#define BC1553_BCB_FLAGS0_RSV9  (1L<<9)  // Reserved
#define BC1553_BCB_FLAGS0_SBUS  (1L<<8)  // Select "Start Bus" - bus that will be initially
                                         // used to communicate with RT (0=Bus A)
#define BC1553_BCB_FLAGS0_DTC   (1L<<7)  // =1 - check returned value of the data from RT
                                         // (compare with MIN and MAX value)
#define BC1553_BCB_FLAGS0_MDC   (1L<<6)  // =1 - check returned value of the mode data word
                                         // (apply AND/OR masks and compare with VAL value
#define BC1553_BCB_FLAGS0_STS2  (1L<<5)  // =1 - check returned value of the 2nd status word
                                         // (apply AND/OR masks and compare with VAL value
#define BC1553_BCB_FLAGS0_STS1  (1L<<4)  // =1 - check returned value of the first
                                         // status word
                                         // (apply AND/OR masks and compare with VAL value
#define BC1553_BCB_FLAGS0_TT(N) ((N)&0xf) // Transfer type - selects one of the available
                                         // 1553 sequences
```

Transfer types:

```
#define BC1553_BCB_TT_BCRT_1E    1 // BC-RT ("1e" sequence of the BC state machine)
#define BC1553_BCB_TT_RTBC_2B    2 // RT-BC ("2b" sequence of the BC state machine)
#define BC1553_BCB_TT_RTRT_3A    3 // RT-RT ("3a" sequence of the BC state machine)
#define BC1553_BCB_TT_MD_1A      4 // MODE W/O Data (TX)
#define BC1553_BCB_TT_MDTX_2A    5 // MODE W Data (TX)
#define BC1553_BCB_TT_MDRX_1C    6 // MODE W Data (RX)
#define BC1553_BCB_TT_BBCRT_1F   7 // Broadcast BC-RT
#define BC1553_BCB_TT_BRTRT_3B   8 // Broadcast RT-RT
#define BC1553_BCB_TT_BMD_1B     9 // Broadcast MODE W/O Data (TX)
#define BC1553_BCB_TT_BMDRX_1D 10 // Broadcast MODE W Data (TX)
```

Most of the transfer types require only one command to be sent and one status to be received. RT-RT (regular `BC1553_BCB_TT_RTRT_3A` and broadcast `BC1553_BCB_TT_BRTRT_3B`) commands are exclusions from this list. To execute one of these commands, a bus controller issues a receive command to one remote terminal following a transmit command to another terminal. Then, the transmitting terminal replies with the status followed by the data words and the receiving terminal confirms reception by issuing a status word (only in `BC1553_BCB_TT_RTRT_3A` case).

When these commands are used, both of the <cmd1> and <cmd2> fields need to be filled with the proper commands. The received status can be checked using `<sts1_or>`, `<sts1_and>` and `<sts1_val>` fields. First, the received status is OR-ed with `<sts1_or>`, then AND-ed with `<sts1_and>` and then compared with the `<sts1_val>` value. If the values do not match, error status BC1553_BCB_ERRSTS0_S1F is set up (in the <err_sts1> field of BCCB_Status structure – see `DqAdv553BCReadBCCB`. The same logic applies to the RT-RT situation when the second command, second compare conditions, and second status are used.

For reference, we have summarized in the following table the various types of 1553 commands supported by a DNR-553-1553 bus controller:

| Command1553 | Code | Cmd | | | | | | | | | | | Case |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RT/SA addressed Commands | | | | | | | | | | | | | |
| Rx BC->RT | 1 | RX | $D_0$ | … | $D_N$ | * | St | *gap* | | | | | 1e |
| Tx RT->BC | 2 | TX | * | St | $D_0$ | … | $D_N$ | *gap* | | | | | 2b |
| RxTx RT->RT | 3 | RX | TX | * | StTx | $D_0$ | … | $D_N$ | * | StRx | *gap* | | 3a |
| TXM$_0$ | 4 | MC | * | St | *gap* | | | | | | | | 1a |
| TXM$_1$ | 5 | MC | * | St | Data | *gap* | | | | | | | 2a |
| RXM | 6 | MC | $D_0$ | * | St | *gap* | | | | | | | 1c |
| Broadcast Commands | | | | | | | | | | | | | |
| Rx BC->RT | 7 | RX | $D_0$ | … | $D_N$ | *gap* | | | | | | | 1f |
| RxTx RT->RT | 8 | RX | TX | * | StTx | $D_0$ | … | $D_N$ | *gap* | | | | 3b |
| TXM$_0$ | 9 | MC | *gap* | | | | | | | | | | 1b |
| RXM | 10 | MC | $D_0$ | *gap* | | | | | | | | | 1d |

RX – receive command
TX – transmit command
$D_0$ … $D_N$ – 16-bit data words (from 1 to 32 words are allowed)
MC – mode command
St – status word

StRx – status issued by the receiving terminal
StTx – status issued by the transmitting terminal
* - time between bus controller command and terminal response (standard calls for 12uS)
*gap* – a gap between messages when both lines are low

<flags1> controls retry behavior as defined in MIL1553 protocol if an error during command execution was encountered:

```
#define BC1553_BCB_FLAGS1_RSV15 (1L<<15) // Reserved
#define BC1553_BCB_FLAGS1_IRT   (1L<<14) // Retry on incorrect RT# in status
#define BC1553_BCB_FLAGS1_RUS   (1L<<13) // Retry on unexpected status reception
#define BC1553_BCB_FLAGS1_RUD   (1L<<12) // Retry on unexpected data reception
#define BC1553_BCB_FLAGS1_RWB   (1L<<11) // Retry on wrong bus response
#define BC1553_BCB_FLAGS1_RIS   (1L<<10) // Retry on illegal bits set in status
#define BC1553_BCB_FLAGS1_RBB   (1L<<9)  // Retry on busy bit in status
#define BC1553_BCB_FLAGS1_RTE   (1L<<8)  // Retry on bus timing error
#define BC1553_BCB_FLAGS1_RWC   (1L<<7)  // Retry on word count
#define BC1553_BCB_FLAGS1_RME   (1L<<6)  // Retry on message error bit in status
#define BC1553_BCB_FLAGS1_RNR   (1L<<5)  // Retry on no-response
#define BC1553_BCB_FLAGS1_ERE   (1L<<4)  // Enable re-transmit on alternative bus each retry
#define BC1553_BCB_FLAGS1_ESR   (1L<<3)  // Enable periodic status request command
                                         // when count reached maximum allowable limit
#define BC1553_BCB_FLAGS1_ERRC(N) ((N)&7)  // Specify maximum number of retry attempts
```

### *4.43.19 DqAdv553ReadBCCB*

**Syntax:**

```
int DAQLIB DqAdv553ReadBCCB(int hd, int devn, int channel, uint32 mn_frame, uint32
mn_block, uint32 index, uint32 size, pBCCB_Status descriptor);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 mn_frame | Minor frame number [0..15] |
| uint32 mn_block | Minor frame block number (0 or 1 or both) to store data |
| uint32 index | Descriptor index in the minor frame table |
| uint32 size | Number of descriptors to read |
| `pBCCB_Status` descriptor | Array of BCCB descriptors |
| **Output** | |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description**

This function reads the status part of BCCB descriptors for the minor frame and slot selected. A BCCB status part has the following structure:

```
typedef struct {
    uint16 sts1;        // First status reply from the RT
    uint16 sts2;        // Second status reply from the RT
    uint16 tsmsb;       // Timestamp of last access to the RT, 16 MSBs
    uint16 tslsb;       // Timestamp of last access to the RT, 16 LSBs
    uint16 rsv84;
    uint16 rsv85;
    uint16 errsts0;     // error status 0
    uint16 errsts1;     // error status 1
    uint16 rx_data[32]; // transmit data
    // 40 uint16s total
} BCCB_Status, *pBCCB_Status;
```

Together with `BCCB_Control`, the Bus Controller control block occupies 120 16-bit words.

```
typedef struct {
    BCCB_Control control;
    BCCB_Status status;
} BCCB, *pBCCB;
```

<errsts0> contains information of the completion of this entry. It holds the status of the entry processing, written by the BC state machine:

```
#define BC1553_BCB_ERRSTS0_CB    (1L<<15) // Status : Current bus (0 = Bus A)
#define BC1553_BCB_ERRSTS0_RSV14 (1L<<14) // Reserved, use as needed
#define BC1553_BCB_ERRSTS0_PD    (1L<<13) // Error : descriptor is permanently disabled
#define BC1553_BCB_ERRSTS0_S1B   (1L<<12) // Warning : Status 1 RT responded with BUSY
#define BC1553_BCB_ERRSTS0_S2B   (1L<<11) // Warning : Status 2 RT responded with BUSY
#define BC1553_BCB_ERRSTS0_BNR   (1L<<10) // Error : No response from bus
#define BC1553_BCB_ERRSTS0_RCR   (1L<<9) // Error : Retry count reached (clear to re-start)
#define BC1553_BCB_ERRSTS0_WBR   (1L<<8) // Error : Response received on the wrong bus
                                         // (also will set WBR bit in PORT_STS)
#define BC1553_BCB_ERRSTS0_IRT   (1L<<7) // Error : Incorrect RT responded (also will set
                                         // IRT bit in PORT_STS)
#define BC1553_BCB_ERRSTS0_LR    (1L<<6) // Warning : Late response
#define BC1553_BCB_ERRSTS0_ER    (1L<<5) // Error : Early response
#define BC1553_BCB_ERRSTS0_S2F   (1L<<4) // Error : Status 2 compare failed
#define BC1553_BCB_ERRSTS0_S1F   (1L<<3) // Error : Status 1 compare failed
#define BC1553_BCB_ERRSTS0_DCF   (1L<<2) // Error : Data compare failed
#define BC1553_BCB_ERRSTS0_TFW   (1L<<1) // Error : Too few words received (also will set
                                         // TFW bit in PORT_STS)
#define BC1553_BCB_ERRSTS0_TMW   (1L<<0) // Error : Too many words received (also will set
                                         // TMW bit in PORT_STS)
```

### 4.43.20 DqAdv553ReadBCStatus

**Syntax:**

```
int DAQLIB DqAdv553ReadBCStatus(int hd, int devn, int channel, int list_size,
uint32* list, uint32* status);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32* list | Status request list |
| int list_size | Status request list size |
| **Output** | |
| uint32* status | Array to store status information of list_size size |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**

This function allows a user to flexibly request status information for bus controller entries of interest. This is a combined function; everything except bus controller status can be retrieved via `DqAdv553ReadMNDescriptor()` and `DqAdv553ReadBCCB()`.

The following status bits can be obtained:

- Bus controller status:

```
#define DQ_SL553_BC_STATUS      (2L<<12)    // BCCB status/control
```

Use the following macro to select what status to retrieve:

```
#define DQ_SL553_BC_VMAP(TYPE, MN, IDX)    ((TYPE)|(((MN)&0xf)<<8)|((IDX)&0xff))
```

Use the following modifier in the MN field

```
#define DQ_SL553_BC_STATUS_BC   1 // return status of BC controllers (register 0x21A4)
```

Returns 32-bit word with the following bits

```
#define SL553_PORT_BCSTS_BSY    (1L<<31) // =1 if BC not in IDLE or WAIT_CLOCK states
#define SL553_PORT_BCSTS_RSV30  (1L<<30) // Reserved
#define SL553_PORT_BCSTS_RSV29  (1L<<29) // Reserved
#define SL553_PORT_BCSTS_RSV28  (1L<<28) // Reserved
#define SL553_PORT_BCSTS_MRF3   (1L<<27) // Reports ID for the currently processed
#define SL553_PORT_BCSTS_MRF0   (1L<<24) // minor frame
#define SL553_PORT_BCSTS_DSC7   (1L<<23) // Reports ID for the currently processed
#define SL553_PORT_BCSTS_DSC0   (1L<<16) // descriptor minor frame
                                         // 12-0 are sticky bits) auto-cleared
#define SL553_PORT_BCSTS_BIR    (1L<<12) // Response received from incorrect RT
#define SL553_PORT_BCSTS_BTO    (1L<<11) // Maximum 1553 access time exceeded
#define SL553_PORT_BCSTS_BAD    (1L<<10) // Bus activity is detected while in wait for clock (idle)
state
#define SL553_PORT_BCSTS_BWB    (1L<<9)  // Response received on the wrong bus
#define SL553_PORT_BCSTS_BCO    (1L<<8)  // BC overrun - major/minor frame clock called in non-idle
state
#define SL553_PORT_BCSTS_HBT    (1L<<7)  // Heartbeat - set to one at the beginning of each minor frame
#define SL553_PORT_BCSTS_MTD    (1L<<6)  // Memory access timeout was detected (hardware error)
#define SL553_PORT_BCSTS_RTR    (1L<<5)  // At least one RT in "dead" state replied with valid status
#define SL553_PORT_BCSTS_MF     (1L<<4)  // At least one Mode command w/o data word failed
#define SL553_PORT_BCSTS_MRBF   (1L<<3)  // At least one Mode TX command with data word failed
#define SL553_PORT_BCSTS_MBRF   (1L<<2)  // At least one Mode RX command with data word failed
#define SL553_PORT_BCSTS_RBF    (1L<<1)  // At least one RT->BC transmission failed
#define SL553_PORT_BCSTS_BRF    (1L<<0)  // At least one BC->RT transmission failed

#define SL553_PORT_BCSTS_MRF(S) (((S)&0xf000000)>>24)   // processed minor frame
#define SL553_PORT_BCSTS_DSC(S) (((S)&0xff0000)>>16)   // processed descriptro in minor frame
```

- Major frame descriptor status

```
#define DQ_SL553_BC_STATUS      (2L<<12)    // BCCB status/control
```

Use the following modifier in MN field

```
#define DQ_SL553_BC_STATUS_MJ   2 // return status of MJ frame descriptor (index in IDX)
```

Use the IDX field to select which index to return.

- Minor frame descriptor status

```
#define DQ_SL553_BC_MN_DESC     (4L<<12)    // MN descriptor
```

Usethe  MN field to select a minor frame to work with. Use the IDX field to select which index to return.

- BCCB status

```
#define DQ_SL553_BC_CB_STATUS   (3L<<12)    // BCCB status word
```

Use the MN field to select a minor frame to work with.

Usethe  IDX field to select which index to return.

### 4.43.21 DqAdv553SelectMNBlock

**Syntax:**

```
int DAQLIB DqAdv553SelectMNBlock(int hd, int devn, int channel, uint32 block_mask,
uint32* status);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 block_mask | Bitmask to select block for each minor frame type |
| **Output** | |
| uint32* status | Returns status information on whether the minor frame was executed at least once (per minor frame) |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**

DNx-1553-553 layer minor frames are divided into two blocks for double-buffered operations. This function allows you to select which block should be set as an active; i.e., used by the Bus Controller for input and output.

<block_mask> contains a bitmask on per minor frame types to select which block to use (please note that in DqAdv553WriteBCCB() a user can select which block to write data to or both)

<status> returns a bitmask that defines which minor blocks were already executed (from major frame descriptors)

Bits 0 thru 15 represent pending buffer selector value (last value written to BCSWP register). Bits 16 through 31 represent current buffer selector value. If not equal to PBS, the swap cycle is still pending and will be executed at the  next major frame clock.

### 4.43.22 DqAdv553BCDebug

**Syntax:**

```
int DAQLIB DqAdv553BCDebug(int hd, int devn, int channel, uint32 todo, int
major_idx, int minor_idx, int* current_major_d, int* current_minor_d, uint32*
bc_status);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 todo | Action item |
| int32 major_idx | Major frame index |
| uint32 minor_idx | Minor frame index |
| **Output** | |
| int* current_major_d | Current major frame descriptor |
| int* current_minor_d | Current minor frame descriptor |
| uint32* bc_status | Bus controller status |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**

This function is used for debugging Bus Controller operations. After setting up bus controller tables, one must call this function instead of DqAdv553Enable() to perform step-by-step execution of the loaded tables.

<todo> specifies the action to perform:

```
#define DQ_SL553_BCDBG_START    1   // Start debugger
#define DQ_SL553_BCDBG_STEP     2   // Perform one step
#define DQ_SL553_BCDBG_GOTO     3   // Go to specified descriptors and then stop
#define DQ_SL553_BCDBG_STOP     4   // Cease BC operations
```

<major_idx> - index in the major frame descriptor table to execute to. Use -1 if this is irrelevant.
<minor_idx> - index in the minor frame descriptor table to execute to. Use -1 if this value is irrelevant.

The function returns:
<current_major_d> - current major frame descriptor where execution was postponed.

<current_minor_d> - current minor frame descriptor where execution was postponed.
<bc_status> - bus controller status [0..15] and addition interrupt status [16..31] registers, combined.

**Note**
User should set up a timeout value in the library long enough to allow the programmed number of commands to execute.

### 4.43.23 DqAdv553WriteTxFifo

**Syntax:**
```
int DAQLIB DqAdv553WriteTxFifo(int hd, int devn, int channel, uint32 tx_size,
uint32* tx_data);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 tx_size | Number of words for transmission |
| uint32* tx_data | Data to transmit |
| **Output** | |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**

This function writes data to Tx FIFO. A write to the FIFO translates to a write to the 1553 bus.

Tx FIFO provides access to the low priority FIFO of the 1553 encoders, a write to this FIFO actually translates to TX activity on the enabled 1553 buses (A or B, but not both). This feature allows use of the RT/BM as a simple bus controller (BC). In such a case, the TX data should be placed directly into the TX FIFO. Note that a programmable data gap between the transfers allows strict hardware timing. The software should be responsible for the receiving all data from the 1553 decoder and should follow the 1553 data exchange protocol.

The results of the command can be monitored by enabling BM mode.

The data format used in TX/RX FIFO operations is as follows:

| PAR | WT | DLY | 0 | Delay | | | | | | | | | | | | Data | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | | 27 | | | | | | | | | | | 16 | 15 | | | | | | | | | | | | | | | 0 |

DLY – 1 means that the register contains a delay in 1us intervals. When this bit is equal to 1, a 29-bit 1us resolution delay counter is used. When this bit is equal to 0, a 12-bit 15.15ns delay counter is used. A 15ns delay counter is started after data is sent over the bus, a 1us delay counter starts immediately after adelay word is proceseds in the FIFO.
PAR – parity 0 or 1
WT – word type 1 is command/status and 0 is data
Delay – 11-bit delay counter in 15.15ns resolution (decrements before data is transferred on the bus, bus is kept in idle state during this delay)
Data – data to send on the bus. Following format applies:

**Command Word**

| Terminal Address | | | | | T/R | Sub-address/Mode | | | | | Word/Mode Count | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | | | | 11 | 10 | 9 | | | | 5 | 4 | | | | 0 |

**Data Word**

| 16-bit data | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | | | | | | | | | | | | | | | 0 |

**Status Word**

| Terminal Address | | | | | ME | INS | SRQ | reserved | | | BC | BUSY | SS | DBA | TERM |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | | | | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

ME – Message Error
INS – Instrumentation bit
SRQ – Service Request bit
BC – Broadcast Command
BUSY – Terminal is busy
SS – Subsystem flag
DBA – Dynamic Bus Acceptance flag
TERM – Terminal Flag

### *4.43.24 DqAdv553Enable*

**Syntax:**

```
int DAQLIB DqAdv553Enable(int hd, int devn, uint32 actions);
```

| **Input** | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int actions | Enable (1) or Disable (0) |
| **Output** | |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**

This function enables or disables 1553 operations for both channels.

### 4.43.25 DqAdv553ReadRAM, DqAdv553WriteRAM

A DNx-1553-553 board has 16MB of SRAM located on the base board of the layer assembly. This memory is used to store remote terminal and bus controller data.
The following two functions are service functionsthat allow you to access device memory directly.

```
int DAQLIB DqAdv553ReadRAM(int hd, int devn, int channel, uint32 memtype, uint32
rt, uint32 sa, uint32 block, uint32 size, uint32* data);
```

```
int DAQLIB DqAdv553WriteRAM(int hd, int devn, int channel, uint32 memtype, uint32
rt, uint32 sa, uint32 block, uint32 size, uint32* data);
```

The memory can be addressed either by using RT/SA and block or by using direct addressing:
If <channel> > 0, then <memtype>, <rt>, <sa> and <block> are used to calculate a memory address.
If <channel> < 0, then <memtype> should contain the address of the internal layer SDRAM to be read.

### 4.43.26 DqRtVmapAddOutputChannelData for DNx-1553-553

To simplify specifying what channels need to be written, we recommend you use the following macros:

`DQ_L553_CH_RT_SA(CH,RT,SA)` – write to specified RT/SA data field. In VMap+ mode, a write always starts at the first location of the data memory dedicated for this channel/remote terminal/subaddress combination and contains the number of words specified in <data_size>. Normally, all access is done using block 0 for both read and write. If use of the second block is required (for example, in case of the file transfer), a different macro should be used:

`DQ_L553_DATA(CH, RT, SA, BLK)` – this macro allows you to specify a block of data to access.

`DQ_L553_CONTROL_CFG(CH, RT, RQ)` – write control word to the specified channel and remote terminal. <RQ> can be one of the following:

```
DQ_L553_RTS_CFG0    (1L<<0) // Bits to control status
DQ_L553_RTS_CFG1    (1L<<1) // Vector (hi) and BIT (low) words
DQ_L553_RTS_TX_BLK  (1L<<2) // Set Tx block per RT (0 or 1)
DQ_L553_RTS_RTEN    (1L<<3) // Enable/disable terminals mask
```

For all output needs, the DQ_SS0OUT subsystem of the device should be specified.

### 4.43.27 DqRtVmapRqInputChannelData for DNx-1553-553

For remote terminal mode:

To simplify specifying what channels need to be written, we recommend that you use the following macros:

`DQ_L553_CH_RT_SA(CH,RT,SA)` – read from specified channel.

`DQ_L553_DATA(CH,RT,SA,BLK)` – this macro allows you to specify a block of data to access for a read.

Input data from a DNx-1553-553 layer can contain status information (described in DqAdv553StatusRT()). To retrieve status information, specify the following channel: `#define DQ_L553_STATUS_STSF(CH, RT, STS)`, where <STS> is one of the follows or their combination (proper data size should be allocated in the packet to accommodate all status data). Status data is always represented as 32-bit (i.e., user should allocate two 16-bit fields per each status entry):

```
DQ_L553_RTS_STS0        (1L<<0)   // Last command (hi) and status (low) by RT
DQ_L553_RTS_STS1        (1L<<1)   // Last SYNC (hi) and Tx shutdown (low) by RT
DQ_L553_RTS_STS2        (1L<<2)   // Last mode command (low 16-bits) by RT
DQ_L553_RTS_CHSTAT      (1L<<3)   // Channel status, RT/SA ignored
DQ_L553_RTS_BERRORS     (1L<<4)   // Bus errors, RT/SA ignored
```

Data status information is a bit mask of which subadresses received or transmitted information per each RT. Use the following macro to specify that data ready or data sent status needs to be received:

```
#define DQ_L553_STATUS_DATA(CH, RT, BLK, RQ)
DQ_L553_BUSMON_DATA(CH)
```

These are special status bits telling whether transmit or receive has happened.

```
DQ_L553_RTS_DATAREADY   (1L<<0)   // Data ready status for this terminal
DQ_L553_RTS_DATASENT    (1L<<1)   // Data sent status for active block
```

For all input data, the DQ_SS0IN subsystem of the device should be specified.
To access the bus monitor FIFO (is enabled), use `DQ_L553_BUSMON_DATA(CH)`.

For bus controller VMap, channel numbers are formed as follows:

```
#define DQ_SL553_BC_VMAP(CH, TYPE, MN, BLK, IDX)
```

Type defines what sort of data needs to be accessed:

```
#define DQ_SL553_BC_DATA        (1L<<12)    // BCCB data (write data/read status)
#define DQ_SL553_BC_STATUS      (2L<<12)    // BCCB (read status/write control)
#define DQ_SL553_BC_BC_STATUS   (3L<<12)    // BC status word
#define DQ_SL553_BC_BUSMON      (4L<<12)    // Bus monitor access (write)
#define DQ_SL553_BC_MN_DESC     (5L<<12)    // MN descriptor (read/write)
#define DQ_SL553_BC_MJ_DESC     (6L<<12)    // MJ descriptor (read/write)
```

For DQ_SL553_BC_STATUS five statuses are requested:

`DQ_SL553_BC_STATUS_BC:` return status of BC controllers (register 0x21A4)

`DQ_SL553_BC_STATUS_PORT:` port status

`DQ_SL553_BC_STATUS_BCPOS:` current MJ and MN position and GOTO status
`DQ_SL553_BC_STATUS_ISRC:` return interrupt sources included IDs when interrupt was requested
`DQ_SL553_BC_STATUS_ERR:` return error source and frame IDs

### DQ_SL553_BC_STATUS_BCPOS:

```
#define SL553_PORT_BCPOS_GP          // =1 if return from GOTO is pending
#define SL553_PORT_BCPOS_GET_GMNFD(N) // returns stored MN return-to GOTO command position
#define SL553_PORT_BCPOS_GET_GMJFD(N) // returns stored MJ return-to GOTO command position
#define SL553_PORT_BCPOS_CUR_BLOCK(N) // returns current MN block
#define SL553_PORT_BCPOS_CUR_MNFD(N)  // returns current MN position (top bit is block)
#define SL553_PORT_BCPOS_CUR_MJFD(N)  // returns current MJ position
```

### DQ_SL553_BC_STATUS_BC:

```
#define SL553_PORT_BCSTS_BSY    // =1 if BC not in IDLE or WAIT_CLOCK states
#define SL553_PORT_BCSTS_RSV30  // Reserved
#define SL553_PORT_BCSTS_RSV29  // Reserved
#define SL553_PORT_BCSTS_RSV28  // Reserved
#define SL553_PORT_BCSTS_MRF3   // Reports ID for the currently processed
#define SL553_PORT_BCSTS_MRF0   // minor frame
#define SL553_PORT_BCSTS_DSC7   // Reports ID for the currently processed
#define SL553_PORT_BCSTS_DSC0   // descriptor minor frame
                                // 12-0 are sticky bits) auto-cleared
#define SL553_PORT_BCSTS_BIR    // Response received from incorrect RT
#define SL553_PORT_BCSTS_BTO    // Maximum 1553 access time exceeded
#define SL553_PORT_BCSTS_BAD    // Bus activity is detected while in wait for clock (idle) state
#define SL553_PORT_BCSTS_BWB    // Response received on the wrong bus
#define SL553_PORT_BCSTS_BCO    // BC overrun - major/minor frame clock called in non-idle state
#define SL553_PORT_BCSTS_HBT    // Heartbeat - set to one at the beginning of each minor frame
#define SL553_PORT_BCSTS_MTD    // Memory access timeout was detected (hardware error)
#define SL553_PORT_BCSTS_RTR    // At least one RT in "dead" state replied with valid status
#define SL553_PORT_BCSTS_MF     // At least one Mode command w/o data word failed
#define SL553_PORT_BCSTS_MRBF   // At least one Mode TX command with data word failed
#define SL553_PORT_BCSTS_MBRF   // At least one Mode RX command with data word failed
#define SL553_PORT_BCSTS_RBF     // At least one RT->BC transmission failed
#define SL553_PORT_BCSTS_BRF     // At least one BC->RT transmission failed
```

### SL553_PORT_BCISRC:

```
//    Holds last ID of the BC sequence that called IRQ request
#define SL553_PORT_BCISRC_MRF3   23  // Reports ID for the minor frame that was processed
#define SL553_PORT_BCISRC_MRF_SH 16  // when interrupt was requested
#define SL553_PORT_BCISRC_MFN7   15  // Reports record # inside of the major frame descriptor table
#define SL553_PORT_BCISRC_MFN_SH 8   // when interrupt was requested
#define SL553_PORT_BCISRC_FRN7   7   // Reports record # inside of the minor frame descriptor table
#define SL553_PORT_BCISRC_FRN_SH 0   // when interrupt was requested
```

### SL553_PORT_BCERR:

```
// R    Last error code and last executed entry ID
#define SL553_PORT_BCERR_CODE15  31  // Error source code
#define SL553_PORT_BCERR_CODE_SH 16  //
#define SL553_PORT_BCERR_MFN7    15  // Reports record # inside of the major frame descriptor table
#define SL553_PORT_BCERR_MFN_SH   8  // when interrupt was requested
#define SL553_PORT_BCERR_FRN7     7  // Reports record # inside of the minor frame descriptor table
#define SL553_PORT_BCERR_FRN_SH   0  // when interrupt was requested
```

Please note that both VMap and VMap+ operations can be used to store/retrieve data to the bus controller, but only VMap can be used for the status fields.

To access bus monitor FIFO (is enabled), use `DQ_L553_BUSMON_DATA(CH).`

### *4.43.28 Configuring DNx-1553-553 for Bus Monitor Mode*

First, one or both ports of the layer need to be configured, as follows:

1. Set operation mode for the channel:
```
mode_0 = DQ_L553_MODE_BM;
flags_0 = DQ_L553_TRANSFORMER;
ret = DqAdv553SetMode(hd0, DEVN, CHAN, mode_0, flags_0);
```

2. Set configuration for bus monitor:
`DqAdv553ConfigBM()`

Typically, bus mode is configured as follows:
```
busmode_0 = DQ_L553_BM_LSTN_A |
            DQ_L553_BM_LSTN_B |
            DQ2_L553_BM_TX_BUS | <- either bus A or bus B
            DQ_L553_STORE_TS | <- receive timestamp
            DQ_L553_STORE_FLAGS; <- receive flags
```

3. Enable operations: `DqAdv553Enable(hd0, DEVN, TRUE);` This function simultaneously enables operations on both channels.

In the data retrieval loop, use `DqAdv553RecvBMMessages()` to retrieve messages:
```
DqAdv553RecvBMMessages(hd0, DEVN, bc_port, rx_size, pMsg, &messages);
for (j = 0; j < messages; j++) {
   for (i = 0; i < pMsg->size; i++) {
      printf("%x ", pMsg->data[i]);
   }
   printf(“\n”);
}
```

To finish up, call `DqAdv553Enable(hd0, DEVN, FALSE);` This call will disable all I/O operations on the layer.

### 4.43.29 Configuring DNx-1553-553 for Remote Terminal Mode

1. Set operation mode for the channel:

```
mode_0 = DQ_L553_MODE_RT;
flags_0 = DQ_L553_TRANSFORMER;
ret = DqAdv553SetMode(hd0, DEVN, CHAN, mode_0, flags_0);
```

2. Configure remote terminal mode:

   a. fill channel list with information as to what remote terminals and what subadresses are included in operations:

```
 for (i = FIRST_RT; i <= LAST_RT; i++) {
       for (j = FIRST_SA; j <= LAST_SA; j++) {
       rt_valid_list[rt_rt_size] = 0;
       rt_rtsa_list[rt_rt_size] = DQ_L553_RT_SA(i, j);
       printf("port:%d rt:%d sa:%d\n", rt_port, i, j);
       rt_rt_size++;
       }
 }
```

   b. select bus mode – enable both buses to listen and to transmit. A remote terminal will reply on the same bus the Bus Controller used to send a command:

```
busmode_rt = DQ_L553_BM_LSTN_A |    // allow listening on both channels
             DQ_L553_BM_LSTN_B |
             DQ_L553_BM_TX_A |      // allow transmission on both
             DQ_L553_BM_TX_B |
             DQ_L553_STORE_TS |
             DQ_L553_STORE_FLAGS;
```

   c. Either program a validation list or set it to NULL if you don't want to impose any limitations on the size of the data allowed to be sent or received.

   d. Call `DqAdv553ConfigRT()` to transmit remote terminal configuration to the firmware.

3. Enable operations: `DqAdv553Enable(hd0, DEVN, TRUE);` This function simultaneously enables operations on both channels.

In the data loop, perform:

1. `DqAdv553ReadStatusRT()` – to retrieve bus status (make sure there is no bus error) and remote terminal status (which subaddresses received or transmitted data). For example, these two status words are most useful:

```
st_list[0] = DQ_L553_RT_RT(TERMADDR)|DQ_L553_RT_DATA_RDY;
st_list[1] = DQ_L553_RT_RT(TERMADDR)|DQ_L553_RT_CHSTAT;
```

1. `DqAdv553ReadRT()` – create a channel list for this call based on information as to which subaddresses received data from the previous step.

2. `DqAdv553WriteRT()` – create a channel list including the subaddresses you need to update.

To finish up, call `DqAdv553Enable(hd0, DEVN, FALSE);` this call will disable all I/O operations on the layer.

### 4.43.30 DqAdv553ConfigEvents

**Syntax:**
```
int DAQLIB DqAdv553ConfigEvents(int handle, int devn, int channel, event553_t
event, uint32 rtsa, uint32* param)
```

**Command:**
Specify what asynchronous notification events user wants to receive from the layer

| **Input** | |
|---|---|
| int handle | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| `event553_t event` | Event type |
| `uint32 rtsa` | Remote terminal and subaddress |
| `uint32* param` | Up to seven additional parameters depends on event type |
| **Output** | |
| None | |
| **Returns** | |
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |
| | |

**Description**
This function configures events to be sent from the layer.

The following events are defined:
```
typedef enum {
  EV553_CLEAR = 0x100,        // clear all events

  // bus or RT controller event
  EV553_NO_ACTIVITY,  // no activity on the bus over certain (programmed) time
  EV553_IN_FIFO,      // input FIFO overrun (BM) or half-full
  EV553_OUT_FIFO,     // output FIFO underrun or half-empty
  EV553_BUS_ERROR,    // bus or protocol errors, including RT-RT errors
  EV553_FSM_ERROR,    // state machine error or MC error
  EV553_BUS_CHANGED,  // warning - bus was changed
  // mode commands received
  EV553_MD_SHUTDOWN,  // transmitter shutdown or override related event
  EV553_MD_SYNCHRONIZE,   // synchronize event
  EV553_MD_RESET,     // reset condition
  EV553_MD_MISC,      // misc mode commands like ITF

  // receive or transmit (specify RT/SA combination)
  EV553_RX,           // data received
  EV553_TX,           // data transmitted
  EV553_BCST_RX,      // data received (broadcast)
  EV553_BCST_TX,      // data transmitted (broadcast)
  EV553_MD,           // mode command received (by code)

  // bus controller events
  EV553_BC_ERROR,     // one of bus controllers errors
  EV553_BC_BUS,       // bus error: bus not idle, transaction took too long
  EV553_BC_IRQ,       // user-requested interrupt from the BC frame
  EV553_BC_OVERRUN,   // BC overrun event (major or minor frame)
  EV553_BC_CMD_ERR,   // one or more command has failed or an RT is dead
  EV553_BC_HEARTBEAT  // Minor frame started

  // custom configuration events
  EV553_RT_CUST_LUT,  // set look-up table for RT/SA
  EV553_RT_CUST_LUT1, // reserved LUT1
  EV553_RT_CUST_LUT2, // reserved LUT2
  EV553_RT_CUST_LUT3, // reserved LUT3

    // update on the current activity
  EV553_RT_PERIODIC,   // update which RT/SA received and sent data (not available
in BC mode)
  EV553_BC_PERIODIC    // <reserved> update which BCCBs are executed (not
available in TR mode)

} event553_t;
```

If an event happens a packet of the following structure is sent from the cube to notify the host about this event:

```
typedef struct {
    uint8   dev;    // device
    uint8   ss;     // subsystem
    uint16  size;   // data size
    uint32  event;  // event type or command and additional flags
    uint8   data[]; // data: for 1553-553 layer of EV553_ID type
} DQEVENT, *pDQEVENT
```

Event data is always layer-specific. In this case EV553_ID structure is used to inform user application about what type of the events has happened on DNx-1553-553 layer:

```
typedef struct {
    uint32 chan;    // channel information
    uint32 evtype;  // type of the event
    uint32 rtsa;    // rtsa information
    uint32 cmd;     // command and status
    uint32 sts;     // applicable status register
    uint32 tstamp;  // timestamp of event
    uint32 size;    // size of the following data in bytes
                    // the data is comprised of cmd, sts and data words…
    uint32 data[];  // data to follow
} EV553_ID, *pEV553_ID
```

Every type of the event requires specific parameters (or none) and sends a notification with specific fields and data.

For all types of events:

`DQPKT:`
dqCounter = number of event on per channel/per device basis [1..65535]
dqCommand = DQCMD_EVENT | DQ_REPLY

`DQEVENT:`
dev = device number
ss = DQ_SS0IN
event = one of the event types described below
size = size of the following EV553_ID structure including data

`EV553_ID:`
chan = channel
evtype = event subtype

By event type:

**`EV553_CLEAR: clear all settings`**

| **In** | Description | Comments |
|---|---|---|
| <param> | None | |
| **Out** | | |
| | None | |

**`EV553_NO_ACTIVITY: no activity on the bus over programmed time`**

| **In** | Description | Comments |
|---|---|---|
| <param> | None | |
| **Out** | | |
| `pDQEVENT->event` | `EV553_NO_ACTIVITY` | |
| `pEV553_ID->evtype` | `None` | |
| `pEV553_ID->tstamp` | `10us resolution timestamp` | |
| `pEV553_ID->data[0]` | `Watchdog register. Bits [29..16] are delay in 100us steps` | |

**`EV553_IN_FIFO: Bus Monitor input FIFO event`**

| **In** | Description | Comments |
|---|---|---|
| <param> | None | |
| **Out** | | |
| `pDQEVENT->event` | `EV553_IN_FIFO` | |

| | | |
|---|---|---|
| pEV553_ID->evtype | SL553_PORT_IR_IFH and/or SL553_PORT_IR_IFF flags | Input FIFO half-full and FIFO half-full flags |
| pEV553_ID->tstamp | 10us resolution timestamp | |
| pEV553_ID->data[0] | Actual amount of data in the FIFO | |

**EV553_OUT_FIFO: transmit FIFO event**

| **In** | Description | Comments |
|---|---|---|
| <param> | None | |
| **Out** | | |
| pDQEVENT->event | EV553_OUT_FIFO | |
| pEV553_ID->evtype | SL553_PORT_IR_OFH and/or SL553_PORT_IR_OFE flags | Output FIFO half-empty and FIFO empty flags |
| pEV553_ID->tstamp | 10us resolution timestamp | |
| pEV553_ID->data[0] | Actual amount of data in the FIFO | |

**EV553_BUS_ERROR: and error or abnormal event on 1553 bus**

| **In** | Description | Comments |
|---|---|---|
| <param> | None | |
| **Out** | | |
| pDQEVENT->event | EV553_BUS_ERROR | |
| pEV553_ID->evtype | One or a combination of the following flags<br>SL553_PORT_IR_ITA<br>SL553_PORT_IR_ITB<br>SL553_PORT_IR_BEA<br>SL553_PORT_IR_BEB<br>SL553_PORT_IR_MED<br>SL553_PORT_IR_TMW<br>SL553_PORT_IR_TFW<br>SL553_PORT_IR_ICD<br>SL553_PORT_IR_RTD<br>SL553_PORT_IR_RVF<br>SL553_PORT_IR_DWG | One of the following errors was detected on bus A/B<br>Invalid Manchester code during SYNC or data bit time<br>Bit timing error was detected on bus A/B: rising/falling edge timing is invalid, bit timing timeout<br>Message error detected - data error, suppress status<br>Too many words were received within RX message, suppressed<br>Too few words were received within RX message, suppressed<br>Message error detected - illegal/illogical command error<br>RTRT Timeout detected<br>RTRT Validation failed<br>Gap within data word detected (use only when xxRTR_DGP>0) |
| pEV553_ID->tstamp | 10us resolution timestamp | |
| pEv553_ID->sts | Port status – see below | |

Port status bits:

| | |
|---|---|
| SL553_PORT_STS_BSF | RT: Current bus that driving BM FIFO (1=BUS A) |
| SL553_PORT_STS_BSR | RT: Current bus that driving RT (1=BUS A) |
| SL553_PORT_STS_ENB | RT: Bus B Encoder is enabled in RT Engine |
| SL553_PORT_STS_ENA | RT: Bus A Encoder is enabled in RT Engine |
| SL553_PORT_STS_DRB | If set - data is ready at the decoder B |
| SL553_PORT_STS_DRA | If set - data is ready at the decoder A |
| SL553_PORT_STS_BEB | BUS: Bit timing error on bus B (sticky, auto-cleared after read) |
| SL553_PORT_STS_BEA | BUS: Bit timing error on bus A (sticky, auto-cleared after read) |
| SL553_PORT_STS_PEB | BUS: Parity error on bus B (sticky, auto-cleared after read) |
| SL553_PORT_STS_PEA | BUS: Parity error on bus A (sticky, auto-cleared after read) |
| SL553_PORT_STS_ILC | RT: Illegal command (sticky, auto-cleared after read) |
| SL553_PORT_STS_MER | RT: Message error (sticky, auto-cleared after read) |
| SL553_PORT_STS_TMW | RT: Too many words were detected in RX message (sticky, auto-cleared after read) |
| SL553_PORT_STS_TFW | RT: Too few words were detected inm RX message (sticky, auto-cleared after read) |
| SL553_PORT_STS_ERB | If set - data overflow was detected at the decoder B (sticky, auto-cleared after read) |
| SL553_PORT_STS_ERA | If set - data overfolw was detected at the decoder A (sticky, auto-cleared after read) |

**EV553_FSM_ERROR: Internal errors of the 1553 RT state machine – debug only**

| **In** | Description | Comments |
|---|---|---|
| <param> | None | |
| **Out** | | |
| pDQEVENT->event | EV553_FSM_ERROR | |
| pEV553_ID->evtype | SL553_PORT_IR_DOA<br>SL553_PORT_IR_DOB<br>SL553_PORT_IR_EMT | 1553 bus A/B data was overriten - critical IRQ<br>FSM stuck somewhere and new 1553 command/data arrived before previous PSRAM memory access timeout |
| pEV553_ID->tstamp | 10us resolution timestamp | |
| pEv553_ID->sts | Port status – see description above | |

**EV553_BUS_CHANGED: Communication bus was switched**

| **In** | Description | Comments |
|---|---|---|
| <param> | None | |
| **Out** | | |
| pDQEVENT->event | EV553_BUS_CHANGED | |
| pEV553_ID->evtype | SL553_PORT_IR_BCH | |
| pEV553_ID->tstamp | 10us resolution timestamp | |
| pEv553_ID->sts | Port status – see description above | |

**EV553_BUS_CHANGED: Communication bus was switched**

| **In** | Description | Comments |
|---|---|---|
| <param> | None | |
| **Out** | | |

| | | |
|---|---|---|
| pDQEVENT->event | EV553_BUS_CHANGED | |
| pEV553_ID->evtype | SL553_PORT_IR_BCH | |
| pEV553_ID->tstamp | 10us resolution timestamp | |
| pEv553_ID->sts | Port status – see description above | |

Mode commands: these are mode commands received at the highest level (decoder) without information about which terminal it belongs to. Due to the fact that DNx-1553-553 is comprised of 32 RTs these are commands that received on a single RT affect the behavior of all terminals. If you are interested in per-terminal information use EV553_MD event instead.

**EV553_MD_SHUTDOWN: Bus shutdown mode command**

| **In** | Description | Comments |
|---|---|---|
| <param> | None | |
| **Out** | | |
| pDQEVENT->event | EV553_MD_SHUTDOWN | |
| pEV553_ID->evtype | SL553_PORT_IR_STS<br>SL553_PORT_IR_OTS<br>SL553_PORT_IR_DBC<br>SL553_PORT_IR_DBA | Mode command: Selected transmitter shutdown received with data<br>Mode command: Override selected transmitter shutdown received with data<br>Mode command: Dynamic bus change request received in mode command<br>Mode command: Dynamic bus change request accepted |
| pEV553_ID->tstamp | 10us resolution timestamp | |
| pEv553_ID->sts | Port status – see description above | |

**EV553_MD_SYNCHRONIZE: Synchronize mode command**

| **In** | Description | Comments |
|---|---|---|
| <param> | None | |
| **Out** | | |
| pDQEVENT->event | EV553_MD_SYNCHRONIZE | |
| pEV553_ID->evtype | SL553_PORT_IR_MDS | |
| pEV553_ID->tstamp | 10us resolution timestamp | |
| pEv553_ID->sts | Port status – see description above | |

**EV553_MD_RESET: Synchronize mode command**

| **In** | Description | Comments |
|---|---|---|
| param[0] | TRUE is actual reset should not be performed and BUSY bit | |

| | | |
|---|---|---|
| | should not be set | |
| **Out** | | |
| `pDQEVENT->event` | `EV553_MD_RESET` | |
| `pEV553_ID->evtype` | `SL553_PORT_IR_RRT` | |
| `pEV553_ID->tstamp` | `10us resolution timestamp` | |
| `pEv553_ID->sts` | Port status – see description above | |

Following are RT events:

**`EV553_RX: message received`**

| **In** | Description | Comments |
|---|---|---|
| param[0] | Maximum number of words received by this terminal/subaddress to return in the event packet | The actual number of returned words is min(number of received words, maximum number) for Uint16. Returns this amount for uint32 |
| param[1] | sizeof(uint16) or sizeof(uint32) | Tells firmware to send received date either in uint16 or uint32 format. ->event, ->evtype, ->tstamp and ->sts will always be 32 bits. If you select uint32, firmware will pad the upper 16 bits with 0. |
| **Out** | | |
| `pDQEVENT->event` | `EV553_RX` | |
| `pEV553_ID->evtype` | 1 in the bitfield that corresponds to each RT that received message | |
| `pEV553_ID->tstamp` | `10us resolution timestamp` | |
| `pEv553_ID->sts` | Port status – see description above | |
| For each flagged RT | | |
| data[i+0] | Command and status | [31..16] = command |
| data[i+1] | 1 in the bitfield that corresponds to each SA that received message for this RT | |
| For each flagged SA of this RT | | |
| data[i+n] | 16 or 32 bit data from that SA | Amount is data is a minimum between requested and contained in the received word for uint16. It is always fixed for uint32 |

**`EV553_TX: messages transmitted`**

| **In** | | Description | Comments |
|---|---|---|---|
| <None> | | | |
| **Out** | | | |
| `pDQEVENT->event` | | `EV553_TX` | |
| `pEV553_ID->evtype` | | 1 in the bitfield that corresponds to each RT that have transmitted message | |
| `pEV553_ID->tstamp` | | `10us resolution timestamp` | |
| `pEv553_ID->sts` | | Port status – see description above | |
| For each flagged RT | | | |
| | data[i+0] | Command and status | [31..16] = command |
| | data[i+1] | 1 in the bitfield that corresponds to each SA that received message for this RT | |

**`EV553_MD: mode command received`**

| **In** | | Description | Comments |
|---|---|---|---|
| <None> | | | |
| **Out** | | | |
| `pDQEVENT->event` | | `EV553_MD` | |
| `pEV553_ID->evtype` | | 1 in the bitfield that corresponds to each RT that have received mode command | |
| `pEV553_ID->tstamp` | | `10us resolution timestamp` | |
| `pEv553_ID->sts` | | Port status – see description above | |
| For each flagged RT | | | |
| | data[i+0] | Command and status | [31..16] = command |
| | data[i+1] | SYNC/transmitter shutdown/override word | |

**`EV553_BCST_TX: broadcast command transmitted`**

| **In** | | Description | Comments |
|---|---|---|---|
| <None> | | | |
| **Out** | | | |
| `pDQEVENT->event` | | `EV553_BCST_TX` | |
| `pEV553_ID->evtype` | | 1 in the bitfield that corresponds to each RT that have transmitted message | |
| `pEV553_ID->tstamp` | | `10us resolution timestamp` | |
| `pEv553_ID->sts` | | Port status – see description above | |
| For each flagged RT | | | |
| | data[i+0] | Command and status | [31..16] = command |
| | | | |

**EV553_BCST_RX: broadcast command is received**

| In | Description | Comments |
|---|---|---|
| <None> | | |
| **Out** | | |
| pDQEVENT->event | EV553_BCST_TX | |
| pEV553_ID->evtype | 1 in the bitfield that corresponds to each RT that have transmitted message | |
| pEV553_ID->tstamp | 10us resolution timestamp | |
| pEv553_ID->sts | Port status – see description above | |
| For each flagged RT | | |
| data[i+0] | Command and status | [31..16] = command |
| | | |

Bus controller events:

**EV553_BC_ERROR: bus controller error**

| In | Description | Comments |
|---|---|---|
| <None> | | |
| **Out** | | |
| pDQEVENT->event | EV553_BC_ERROR | |
| pEV553_ID->evtype | SL553_PORT_BCI_EIRQ<br>SL553_PORT_BCI_MTD | "Entry Error" - problem with one or more enabled entry<br>Memory access timeout was detected (hardware error) |
| pEV553_ID->tstamp | 10us resolution timestamp | |
| pEv553_ID->sts | Port status – see description above | |
| data[0] | SL553_PORT_BCISRC | Error source |
| data[1] | SL553_PORT_BCERR | Error code |

**EV553_BC_BUS: bus controller bus error**

| In | Description | Comments |
|---|---|---|
| <None> | | |
| **Out** | | |
| pDQEVENT->event | EV553_BC_BUS | |
| pEV553_ID->evtype | SL553_PORT_BCI_BIA<br>SL553_PORT_BCI_MTO | Critical IRQ - indicates bus activity when idle state is expected<br>Maximum 1553 access time exceeded |
| pEV553_ID->tstamp | 10us resolution timestamp | |
| pEv553_ID->sts | Port status – see description above | |

| | | |
|---|---|---|
| `data[0]` | SL553_PORT_BCISRC | Error source |
| `data[1]` | SL553_PORT_BCERR | Error code |

**`EV553_BC_IRQ: bus controller user requested IRQ`**

| **In** | Description | Comments |
|---|---|---|
| <None> | | |
| **Out** | | |
| `pDQEVENT->event` | `EV553_BC_IRQ` | |
| `pEV553_ID->evtype` | SL553_PORT_BCI_UIRQ | User requested IRQ in a minor frame entry |
| `pEV553_ID->tstamp` | `10us resolution timestamp` | |
| `pEv553_ID->sts` | Port status – see description above | |
| `data[0]` | SL553_PORT_BCISRC | [23..16]= minor frame number when interrupt was requested<br>[15...8] = major frame entry<br>[7…0] = minor frame entry |
| `data[1]` | SL553_PORT_BCERR | [23..16]= error code<br>[15...8] = major frame entry<br>[7…0] = minor frame entry |

**`EV553_BC_HEARTBEAT: bus controller informs when minor frame starts`**

| **In** | Description | Comments |
|---|---|---|
| <None> | | |
| **Out** | | |
| `pDQEVENT->event` | `EV553_BC_HEARTBEAT` | |
| `pEV553_ID->evtype` | SL553_PORT_BCI_HBT | |
| `pEV553_ID->tstamp` | `10us resolution timestamp` | |
| `pEv553_ID->sts` | Port status – see description above | |
| `data[0]` | SL553_PORT_BCISRC | [23..16]= minor frame number when interrupt was requested<br>[15...8] = major frame entry<br>[7…0] = minor frame entry |
| `data[1]` | SL553_PORT_BCERR | [23..16]= error code<br>[15...8] = major frame entry<br>[7…0] = minor frame entry |

**`EV553_BC_OVERRUN: bus controller minor and major frame overrun – no time to complete current frame, BC starts new frame`**

| **In** | Description | Comments |
|---|---|---|
| <None> | | |
| **Out** | | |
| `pDQEVENT->event` | `EV553_BC_OVERRUN` | |
| `pEV553_ID->evtype` | SL553_PORT_BCI_BCRO | BC overrun - minor frame clock called in non-idle state |

| | | |
|---|---|---|
| | SL553_PORT_BCI_BCJO | BC overrun - major frame clock called in non-idle state |
| pEV553_ID->tstamp | 10us resolution timestamp | |
| pEv553_ID->sts | Port status – see description above | |
| data[0] | SL553_PORT_BCISRC | Error source |
| data[1] | SL553_PORT_BCERR | Error code |

**EV553_BC_CMD_ERR:**

| **In** | Description | Comments |
|---|---|---|
| <None> | | |
| **Out** | | |
| pDQEVENT->event | EV553_BC_CMD_ERR | |
| pEV553_ID->evtype | SL553_PORT_BCI_RTR\|<br>SL553_PORT_BCI_MF<br>SL553_PORT_BCI_MRBF<br>SL553_PORT_BCI_MBRF<br>SL553_PORT_BCI_RBF<br>SL553_PORT_BCI_BRF | At least one RT in "dead" state replied with valid status<br>At least one Mode command w/o data word failed<br>At least one Mode TX command with data word failed<br>At least one Mode RX command with data word failed<br>At least one RT->BC transmission failed<br>At least one BC->RT transmission failed |
| pEV553_ID->tstamp | 10us resolution timestamp | |
| pEv553_ID->sts | Port status – see description above | |
| data[0] | SL553_PORT_BCISRC | Error source |
| data[1] | SL553_PORT_BCERR | Error code |

**EV553_RT_PERIODIC: message received**

| **In** | Description | Comments |
|---|---|---|
| param[0] | 66Mhz clock divider - 1 | Minimum is 10us or DQ_LN_10us_TIMESTAMP constant<br>Maximum is 0xffffffff or 65 seconds |
| param[1] | Bitmask of RTs to inform about receive and sent commands | RT1 mask is (1L<<1) |
| **Out** | | |
| pDQEVENT->event | EV553_RT_PERIODIC | |
| pEV553_ID->evtype | 1 in the bitfield that | |

| | | |
|---|---|---|
| | corresponds to each RT that is monitors | |
| `pEV553_ID->tstamp` | `10us resolution timestamp` | |
| `pEv553_ID->sts` | Port status – see description above | |
| For each flagged RT | | |
| data[i+0] | Data ready bitmask, “1” in the positions of SAs that received new data | SA mask is cleared by reading data from this RT/SA |
| data[i+1] | Data sent bitmask, “1” in the positions of SAs that sent new data | SA mask is cleared by writing new data to this RT/SA |

**Note:**

- Maximum event rate is limited to 50us or 20kHz to avoid interrupt overrun. If this interrupt rate is exceeded interrupt routine disables events and firmware status flag STS_FW is set to STS_FW_OVERLOAD.

### *4.43.31 DqRtAsync553WriteRT*

**Syntax:**

```
int DAQLIB DqRtAsync553WriteRT(int hd, int devn, int channel, int
request_ack, uint32 rt_size, uint32* rtsa_list, uint16** data)
```

**Command:**

Write remote terminal data using asynchronous socket with or without acknowledge

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number |
| `int channel` | Channel (0 or 1) |
| `int request_ack` | |
| `uint32 rt_size` | Size of the following RT/SA list |
| `uint32* rtsa_list` | Array of RT/SA to write to |
| `uint16** data` | Pointer to the array of pointers to the data for each entry in rtsa_list |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description**

This function writes data to the RT/SA memory, which is used to send out data upon receiving a Transmit command from a BC. See `DqAdv553WriteRT()` for full description.

### *4.43.32 DqRtAsync553ReadRT*

**Syntax:**

```
int DAQLIB DqRtAsync553ReadRT(int hd, int devn, int channel, uint32 rt_size,
uint32* rtsa_list, uint16** data)
```

**Input**

| | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int channel | Channel (0 or 1) |
| uint32 rt_size | Size of the following RT/SA list |
| uint32* rtsa_list | Array of RT/SA to read from |
| uint16** data | Pointer to arrays to store data from the specified sub-addresses |

**Output**

**Returns**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 1553-553 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description**

This function reads data from the RT/SA memory, which is updated upon a Receive command from a BC. See DqAdv553WriteRT() for full description.

## 4.44 DNA-CT-601 and DNA-CT-602 layer

The DNA-CT-601 layer is an 8-channel counter-timer layer.
The DNA-CT-602 layer is a 4-channel counter-timer layer that has differential inputs and outputs. The following DqAdv601 functions may be used with the CT-602 layer. Because the CT-602 has 4 counter/timer channels the `counter` input parameter may only be in the range of 0 to 3.

### 4.44.1 DqAdv601EnableAll

**Syntax:**

`int DqAdv601EnableAll(int hd, int devn)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function enables all counters on the layer.

**Note:**

None.

### 4.44.2 DqAdv601DisableAll

**Syntax:**

`int DqAdv601DisableAll(int hd, int devn)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |

| | |
|---|---|
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function disables all counters on the layer. Counters will hold the last value until enabled again.

**Note:**

None.

### *4.44.3 DqAdv601StartCounter*

**Syntax:**

```
int DqAdv601StartCounter(int hd, int devn, int counter)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 7) |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function starts the specified counter. Can be used as a soft-start for counters with startmode = DQ_PL601_SMSOFT.

**Note:**

None.

### *4.44.4 DqAdv601StopCounter*

**Syntax:**

```
int DqAdv601StopCounter(int hd, int devn, int counter)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 7) |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function stops the specified counter.

**Note:**

None.

### *4.44.5 DqAdv601ClearCounter*

**Syntax:**

```
int DqAdv601ClearCounter(int hd, int devn, int counter)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 7) |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function clears the current count (CR, CRH, or CRH/CRL) of the specified counter.

**Note:**

None.

### *4.44.6 DqAdv601Read*

**Syntax:**

```
int DqAdv601Read (int hd, int devn, int CLSize, uint32 *cl, uint32
*data)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Number of channels to read |
| `uint32 *cl` | Channel list |
| `uint32 *data` | Pointer to receive data values |

**Output:**

| | |
|---|---|
| `uint32 *data` | Read data values |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid, `reg` is not one of the `DQ_CTU` constants, or `value` is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads the current count values from the counters specified in the channel list. Registers are specified by bitwise ORing the channel list (`cl`) with either of the following contants predefined in the powerdna.h header:

- DQ_PL_601_CHNLTYPE_CR0 (CR0 register)
- DQ_PL_601_CHNLTYPE_CR1 (CR1 register)

This function is used for immediate mode.

**Note:**

Period measurement returns 2 uint32 per channel. It is up to the user to determine the size of uint32 samples per channel when calling this function.

DqAdv601ReadRegisterValue() offers the same functionality as this function; however, DqAdv601ReadRegisterValue() provides access to a range of registers. DqAdv601Read() only allows reading from CR0 and/or CR1.

### *4.44.7 DqAdv601Write*

**Syntax:**

```
int DqAdv601Write (int hd, int devn, int CLSize, uint32 *cl, uint32
*data)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Number of channels to write |
| `uint32 *cl` | Channel list |
| `uint32 *data` | Pointer to data values to write |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, `*`data is `NULL`, or a channel number in `cl` is too high. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function writes count values to either or both of the compare registers (CR0, CR1) for the channel specified in the channel list. Registers are specified by bitwise ORing the channel list (`cl`) with either of the following predefined contants:

- DQ_PL_601_CHNLTYPE_CR0
- DQ_PL_601_CHNLTYPE_CR1

This function is used for immediate mode.

**Note:**

DqAdv601WriteRegisterValue() offers the same functionality as this function; however, DqAdv601WriteRegisterValue() provides access to a range of registers. DqAdv601Write() only allows writing to CR0 and/or CR1.

### 4.44.8 DqAdv601SetChannelCfg

**Syntax:**

```
int DqAdv601SetChannelCfg(int hd, int devn, int ss, int counter,
pDQCHNLSET_601_ pCfg, uint32 *cfg)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ss` | Device subsystem |
| `int counter` | Counter number (0 - 7) |
| `pDQCHNLSET_601_ pCfg` | Pointer to channel configuration structure |
| `int *cfg` | Pointer to cfg values |

**Output:**

| | |
|---|---|
| `int *cfg` | Returned config/status |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This internal function sets the low level configuration for a counter channel.

**Note:**

Currently *cfg has no meaning, but may return extra config/status values in the future.

### 4.44.9 DqAdv601ReadRegisterValue

**Syntax:**

```
int DqAdv601ReadRegisterValue(int hd, int devn, int counter, uint32
reg, uint32 *value)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 7) |
| `int reg` | One of the `DQ_CTU_XXX` constants indicating which register to read |
| `uint32 *value` | Pointer to receive data value |

**Output:**

| | |
|---|---|
| `uint32 *value` | Read data value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid, `reg` is not one of the `DQ_CTU` constants, or `value` is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads the value from a register `reg` of the `counter` relative to layer `devn`/counter channel. The following registers are allowed:

```
DQ_CTU_STR
DQ_CTU_PS
DQ_CTU_CR
DQ_CTU_PC
DQ_CTU_CRH
DQ_CTU_CRL
DQ_CTU_FCNTI
DQ_CTU_FCNTO
DQ_CTU_ISR
```

**Note:**

None.

### *4.44.10 DqAdv601WriteRegisterValue*

**Syntax:**

```
int DqAdv601WriteRegisterValue(int hd, int devn, int counter, uint32
reg, uint32 value)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 7) |
| `int reg` | One of the `DQ_CTU_XXX` constants indicating which register to write |
| `uint32 value` | Value to write |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid, `reg` is not one of the `DQ_CTU` constants, or `value` is NULL |

| | |
|---|---|
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads the value from a register `reg` of the `counter` relative to layer `devn`/counter channel. The following registers are allowed:

```
DQ_CTU_CTR
DQ_CTU_CCR
DQ_CTU_PS
DQ_CTU_LR
DQ_CTU_IDBC
DQ_CTU_IDBG
DQ_CTU_PC
DQ_CTU_CR0
DQ_CTU_CR1
DQ_CTU_TBR
DQ_CTU_FIRQI
DQ_CTU_FDTI
DQ_CTU_FIRQO
DQ_CTU_IER
DQ_CTU_ICR
DQ_CTU_FDDO
DQ_CTU_TEST0
DQ_CTU_TEST1
```

**Note:**

None.

### *4.44.11 DqAdv601SetRegister*

**Syntax:**

```
int DqAdv601SetRegister(int hd, int devn, int counter, uint32 reg,
uint32 value)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 7) |
| `uint32 reg` | Register offset |
| `uint32 value` | Value to write |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |

| | |
|---|---|
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function does not perform error-checking on `reg` validity and primarily exists for backward compatibility.

DqAdv601WriteRegisterValue() offers the same functionality as this function; however, DqAdv601WriteRegisterValue () provides error-checking.

**Note:**

None.

### *4.44.12 DqAdv601GetRegister*

**Syntax:**

```
int DqAdv601GetRegister(int hd, int devn, int counter, uint32 reg,
uint32 *value)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `Int counter` | Counter number (0 - 7) |
| `uint32 reg` | Register offset |
| `uint32 value` | Pointer to hold value to read |

**Output:**

| | |
|---|---|
| `uint32 value` | value read |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | `value` is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function does not perform error-checking on the `reg` validity and primarily exists for backward compatibility.

DqAdv601ReadRegisterValue() offers the same functionality as this function; however, DqAdv601ReadRegisterValue () provides error-checking.

**Note:**

None.

### *4.44.13 DqAdv601ConfigCounter*

**Syntax:**

```
     int DqAdv601ConfigCounter(
                                  int hd, int devn, int ss,
                                  int counter,
                                  int startmode, int ps,
                                  int pc, int cr0,
                                  int cr1, int tbr,
                                  int dbg, int dbc,
                                  int iie, int gie,
                                  int oie, int mode,
                                  int trs, int enc,
                                  int gated, int re,
                                  int end_mode, int lr,
                                  int *cfg
                                  )
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ss` | Subsystem this counter belongs to |
| `int counter` | Counter number (0 - 7) |
| `int startmode` | Configure startup mode of counter channel via DQ_PL601_SM### const<br><br>DQ_PL601_SMAUTO<br>counter starts when DqEnable/Read is called.<br>DQ_PL601_SMSOFT<br>counter starts when DqAdv601StartCounter<br>DQ_PL601_SMHARD<br>counter starts via gate line |
| `int ps` | Prescaler register value<br><br>32-bit prescaler divides the 66Mhz counter clock<br>Each counter has an independent prescaler |
| `int pc` | Period counter value<br>Initial value of the period count – period count is started at rising edge of clkin |
| `int cr0` | Compare register 0 value |
| `int cr1` | Compare register 1 value |
| `int tbr` | Interval divider for timebase register<br><br>divider for time-based capture modes (binning) |
| `int dbg` | Input debouncing gate register value |

| | |
|---|---|
| | 16-bit value specifying number of 66Mhz clocks before qualifying the gate signal |
| `int dbc` | Input debouncing clock register value<br>16-bit value specifying number of 66Mhz clocks before qualifying the input (clock) signal |
| `int iie` | `TRUE` to turn on pre-inversion of the input pin |
| `int gie` | `TRUE` to turn on pre-inversion of the gate pin |
| `int oie` | `TRUE` to turn on post-inversion of the output pin |
| `int mode` | One of the `DQ_CM_XXX` counter mode constants - determines which mode to put the counter in<br>DQ_CM_CT<br>Basic timer<br>DQ_CM_ECT<br>external event counter<br>DQ_CM_HP<br>½ period capture<br>DQ_CM_NP<br>N-period capture<br>DQ_CM_QE<br>Quadrature encoder<br>DQ_CM_TCT<br>Triggered event counter<br>DQ_CM_THP<br>Triggered ½ period capture<br>DQ_CM_TNP<br>Triggered n-period capture |
| `int trs` | `TRUE` to use external trigger source (only w/ a triggered mode) |
| `int enc` | `TRUE` to use auto-clear at the end of the count (only in triggered mode) |
| `int re` | `TRUE` to enable re-load (continuous operation) |
| `int gated` | TRUE gate line as hardware gate on the prescaler, enabled when high (or gie inverts) |
| `int end_mode` | One of the `DQ_EM_XXX` end mode constants – determines count termination (or continuous if enabled)<br>DQ_EM_CR0 0<br>End when CR=CR0<br>DQ_EM_CR1<br>End when CR=CR1<br>DQ_EM_FFF<br>End when CR=0xFFFFFFFF<br>DQ_EM_PC<br>End when CR=0<br>DQ_EM_TBR<br>End when TBR=0<br>DQ_EM_GT |

| | |
|---|---|
| | End when H->L on gate (not supported) |
| `int lr` | Load register value – initial value of CR, in continuous modes LR will be reloaded into CR each time |
| `int *cfg` | Output parameter that receives the configuration value |

**Output:**

| | |
|---|---|
| `int *cfg` | the configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid, `mode` is not one of the valid `DQ_CM` constants, `end_mode` is not one of the valid `DQ_EM` constants, or `cfg` is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets up the configuration for a counter in a 601 layer.

**Note:**

None.

### *4.44.14 DqAdv601CfgForGeneralCounting*

**Syntax:**

```
int DqAdv601CfgForGeneralCounting(int hd, int devn, int counter,
int startmode, int ps, int cr0, int cr1, int dbg, int dbc,
int iie, int gie, int oie, int extclk, int trig, int trs,
int enc, int gated, int re, int end_mode, int lr, int *cfg)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 7) |
| `int startmode` | DQ_PL601_SM### const |
| `int ps` | Prescaler register value |
| `int cr0` | Compare register 0 value |
| `int cr1` | Compare register 1 value |
| `int dbg` | Input debouncing gate register value |
| `int dbc` | Input debouncing clock register value |
| `int iie` | `TRUE` to turn on pre-inversion of the input pin |
| `int gie` | `TRUE` to turn on pre-inversion of the gate pin |
| `int oie` | `TRUE` to turn on post-inversion of the output pin |
| `int extclk` | `TRUE` to use external clock |

| | |
|---|---|
| `int trig` | `TRUE` to use trigger |
| `int trs` | `TRUE` to use external trigger source (only w/ a triggered mode) |
| `int enc` | `TRUE` to use auto-clear at the end of the count (only w/ a triggered mode) |
| `int gated` | TRUE gate line as hardware gate on the prescaler, enabled when high (or gie inverts) |
| `int re` | `TRUE` to enable re-load (continuous operation) |
| `int end_mode` | One of the `DQ_EM_XXX` end mode constants |
| `int lr` | Load register value |
| `int *cfg` | Output parameter that receives the configuration value |

**Output:**

| | |
|---|---|
| `int *cfg` | the configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid, `end_mode` is not one of the valid `DQ_EM` constants, or `cfg` is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up the configuration for a counter in a 601 layer for general counting, without period counting or timebase division. For an extended parameter description, see DqAdv601ConfigCounter.

**Note:**

None.

### *4.44.15 DqAdv601CfgForBinCounter*

**Syntax:**

```
int DqAdv601CfgForBinCounter(int hd, int devn,
int counter, int startmode, int ps, int cr0, int cr1,
int tbr, int dbg, int dbc, int iie, int gie, int extclk,
int trig, int trs, int enc, int gated, int re,
int end_mode, int lr, int *cfg)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 7) |
| `int startmode` | DQ_PL601_SM### const |
| `int ps` | Prescaler register value |
| `int cr0` | Compare register 0 value |
| `int cr1` | Compare register 1 value |

| | |
|---|---|
| `int tbr` | Interval divider for timebase register |
| `int dbg` | Input debouncing gate register value |
| `int dbc` | Input debouncing clock register value |
| `int iie` | `TRUE` to turn on pre-inversion of the input pin |
| `int gie` | `TRUE` to turn on pre-inversion of the gate pin |
| `int extclk` | `TRUE` to use external clock |
| `int trig` | `TRUE` to use trigger |
| `int trs` | `TRUE` to use external trigger source (only w/ a triggered mode) |
| `int enc` | `TRUE` to use auto-clear at the end of the count (only w/ a triggered mode) |
| `int gated` | TRUE gate line as hardware gate on the prescaler, enabled when high (or gie inverts) |
| `int re` | `TRUE` to enable re-load (continuous operation) |
| `int end_mode` | One of the `DQ_EM_XXX` end mode constants |
| `int lr` | Load register value |
| `int *cfg` | Output parameter that receives the configuration value |

**Output:**

| | |
|---|---|
| `int *cfg` | the configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid, `end_mode` is not one of the valid `DQ_EM` constants, or `cfg` is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up the configuration for a counter in a 601 layer for getting the number of counts during a specified timeframe. Read the register for an immediate measurement. For an extended parameter description, see DqAdv601ConfigCounter.

**Note:**

None.

### *4.44.16 DqAdv601CfgForQuadrature*

**Syntax:**

```
int DqAdv601CfgForQuadrature(int hd, int devn,
int counter, int startmode, int tbr, int dbg, int dbc, int iie, int
gie, int end_mode, int lr, int *cfg)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 7) |
| `int startmode` | DQ_PL601_SM### const |
| `int tbr` | Interval divider for timebase register |
| `int dbg` | Input debouncing gate register value |
| `int dbc` | Input debouncing clock register value |
| `int iie` | `TRUE` to turn on pre-inversion of the input pin |
| `int gie` | `TRUE` to turn on pre-inversion of the gate pin |
| `int end_mode` | One of the `DQ_EM_XXX` end mode constants |
| `int lr` | Sets quadrature midpoint – usually 0x80000000 |
| `int *cfg` | Output parameter that receives the configuration value |

**Output:**

| | |
|---|---|
| `int *cfg` | the configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid, `end_mode` is not one of the valid `DQ_EM` constants, or `cfg` is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up the configuration for a counter in a 601 layer for quadrature decoding. For an extended parameter description, see DqAdv601ConfigCounter.

**Note:**

None.

### *4.44.17 DqAdv601CfgForHalfPeriod*

**Syntax:**

```
int DqAdv601CfgForHalfPeriod(int hd, int devn,
int counter, int startmode, int tbr, int dbg, int dbc,
int iie, int gie, int trig, int trs, int enc, int gated,
int re, int end_mode, int *cfg)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 7) |
| `int startmode` | DQ_PL601_SM### const |
| `int tbr` | Interval divider for timebase register |
| `int dbg` | Input debouncing gate register value |
| `int dbc` | Input debouncing clock register value |
| `int iie` | `TRUE` to turn on pre-inversion of the input pin |
| `int gie` | `TRUE` to turn on pre-inversion of the gate pin |
| `int trig` | `TRUE` to use trigger |
| `int trs` | `TRUE` to use external trigger source (only w/ a triggered mode) |
| `int enc` | `TRUE` to use auto-clear at the end of the count (only w/ a triggered mode) |
| `int gated` | TRUE gate line as hardware gate on the prescaler, enabled when high (or gie inverts) |
| `int re` | `TRUE` to enable re-load (continuous operation) |
| `int end_mode` | One of the `DQ_EM_XXX` end mode constants |
| `int *cfg` | Output parameter that receives the configuration value |

**Output:**

| | |
|---|---|
| `int *cfg` | the configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid, `end_mode` is not one of the valid `DQ_EM` constants, or `cfg` is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up the configuration for a counter in a 601 layer for a half period capture to measure waveform width. For an extended parameter description, see DqAdv601ConfigCounter.

**Note:**

None.

### *4.44.18 DqAdv601CfgForPeriodMeasurment*

**Syntax:**

```
int DqAdv601CfgForPeriodMeasurment(int hd, int devn,
int counter, int startmode, int pc, int tbr,
int dbg, int dbc, int iie, int gie, int trig, int trs,
int enc, int gated, int re, int end_mode, int *cfg)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 7) |
| `int startmode` | DQ_PL601_SM### const |
| `int pc` | Period counter value |
| `int tbr` | Interval divider for timebase register |
| `int dbg` | Input debouncing gate register value |
| `int dbc` | Input debouncing clock register value |
| `int iie` | `TRUE` to turn on pre-inversion of the input pin |
| `int gie` | `TRUE` to turn on pre-inversion of the gate pin |
| `int trig` | `TRUE` to use trigger |
| `int trs` | `TRUE` to use external trigger source (only w/ a triggered mode) |
| `int enc` | `TRUE` to use auto-clear at the end of the count (only w/ a triggered mode) |
| `int gated` | TRUE gate line as hardware gate on the prescaler, enabled when high (or gie inverts) |
| `int re` | `TRUE` to enable re-load (continuous operation) |
| `int end_mode` | One of the `DQ_EM_XXX` end mode constants |
| `int *cfg` | Output parameter that receives the configuration value |

**Output:**

| | |
|---|---|
| `int *cfg` | the configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid, `end_mode` is not one of the valid `DQ_EM` constants, or `cfg` is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up the configuration for a counter in a 601 layer for getting period measurements. For an extended parameter description, see DqAdv601ConfigCounter.

**Note:**

None.

### *4.44.19 DqAdv601CfgForPWM*

**Syntax:**

```
int DqAdv601CfgForPWM(int hd, int devn,
int counter, int startmode, int sampwidth, int ps,
int cr0, int cr1, int dbg, int dbc, int iie, int gie,
int oie, int extclk, int trig, int trs, int enc, int gated,
int re, int end_mode, int lr, int *cfg)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 7) |
| `int startmode` | DQ_PL601_SM### const |
| `int sampwidth` | DQ_PL601_SW## const<br>DQ_PL601_SW/8/16/32 specify sample width for period update. |
| `int ps` | Prescaler register value |
| `int cr0` | Compare register 0 value |
| `int cr1` | Compare register 1 value |
| `int dbg` | Input debouncing gate register value |
| `int dbc` | Input debouncing clock register value |
| `int iie` | `TRUE` to turn on pre-inversion of the input pin |
| `int gie` | `TRUE` to turn on pre-inversion of the gate pin |
| `int oie` | `TRUE` to turn on post-inversion of the output pin |
| `int extclk` | `TRUE` to use external clock |
| `int trig` | `TRUE` to use trigger |
| `int trs` | `TRUE` to use external trigger source (only w/ a triggered mode) |
| `int enc` | `TRUE` to use auto-clear at the end of the count (only w/ a triggered mode) |
| `int gated` | TRUE gate line as hardware gate on the prescaler, enabled when high (or gie inverts) |
| `int re` | `TRUE` to enable re-load (continuous operation) |
| `int end_mode` | One of the `DQ_EM_XXX` end mode constants |
| `int lr` | Load register value |
| `int *cfg` | Output parameter that receives the configuration value |

**Output:**

| | |
|---|---|
| `int *cfg` | the configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid, `end_mode` is not one of the valid `DQ_EM` constants, or `cfg` is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |

| | |
|---|---|
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up the configuration for a counter in a 601 layer for PWM output generation.

**Note:**

CR1 specifies the PWM period and remains fixed while CR0 is updated.

## *4.44.20 DqAdv601CfgForPWMTrain*

**Syntax:**

```
int DqAdv601CfgForPWMTrain(int hd, int devn,
int counter, int startmode, int sampwidth, int ps,
int n_pulses, int cr0, int cr1, int dbg, int dbc,
int iie, int gie, int oie, int extclk, int trig, int trs,
int enc, int gated, int re, int end_mode, int lr,
int *cfg)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 7) |
| `int startmode` | DQ_PL601_SM### const |
| `int sampwidth` | DQ_PL601_SW## const<br>DQ_PL601_SW/8/16/32 specify sample width for period update. |
| `int ps` | Prescaler register value |
| `int n_pulses` | Number of pulses to produce |
| `int cr0` | Compare register 0 value |
| `int cr1` | Compare register 1 value |
| `int dbg` | Input debouncing gate register value |
| `int dbc` | Input debouncing clock register value |
| `int iie` | `TRUE` to turn on pre-inversion of the input pin |
| `int gie` | `TRUE` to turn on pre-inversion of the gate pin |
| `int oie` | `TRUE` to turn on post-inversion of the output pin |
| `int extclk` | `TRUE` to use external clock |
| `int trig` | `TRUE` to use trigger |
| `int trs` | `TRUE` to use external trigger source (only w/ a triggered mode) |
| `int enc` | `TRUE` to use auto-clear at the end of the count (only w/ a triggered mode) |
| `int gated` | TRUE gate line as hardware gate on the prescaler, enabled when high (or gie inverts) |
| `int re` | `TRUE` to enable re-load (continuous operation) |
| `int end_mode` | One of the `DQ_EM_XXX` end mode constants |

| | |
|---|---|
| `int lr` | Load register value |
| `int *cfg` | Output parameter that receives the configuration value |

**Output:**

| | |
|---|---|
| `int *cfg` | the configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid, `end_mode` is not one of the valid `DQ_EM` constants, or `cfg` is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up the configuration for a counter in a 601 layer for PWM output with a set number of re-triggerable pulses.

**Note:**

PC counter only valid in CM_CT/CM_TCT/CM_RTCT CM_ECT/CM_TECT modes.
When desired number of pulses is outputted

- in CM_CT/CM_ECT mode CTR_EN is cleared and re-enabling CTR_EN will restart counter
- in CM_TCT/CM_RTCT/CM_TECT trigger condition is cleared and low->high transition on the gate will restart counter

Limitations: CTU_CR1 should be >= 5 for the CM_CT/CM_TCT/CM_RTCT and for the CM_ECT/CM_TECT value loaded to CTU_CR0 should correspond to >75.75nS based on expected external frequency specifies the PWM period and remains fixed while CR0 is updated.

### *4.44.21 DqAdv601CfgForTPPM*

**Syntax:**

```
int DqAdv601CfgForTPPM(int hd, int devn,
int counter, int startmode, int tbr, int dbg, int dbc,
int iie, int gie, int re, int end_mode, int *cfg)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 7) |
| `int startmode` | DQ_PL601_SM### const |
| `Int tbr` | Interval divider for timebase register |
| `int dbg` | Input debouncing gate register value |
| `int dbc` | Input debouncing clock register value |
| `int iie` | `TRUE` to turn on pre-inversion of the input pin |
| `int gie` | `TRUE` to turn on pre-inversion of the gate pin |
| `int re` | `TRUE` to enable re-load (continuous operation) |
| `int end_mode` | One of the `DQ_EM_XXX` end mode constants |
| `int *cfg` | Output parameter that receives the configuration value |

**Output:**

| | |
|---|---|
| `int *cfg` | the configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid, `end_mode` is not one of the valid `DQ_EM` constants, or `cfg` is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up the configuration for a counter in a 601 layer for TPPM mode.

**Note:**

### *4.44.22 DqAdv601SetAltClocks*

**Syntax:**

```
int DqAdv601SetAltClocks(int hd, int devn, int counter, uint32
timebase, uint32 prescaler)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 7) |
| `Uint32 timebase` | One of the constants listed below indicating which clock source to use for the timebase register |
| `uint32 prescaler` | One of the constants listed below indicating which clock source to use for the prescaler register |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid, `timebase` or `prescaler` is not one of the listed constants |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function configures the `counter` to connect an alternate clock source to either the timebase or prescaler registers. The following clock sources are allowed for the timebase register:

```
DQ_EXT_SYNC0        // sync bus 0
DQ_EXT_SYNC1        // sync bus 1
DQ_EXT_SYNC2        // sync bus 2
DQ_EXT_SYNC3        // sync bus 3
DQ_STR_GT1          // debounced gate pin
0 or DQ_PL_601_BASE  // 66MHz    <-- default value
```

The following clock sources are allowed for the prescaler register:

```
DQ_EXT_SYNC0        // sync bus 0
DQ_EXT_SYNC1        // sync bus 1
DQ_EXT_SYNC2        // sync bus 2
DQ_EXT_SYNC3        // sync bus 3
0 or DQ_PL_601_BASE  // 66MHz    <-- default value
```

**Note:**

This function must be called after the DqAdv601CfgFor...()function or DqAdv601ConfigCounter() function. Set up all other CT-601 sync routing (e.g. DqAdvRouteSyncIn(), DqCmdSetSyncRt() ) before calling this function.

## *4.44.23 DqAdv601ConfigEvents*

**Syntax:**

```
int DAQLIB DqAdv601ConfigEvents(int hd, int devn, int channel, int flags,
event601_t event, uint32 mode, uint32* param)
```

**Command:**

Specify what asynchronous notification events user wants to receive from the layer

**Input**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number |
| `int channel` | Channel number to configure |
| `int flags` | <rsvd> |
| `event601_t event` | Event type |
| `uint32 mode` | A parameter that defines sub-event |
| `uint32* param` | Up to seven additional parameters depending on event type |

**Output**

None

**Returns**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601/602 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function configures events to be sent from the layer.

The following events are defined:

```
// Types of event
typedef enum {
    EV601_CLEAR = 0x100,          // clear all events



    // bus or RT controller event
    EV601_COUNT_COMPLETE,       // Event when programmed count is reached (end-
mode)
    EV601_CR0_LESSTHEN,         // Count is less than CR0
    EV601_CR0_EXCEEDED,         // Count exceeded CR0
    EV601_CR1_EXCEEDED,         // Count exceeded CR1
    EV601_DATA_AVAILABLE,       // Data available in CRH/CRL (half) period
calculated
    EV601_INP_TRANSITION,       // Input line transition detected
    EV601_GATE_TRANSITION       // Gate line transition detected
} event601_t;
```

For `EV601_INP_TRANSITION` and `EV601_GATE_TRANSITION` use

```
#define EV601_LOW_TO_HI     1
#define EV601_HI_TO_LOW     2
```

To select whether to send event upon input or gate line goes from the logical zero to one or back or both.

If an event happens, a packet of the following structure is sent from the cube to notify the host about this event:

```
typedef struct {
    uint8   dev;    // device
    uint8   ss;     // subsystem
    uint16  size;   // data size
    uint32  event;  // event type or command and additional flags
    uint8   data[]; // data: for CT-601 layer of EV601_ID type
} DQEVENT, *pDQEVENT
```

Event data is always layer-specific. In this case EV601_ID structure is used to inform user application about what type of event has happened on DNx-CT-601 layer:

```
// Event data for 601 layer
typedef struct {
    uint32 chan;    // channel information
    uint32 evtype;  // type of the event
    uint32 count;   // counter register (CR)
    uint32 cr0;     // CR0 (W) or CRL (R) register
    uint32 cr1;     // CR1 (W) or CRH (R) register
    uint32 sts;     // applicable status register
    uint32 tstamp;  // timestamp of event
    uint32 size;    // size of the following data in bytes
    uint32 data[];  // data to follow
} EV601_ID, *pEV601_ID;
```

Every type of the event requires specific parameters (or none) and sends a notification with specific fields and data.

For all types of events:

`DQPKT:`

dqCounter = number of event on per channel/per device basis [1..65535]
dqCommand = DQCMD_EVENT | DQ_REPLY

`DQEVENT:`
dev = device number
ss = DQ_SS0IN
event = one of the event types described below
size = size of the following EV553_ID structure including data

`EV601_ID:`
chan = channel
evtype = event subtype

**Note:**

1. Maximum event rate is limited to 50us or 20kHz to avoid interrupt overrun. If this interrupt rate is exceeded interrupt routine disables events and firmware status flag STS_FW is set to STS_FW_OVERLOAD.
2. <data> in `EV601_ID` and <param> in the function call are reserved for the future extensions.

### *4.44.24 DqAdv601WaitForEvents*

**Syntax:**

```
int DAQLIB DqAdv601WaitForEvents(int hd, int devn, event601_t* event,
uint32* count, uint32* crl, uint32* crh, uint32* sts, uint32* tstamp)
```

**Command:**

Wait for the next CT-601 event to occur

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number |

**Output:**

| | |
|---|---|
| `event601_t* event` | Type of the received event |
| `uint32* count` | Content of count register |
| `uint32* crl` | Content of low capture register |
| `uint32* crh` | Content of high capture register |
| `uint32* sts` | Content of status register |
| `uint32* tstamp` | Timestamp of event |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601/602 |

| | |
|---|---|
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | No event was received for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**
This function waits for the next 601 event to occur. It returns the content of registers as well as timestamp at the time the event occurred

It returns DQ_TIMEOUT_ERROR if no event occurred within the time delay specified in `DqAddIOMPort().`

## 4.45 DNA-CT-602 layer

This section defines functions that are unique to the CT-602. All of the DqAdv601 functions may also be used with the CT-602, the only difference being that the maximum `counter` value is 3.

### 4.45.1 DqAdv602ConfigDo

**Syntax:**

```
int DqAdv602ConfigDo(int hd, int devn, int counter, uint32 buffer,
uint32 enables)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 3) |
| `uint32 buffer` | bit1 controls CLKOUT pin, bit0 controls TRIGOUT pin<br>0 = output driver is turned OFF (high impedance)<br>1 = output driver is turned ON |
| `uint32 enables` | bit1 controls CLKOUT pin, bit0 controls TRIGOUT pin<br>0 = use as normal Counter/Timer output<br>1 = use as DIO |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-602 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets the configuration of the CT-602's Digital outputs.

Allows the user to turn the buffer drivers ON/OFF and to switch any output between normal Counter/Timer operation and general purpose Digital Output.

**Note:**

### 4.45.2 DqAdv602CfgPWMDuringMeasurement

**Syntax:**

```
int DqAdv602CfgPWMDuringMeasurement (int hd, int devn, int counter,
int ps, int cr0, int cr1, int oie, int re)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 3) |
| `int ps` | Prescaler register value |
| `int cr0` | Compare register 0 value |
| `int cr1` | Compare register 1 value |
| `int oie` | `TRUE` to turn on post-inversion of the output pin |
| `int re` | `TRUE` to enable re-load (continuous operation) |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-602 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function configures PWM output of the CT-602 counter during measurement (input) mode. On CT-602 main counter can be used during measurement modes to supply continuous pulse train to the output pin, it always internally uses EM_CR1 end-mode condition and starts when CTR_EN is set to 1. In MMCO mode prescaler is always set to use 66MHz internal clock and bypassed when PS=0; GATE input still can be used if gate is enabled during measurement mode configuration. Both CR0 and CR1 should be programmed for proper operation, one-shot mode can be set using "re=0".

**Note:**

PWM output during measurement should be enabled by DqAdv602EnPWMDuringMeasurement function.

### *4.45.3 DqAdv602EnPWMDuringMeasurement*

**Syntax:**

```
int DqAdv602EnPWMDuringMeasurement (int hd, int devn, int counter,
int en)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 3) |
| `int en` | enable(1)/disable(0) PWM output |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-602 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function enables/disables PWM output during measurement(input) mode.

**Note:**

PWM output during measurement should be configured via DqAdv602CfgPWMDuringMeasurement.

### *4.45.4 DqAdv602ReadDioIn*

**Syntax:**

```
int DqAdv602ReadDioIn(int hd, int devn, int counter, uint32* din)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 3) |

**Output:**

| | |
|---|---|
| `uint32* din` | digital input value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-602 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads the current state of the GATE and CLKIN pins. The CLKIN signal will appear on bit 1 of *din and the GATE signal will appear on bit 0 of *din.

**Note:**

None.

### *4.45.5 DqAdv602ReadDioOut*

**Syntax:**

```
int DqAdv602ReadDioOut(int hd, int devn, int counter, uint32* din)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 3) |

**Output:**

| | |
|---|---|
| `uint32* dout` | digital output value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-602 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid |

| | |
|---|---|
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads back the current state of the CLKOUT and TRIGOUT pins. The state of the CLKOUT signal will appear on bit1 of *din and the TRIGOUT signal will appear on bit 0 of *din.

**Note:**

None.

### 4.45.6 DqAdv602SetEvents

**Syntax:**

```
int DqAdv602SetEvents(int hd, int devn, int evt_chan, int flags,
pTR602_CFG event, uint32* param)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int evt_chan` | Counter number (0 - 3) |
| `int flags` | Additional event flags <reserved> |
| `pEV650_CFG event` | Event configuration. |
| `uint32* param` | Additional parameters for some events <reserved> |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-602 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function configures TRIGOUT pin for the corresponding channel on CT-602. TRIGOUT pin can carry pulse that is based on one of the 24 digital sources. Pulse length can be set from 1 to 14 66MHz clocks, 1uS or 1mS, one or two pulses with programmable delay can be generated. Event can be set to run in one-time or continuous mode

**Note:**

This function assumes that the TRIGOUT pin has had its output driver is turned ON and is in Counter/Timer mode  by using the DqAdv602ConfigDo() function.

### *4.45.7 DqAdv602SetTermination*

**Syntax:**

```
int DqAdv602SetTermination(int hd, int devn, uint32 txterm, uint32
rxterm)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 txterm` | Bit field that turns on termination for the output signals. The bit position corresponds to the channel number. |
| `uint32 rxterm` | Bit field that turns on termination for the input signals. The bit position corresponds to the channel number. |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-602 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function configures the termination on the CT-602’s transmit and receive lines. The bit postion determines which channel has the termination enabled,
e.g. a 1 in bit postion 2 turns on the termination for channel 2.

**Note:**

None.

### *4.45.8 DqAdv602WriteDioOut*

**Syntax:**

```
int DqAdv602WriteDioOut(int hd, int devn, int counter, uint32 dout,
uint32 *last_dout)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 3) |
| `uint32 dout` | new value to send to dout pins, bit1 = CLKOUT, bit0 = TRIGOUT. |
| `uint32 *last_dout` | last dout value |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-602 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function writes to the general-purpose DO port on the specified channel and returns the last written value.

This function assumes that the outputs have been selected to be in the general-purpose DO mode by using the DqAdv602ConfigDo function.

**Note:**

None.

## *4.46 DNA-CT-602-804 layer*

In addition to the functions listed in the previous 2 sections, the CT-602-804 additionally supports the following 3 functions.

### *4.46.1 DqAdv602SetGPSSConfig*

**Syntax:**

```
int DqAdv602SetGPSSConfig(int hd, int devn, int channel,
pCT602_GPSS_CFG pCfg)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | Channel to apply config to (0 - 3) |
| `pCT602_GPSS_CFG config` | channel configuration |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-602-804 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Configures CT-602-804 card to GPSS mode of operations.

Pinout (37-pin connector):

| Channel | 0 | 1 | 2 | 3 | TX | TX+1(*) | RX | RX+1(*) |
|---|---|---|---|---|---|---|---|---|
| CLK-IN | 4 | 9 | 13 | 18 | SER CLK | ACK | SER DATA | SER CLK |
| return | 23 | 28 | 32 | 37 | | | | |
| GATE (TRIG-IN) | 2 | 7 | 11 | 16 | EXT TRG | - | FRM SYNC | EXT TRG |
| return | 21 | 26 | 30 | 35 | | | | |
| CLK-OUT | 1 | 6 | 10 | 15 | SER DATA | SER CLK | SER CLK | - |
| return | 20 | 25 | 29 | 34 | | | | |
| TRIG-OUT | 3 | 8 | 12 | 17 | FRM SYNC | - | ACK | - |
| return | 22 | 27 | 31 | 36 | | | | |

* signals are optional. If not used the channel can be used as a regular counter

SER CLK = serial clock
SER DATA = serial data
ACK = acknowledge
FRM SYNC = frame sync strobe
All of them can be either input or output depends on the mode of operation
A channel can be either RX or TX but not both

**Note:**

Refer to Sample602GPSS for configuring GPSS on CT-602-804.

### *4.46.2 DqAdv602RecvGPSSMessage*

**Syntax:**

```
int DqAdv602RecvGPSSMessage(int hd, int devn, int chnl, int flags,
uint8 *data, int size, int *received, int *available)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | device channel |
| `int flags` | miscellaneous flags |
| `uint8 *data` | received data |
| `int size` | number of bytes to read |

**Output:**

| | |
|---|---|
| `int* received` | number of bytes received |
| `int* available` | number of bytes available in the buffer for further reads |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-602-804 |
| `DQ_BAD_PARAMETER` | counter number is invalid |

| | |
|---|---|
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Reads a block of data from a synchronous serial port.

If a bit `CT602_READCHNL_CONT` is set in `<received>` it means that the frame reception is not completed.

**Notes:**

For a fixed frame size multiple frames can be read.

For a variable frame size only a single frame can be read.

For a variable word size a single frame can be read.

### *4.46.3 DqAdv602SendGPSSMessage*

**Syntax:**

```
int DqAdv602SendGPSSMessage(int hd, int devn, int chnl, int flags,
uint8 *data, int size, int* written, int* available)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | device channel |
| `int flags` | miscellaneous flags |
| `int size` | number of words to write |

**Output:**

| | |
|---|---|
| `uint8 *data` | data to send |
| `int* written` | number of words written |
| `int* available` | Available room in the buffer for further writes (multiply by 4 for bytes) |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-602-804 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Writes a block of data to a synchronous serial port.
If (`size == written`) then all data fit into the outbound fifo.

**Notes:**

For a fixed frame size multiple frames can be read.
For a variable frame size only a single frame can be read.
For a variable word size a single frame can be read.

## 4.47 DNA-QUAD-604 layer

### 4.47.1 DqAdv604StartCounter

**Syntax:**

```
int DqAdv604StartCounter(int hd, int devn, int counter)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 3) |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-604 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function starts the specified counter.

**Note:**

None.

### *4.47.2 DqAdv604StopCounter*

**Syntax:**

```
int DqAdv604StopCounter(int hd, int devn, int counter)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 3) |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-604 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function stops the specified counter.

**Note:**

None.

### *4.47.3 DqAdv604ClearCounter*

**Syntax:**

```
int DqAdv604ClearCounter(int hd, int devn, int counter)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 3) |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-604 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function clears the current count (CR) of the specified counter.

**Note:**

None.

### 4.47.4 DqAdv604Read

**Syntax:**

```
int DqAdv604Read (int hd, int devn, int CLSize, uint32* cl, uint32
*data)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | Number of channels to read |
| `uint32 *cl` | Channel list |
| `uint32 *data` | Pointer to receive data values |

**Output:**

| | |
|---|---|
| `uint32 *data` | Read data values |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-604 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid, `reg` is not one of the `QDU` constants, or `value` is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads the current count values from the counters specified in the channel list. It is used for immediate mode.

**Note:**

None

### 4.47.5 DqAdv604SetChannelCfg

**Syntax:**

```
int DqAdv604SetChannelCfg(int hd, int devn, int ss, int counter,
pDQCHNLSET_604_ pCfg, uint32 *cfg)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ss` | Device subsystem |

| | |
|---|---|
| `int counter` | Counter number (0 - 3) |
| `pDQCHNLSET_604_ pCfg` | Pointer to channel configuration structure |
| `int *cfg` | Pointer to cfg values |

**Output:**

| | |
|---|---|
| `int *cfg` | Returned config/status |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-604 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This internal function sets the low level configuration for a counter channel.

**Note:**

Currently *cfg has no meaning but may return extra config/status values in the future.

### *4.47.6 DqAdv604ReadRegisterValue*

**Syntax:**

```
int DqAdv604ReadRegisterValue(int hd, int devn, int counter, uint32
reg, uint32 *value)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 3) |
| `int reg` | One of the `QD604_QDU_XXX` constants indicating which register to read |
| `uint32 *value` | Pointer to receive data value |

**Output:**

| | |
|---|---|
| `uint32 *value` | Read data value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-604 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid, `reg` is not one of the `QDU` constants, or `value` is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads the value from a register `reg` of the `counter` relative to layer `devn`/counter channel. The following registers are allowed:

```
QD604_QDU_STR
QD604_QDU_CTR
QD604_QDU_CCR
QD604_QDU_CR
QD604_QDU_LR
QD604_QDU_CR0
QD604_QDU_CR1
QD604_QDU_TBR
QD604_QDU_QED
QD604_QDU_IFWR
```

**Note:**

None.

### 4.47.7 DqAdv604WriteRegisterValue

**Syntax:**

```
int DqAdv604WriteRegisterValue(int hd, int devn, int counter, uint32
reg, uint32 value)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 3) |
| `int reg` | One of the `DQ_CTU_XXX` constants indicating which register to write |
| `uint32 value` | Value to write |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-604 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid, `reg` is not one of the `QDU` constants, or `value` is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads the value from a register `reg` of the `counter` relative to layer `devn`/counter channel. The following registers are allowed:

```
QD604_QDU_CTR
QD604_QDU_CCR
QD604_QDU_LR
QD604_QDU_IDBA
QD604_QDU_IDBB
QD604_QDU_IDBZ
QD604_QDU_IDBT
QD604_QDU_CR0
QD604_QDU_CR1
QD604_QDU_TBR
QD604_QDU_QED
QD604_QDU_OW
QD604_QDU_INC
QD604_QDU_IFWR
```

**Note:**

None.

### *4.47.8 DqAdv604ConfigCounter*

**Syntax:**

```
int DqAdv604ConfigCounter(int hd, int devn,
int counter, int en, int lr, int cr0, int cr1, int tbr,
int idba, int idbb, int idbz, int idbt, int inv_a, int inv_b,
int inv_z, int inv_t, int inv_do0, int inv_do1,
int mode, int rl_mode, int evt_b, int evt_src,
int tb_src, int clkout_en, int clkout_mode, int trg_src,
int trg_clr, int trgout_en, int trgout_mode, int gtstart_en,
int gtstop_en, int out_width, int qe_mode, int qe_delay,
int qe_err, int qe_swap, int ts_mode, int end_mode,
int inc, int *cfg)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| int hd | Handle to the IOM received from DqOpenIOM() |
| int devn | Layer inside the IOM |
| int counter | Counter number (0 - 3) |
| int en | Enable counter |
| int lr | Load value |
| int cr0 | Compare register 0 value |
| int cr1 | Compare register 1 value |
| int idba | A debounce |
| int idbb | B debounce |
| int idbz | Z debounce |
| int idbt | T debounce |
| int inv_a | TRUE to turn on inversion of the A pin |
| int inv_b | TRUE to turn on inversion of the B pin |

| | |
|---|---|
| int inv_z | TRUE to turn on inversion of the Z pin |
| int inv_t | TRUE to turn on inversion of the T pin |
| int inv_do0 | TRUE to turn on inversion of the DO0/TRGOUT pin |
| int inv_do1 | TRUE to turn on inversion of the DO1/CLKOUT pin |
| int mode | One of the QDU_CM_XXX mode constants<br>QDU_CM_CDU<br>count-up A input<br>QDU_CM_CDN<br>count-down A input<br>QDU_CM_DC<br>direction counter<br>QDU_CM_QE<br>quadrature encoder<br>QDU_CM_TCDU<br>triggered count-up A input<br>QDU_CM_TCDN<br>triggered count-down A input<br>QDU_CM_TDC<br>triggered direction counter<br>QDU_CM_TQE<br>triggered quadrature encoder<br>QDU_CM_RTCDU<br>re-triggered count-up A input<br>QDU_CM_RTCDN<br>re-triggered count-down A input<br>QDU_CM_RTDC<br>re-triggered direction counter<br>QDU_CM_RTQE<br>re-triggered quadrature encoder |
| int rl_mode | One of the QDU_CRM_XXX reload mode constants<br>QDU_CRM_LR<br>load register<br>QDU_CRM_CR01<br>CR0 for the down count and CR1 for the up count<br>QDU_CRM_CR10<br>CR1 for the down count and CR0 for the up count<br>QDU_CRM_NR<br>no reload (counter keeps its value)<br>QDU_CRM_OTR<br>One-time reload (load counter only once per every start trigger and following event) |
| int evt_b | One of the QDU_EB_XXX event behavior constants<br>QDU_EB_NO |

| | |
|---|---|
| | no event<br>QDU_EB_RE<br>rising edge on the debounced input<br>QDU_EB_RE_LL<br>rising edge on the debounced input followed by A/B = low/low<br>QDU_EB_RE_LH<br>rising edge on the debounced input followed by A/B = low/high<br>QDU_EB_RE_HL<br>rising edge on the debounced input followed by A/B = high/low<br>QDU_EB_RE_HH<br>rising edge on the debounced input followed by A/B = high/high |
| int evt_src | One of the QDU_ES_XXX event source constants<br>QDU_ES_Z<br>Z input<br>QDU_ES_T<br>T input |
| int tb_src | One of the QDU_TBS_XXX timebase source constants<br>QDU_TBS_66M<br>internal 66Mhz timebase<br>QDU_TBS_TRIG<br>debounced TRIGIN pin<br>QDU_TBS_SYNC0<br>SYNC0 1ine<br>QDU_TBS_SYNC1<br>SYNC1 1ine<br>QDU_TBS_SYNC2<br>SYNC2 1ine<br>QDU_TBS_SYNC3<br>SYNC3 1ine |
| int clkout_en | TRUE to turn on CLKOUT |
| int clkout_mode | One of the QDU_COM_XXX clock mode constants<br>QDU_COM_CR1G<br>CLKOUT=1 if CR>CRl<br>QDU_COM_CR1E<br>CLKOUT=1 if CR=CRl<br>QDU_COM_1X |

| | |
|---|---|
| | CLKOUT = lx QE clock<br>QDU_COM_2X<br>CLKOUT = 2x QE clock<br>QDU_COM_4X<br>CLKOUT = 4x QE clock<br>QDU_COM_IE<br>CLKOUT = Index event<br>QDU_COM_N<br>Inc/Dec N - pulse when CR increments/decrements by QDU_INC counts |
| int trg_src | One of the QDU_TRS_XXX trigger source constants<br>QDU_TRS_SW<br>software trigger source<br>QDU_TRS_HW<br>hardware trigger source |
| int trg_clr | TRUE to use auto-clear at the end of the count (only in triggered mode) |
| int trgout_en | TRUE to turn on TRGOUT |
| int trgout_mode | One of the QDU_TOM_XXX trigger output mode constants<br>QDU_TOM_CR0L<br>TRIGOUT=1 if CR<CR0<br>QDU_TOM_CR0E<br>TRIGOUT=1 if CR=CR0<br>QDU_TOM_GTS<br>TRIGOUT=global trigger status<br>QDU_TOM_STD<br>TRIGOUT=start trigger detected pulseQDU_TOM_DIR<br>QDU_TOM_DIR<br>TRIGOUT=direction (0=clockwise, 1= counter-clockwise)<br>QDU_TOM_EM<br>TRIGOUT=EM event |
| int gtstart_en | Enable global start trigger |
| int gtstop_en | Enable global stop trigger |
| int out_width | 16-bit value in 16.5Mhz clocks specifying CLKOUT/TRGOUT pulse width |
| int qe_mode | One of the QDU_QEM_XXX end mode constants |
| int qe_delay | 16-bit value in 16.5Mhz clocks specifying encoder delay |
| int qe_err | TRUE to enable encoder error detection |
| int qe_swap | TRUE to enable A/B input swapping |
| int ts_mode | One of the QDU_TS_XXX timestamp mode constants (only in buffered modes)<br>QDU_TSM_NOTS<br>no timestamps<br>QDU_TSM_TSADD |

| | |
|---|---|
| | add timestamp to every sample copied at EM event<br>QDU_TSM_TSONLY<br>put only timestamp into the FIFO at EM event |
| int end_mode | One of the QDU_EM_XXX end mode constants<br>QDU_EM_CR0<br>end, when CR<CR0<br>QDU_EM_CR1<br>end, when CR>CR1<br>QDU_EM_CR01<br>end, when counter value is less, then CR0 or more, then CR1<br>QDU_EM_IE<br>end on event (index or user)<br>QDU_EM_TBR<br>end, when time-base counter reaches 0<br>QDU_EM_INC<br>end, when counter changes by pre-defined number<br>QDU_EM_INF<br>indefinite wrap-around counter |
| int inc | 16-bit value used to specify CLKOUT or EM event |
| int *cfg | Output parameter that receives the configuration value |

**Output:**

| | |
|---|---|
| `int *cfg` | the configuration value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-601 |
| `DQ_BAD_PARAMETER` | `counter` number is invalid, `end_mode` is not one of the valid `DQ_EM` constants, or `cfg` is NULL |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up the configuration for a counter in a 604 layer.

**Note:**

None

### 4.47.9 DqAdv604SetWatermark

**Syntax:**

```
int DqAdv604SetWatermark(int hd, int devn, int counter)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int counter` | Counter number (0 - 3) |
| `uint32 dir` | Always DQSS0_IN |
| `uint16 len` | watermark level (0-1024) |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-604 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets the FIFO watermark level during buffered (ACB/Messaging) modes.

**Note:**

None.

### 4.47.10 DqAdv604ReadDioIn

**Syntax:**

```
int DqAdv604ReadDioIn(int hd, int devn, uint32 *din)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 *din` | Output parameter that receives the din value |

**Output:**

| | |
|---|---|
| `uint32 *din` | din value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-604 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |

| | |
|---|---|
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function returns the status of all digital input lines.

**Note:**

None.

### 4.47.11 DqAdv604ReadDioOut

**Syntax:**

```
int DqAdv604ReadDioOut(int hd, int devn, uint32 *dout)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 *dout` | Output parameter that receives the din value |

**Output:**

| | |
|---|---|
| `uint32 *dout` | din value |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-604 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function returns the status of all digital out lines.

**Note:**

None.

### 4.47.12 DqAdv604WriteDioOut

**Syntax:**

```
int DqAdv604WriteDioOut(int hd, int devn, uint32 dout, uint32
*last_dout)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 dout` | Dout value to write |
| `uint32 *last_dout` | Output parameter that receives the previous dout value |

**Output:**

| | |
|---|---|
| `uint32 *last_dout` | Previous dout value before write |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-604 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Updates status of all digital out lines returning last value.

**Note:**

None.

## 4.48 DNA-VR-608 layer

### 4.48.1 DqAdv608SetCfg

**Syntax:**

```
int DqAdv608SetCfg(int hd, int devn, uint32 chnl, uint32 front_cfg,
uint32 mode, pVR608_CFG extcfg)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chnl` | Channel (0 .. 7) |
| `uint32 front_cfg` | Front-end configuration |
| `uint32 mode` | Mode of operation |

**Output:**

(none)

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocatin buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-604 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up the configuration of VR-608 channel(s).

Settings can affect the individual **C**hannel, **P**air of channels (0&1 2&3 4&5 6&7), or the **L**ayer:

- C: Individual **C**hannel settings only affect that channel.
- P: Every **p**air of even and odd channels (i.e. channels 0 and 1, 2 & 3, 4 & 5, 6 & 7) must be configured to the same configuration. Writing a different parameter to the second channel will over write the previously configured channel. This is because every two channels share certain logic elements, so setting the front-end for channel 1 will also become the setting for channel 0, or in a more obvious example, a channel pair must both be set to either `VR608_MODE_COUNTER` mode or `VR608_MODE_DECODER` mode.
- L: **L**ayer settings affect all channels at once. So setting the ain_rate for channel 1 will actually change the sampling rate for channels 0 to 7.

The parameter `front_cfg` sets the zero crossing and the analog/adaptive peak threshold (APT) for the channel pair. The settings for this parameter are listed below in two groups. Each setting within these groups is mutually exclusive.

Zero crossing (Pair):

```
VR608_ZC_ONCHIP   // zero crossing detection enabled by the chip
VR608_ZC_FIXED    // zero level is set by the user
VR608_ZC_LOGIC    // Reserved: auto calculate as a fraction of AIn minmax
```

Adaptive peak threshold (Pair):

```
VR608_APT_ONCHIP  // adaptive threshold automatically set by front-end MAX9926
VR608_APT_FIXED   // threshold is selected by user writing threshold DACs
VR608_APT_LOGIC   // Reserved: APT selected automatically based the ADCs
```

The `mode` parameter has three different settings. These are described below.

Channels can be set into VR mode (a.k.a. `COUNTER` mode) where each channel counts independently or they can be set to `DECODER` mode where each channel pair (i.e. channels 0 and 1, etc.) is used to read a quadrature encoder as input. This setting must be the same for both channels of a channel pair.

```
VR608_MODE_COUNTER // counts number of teeth for each channel separately
VR608_MODE_DECODER // channel pairs together determine direction and position
```

If `COUNTER` mode is selected each channel must be assigned a mutually-exclusive submode which determine how the VR-608 latches data from the counter-timers. They can be latched when the time base expires, after N pulses, or when the Z-tooth is detected.

```
VR608_MODE_TIMED   // latch data or clock FIFO upon timebase expiration
VR608_MODE_NPULSE  // latch data or clock data into the FIFO each N pulses
VR608_MODE_ZPULSE  // latch data or clock data into the FIFO when Z-tooth is found
```

The last setting controlled by the `mode` parameter is whether to write to the FIFO. The counter registers can be written to the FIFO whenever data is latched. This optionally includes the timestamp.

```
VR608_FIFO_POS     // store position/count into the FIFO
VR608_FIFO_TS      // accompany FIFO data with the timestamps (any mode)
```

The extcfg parameter takes a `VR608_ExtCfg` structure for the remainder of the configuration settings. These can be set for the (L)ayer, (P)air of channels (0&1 2&3 4&5 6&7), or an individual (C)hannel:

| | | |
|---|---|---|
| `cfg_flags` | | Bit field: set the bits to mark which parameters below should be set. In case that a parameter is needed for a specific mode but the bit field is not set, the default value will be used. If a parameter is set but not in the valid range then the function will clear the bit for the first faulty parameter it finds and return a `DQ_BAD_PARAMETER_6` error code. |
| `adc_rate` | L | Sampling rate, in Hz, for all channels on this layer (not just this channel).<br>For all modes the maximum sampling rate value is `VR608_MAXCLFRQ` Hz.<br>The VR input pulse rate (not the sampling rate) should not exceeed `adc_rate/2` in `VR608_ZC_ONCHIP` mode or 18kHz in any other mode. |
| `tmode_rate` | C | `TIMED` mode rate: how often to store accumulated data scans, up to 1000 Hz. |
| `adc_mv_avg` | P | For `VR608_ZC_FIXED`, `VR608_*_LOGIC` modes, sets number of samples for moving average: `0` is off/every sample, `3` is every $2^3$=8 ADC samples.<br>The moving average can soften/smooth out noise seen from the sensor input line. The averaging engine requires a high ADC sampling rate (200kHz) for the VR-608 and slow VR sensor input pulse rate: 200kHz with 8x averaging the fastest VR sensor input pulse should be under 200kHz/(2*8) = 12500Hz. Moving averaging should not be used for a low VR-608 ADC sampling rate or if the VR sensor produces faster pulses; the averaging will over-soften and delay/shift/skew the readings and some parts will not be detected. |
| `zc_level` | C | For `VR608_ZC_FIXED` mode this value is the threshold level that is used. This value (-5V to 5V) will be directly written to the threshold DACs. The channel’s limiter circuit clips voltages above/below this range |
| `apt_th` | C | For `VR608_APT_FIXED` this sets the threshold level to between 0V - 5V |
| `apt_div` | C | For `VR608_APT_LOGIC` this configures the divider for AIn to have an automatic AOut threshold of 1 = ½, 2 = ¼, 3 = 1/8th, or 4 = 1/16th; by comparison, the `VR608_APT_ONCHIP` mode has an AOut threshold of 1/3. |
| `num_teeth` | C | The number of teeth on the gear / wheel. When set to 0 it returns teeth/second for the input data; a positive value of >=1 returns RPM. |
| `z_tooth_sz` | C | For all modes sets the size of the index tooth/z-tooth in teeth. This parameter is a correction factor to compensate for the number of teeth that the z-tooth substitutes (either missing teeth or a wider tooth). Values are restricted to between -3 to 3 teeth (if you require more please contact Support). |
| `sync_in` | - | Reserved to select the sync_in line settings for input clock(s) |
| `sync_in_ps` | - | Reserved to select the divider for the input clock |

| | | |
|---|---|---|
| `sync_out` | L | Selects the sync_out line settings |
| `d_out` | L | Selects the 4 digital output line settings |

**Note:**

Be careful not to overwrite settings for pairs of channels or the entire layer.

### *4.48.2 DqAdv608GetCfg*

**Syntax:**

```
int DqAdv608GetCfg(int hd, int devn, uint32 chnl, uint32 *front_cfg, uint32
*mode, pVR608_ExtCfg extcfg)
```

**Input**

| | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int chnl | Channel (0 - 7) |
| uint32 *front_cfg | Set front-end configuration, if required or NULL if not |
| uint32 *mode | Set mode of operation, if required or NULL if not |
| pVR608_ExtCfg extcfg | Extended configuration, if required or NULL if not |

**Output**

**Returns**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by devn does not exist or is not a CT-608 |
| `DQ_BAD_PARAMETER` | configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description**

This function returns a previously set configuration for the VR-608 channel.

### *4.48.3 DqAdv608Enable*

**Syntax:**

```
int DqAdv608Enable(int hd, int devn, int enable_mask)
```

**Input**

| | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int enable_mask | Bit mask of channels to enable |

**Output**

**Returns**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by devn does not exist or is not a CT-608 |
| `DQ_BAD_PARAMETER` | configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description**

This function starts or stops a VR-608 device.

Through the `enable_mask` parameter you can enable all 8 channels when `enable_mask = 0xFF`; or you can enable/disable specific channels by setting/clearing it with (`1<<ch|1<<anotherch|...`). This programs the hardware with the configuration sent with DqAdv608SetCfg() and begins operation. Calling DqAdv608Enable() again with enable_mask = 0 will disable all channels.

### *4.48.4 DqAdv608Read*

**Syntax:**

```
int DqAdv608Read(int hd, int devn, int CLSize, uint32* CL, uint32*
bin_data, uint32** CL_ptr, pVR608_READ_DATA data)
```

**Inputs:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number |
| `int CLSize` | Size of the following channel list |
| `uint32 *CL` | Channel list to read from 0 – 7 |
| `uint32 *bin_data` | Binary data received, if required or NULL if not |
| `uint32** CL_ptr` | Extended configuration, if required or NULL if not |
| `pVR608_READ_DATA data` | Data structure which holds the values from the VR-608. |

**Outputs:**

none

**Returns:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by devn does not exist or is not a CT-608 |
| `DQ_BAD_PARAMETER` | configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function returns the current values from the channel in the channel list. The data is returned in the array `bin_data` in its raw form and in the `pVR608_READ_DATA` structure converted to velocity, position, and tooth count depending on the mode set in DqAdv608SetCfg().

```
typedef struct {
    double velocity;
    uint32 position;
    uint32 tcount;      // teeth count
    uint32 tstamp;      // default is DQ_LN_10us_TIMESTAMP which means each
                        // increment is 10us
    uint32 rd_flags;    // see DQ_VR608_RD_FLAG constant
    uint32 status;      // channel status register (see DQ_VR608_ADC_STS.status)
    double adc_val;     // last calibrated ADC value (see DQ_VR608_ADC_STS.last)
} DQ_VR608_READ_DATA, *pDQ_VR608_READ_DATA;
```

Some of the structure members will not have data in every mode. When there is no data they will be set to 0; this is summarized in the table below. To read the timestamp a channel in `CL[]` must be bitwise ORed with `DQ_VR608_CHNLRQ_TSTAMP` when it is passed in.

This table gives `VR608_READ_DATA` struct members `velocity`, `position`, `tcount` depending on the mode that the user initially set the channel(s) into by DqAdv608SetCfg:

| **Mode** | | **Velocity** | **Position** | **Tcount** |
|---|---|---|---|---|
| Quadrature | Decoder | No | Yes | Yes |
| Timed | Counter | Yes | Yes | Yes |
| Z-pulse | Counter | Yes | Yes | Yes |
| N-pulse | Counter | Yes | No | No |

Above, when a channel's `VR608_ExtCfg` struct member `ps_param` is set to be >1, the value of the `VR608_READ_DATA` struct member `velocity` will be in RPM calculated based on the `ps_param` value. A value of 1 will there for give teeth per minute. A value of 0 will give teeth per second.

The `VR608_READ_DATA` struct member `status`, `adc_val`, and `rd_flags`, are returned when the channel in `CL[]` is bitwise ORed with `DQ_VR608_CHNLRQ_STATUS`, `DQ_VR608_CHNLRQ_ADC_RD`, or both.

The `status` returns the hardware register for edge detection using `DQ_VR608_AIN_STS_` bit flags. The `adc_val` returns the hardware's ADC voltage value at the time of read; useful for debug purposes. Without both flags `rd_flags` returns `DQ_VR608_RD_FLAG_CKT_NODATA`; with them it returns the open/closed circuit detection bit flag value of `DQ_VR608_RD_FLAG_CKT_OPEN` or `CKT_CLOSED`.

**Notes**:

In N-tooth mode, output data does not change unless a requisite number of pulses occurs. In your application, if the system stops rotating, the velocity data stored in the FIFO will stop updating until the requisite number of pulses occurs, which can result in subsequent reads of a stopped system continually reading a stale velocity value.

A way to verify you are reading fresh data is to check the `DQ_VR608_ADC_STS.status` register that is returned with the `DqAdv608Read` API or using the `DqAdv608ReadADCStatus` API.

`DQ_VR608_ADC_STS.status` provides flags that indicate whether an edge was detected on the ADC circuitry. **However note** that reading this register provides status data for a channel pair (two channels with each read, the even and the odd channels share a register). The status data is sticky, and after you read the status, the status bits are cleared. Therefore, if you read the even channel and then the odd, the second reading of the odd channel will result in reading cleared status states (you'll lose your data). You should read (and use) the status register of only the even or odd channel to avoid this.

By ORing `DQ_VR608_ADC_STS.status` bit states for the even channel `(DQ_VR608_AIN_STS_ZLHA | DQ_VR608_AIN_STS_ZHLA)` and bit states for the odd channel `(DQ_VR608_AIN_STS_ZLHB | DQ_VR608_AIN_STS_ZHLB)` you can verify whether H/L or L/H transitions have occurred on the channels.

If the status indicates no edge is detected, then the user can ignore the stale data read using the ADC read, and report 0 velocity (nothing is moving).
`DQ_VR608_ADC_STS.status` register bits are mapped as follows:

- `DQ_VR608_AIN_STS_ZLHB (1L<<20) //Rising edge zero crossing was detected on "B" (odd channel)`
- `DQ_VR608_AIN_STS_ZHLB (1L<<19) //Falling edge zero crossing was detected on "B" (odd channel)`
- `DQ_VR608_AIN_STS_ZLHA (1L<<18) //Rising edge zero crossing was detected on "A" (even channel)`
- `DQ_VR608_AIN_STS_ZHLA (1L<<17) //Falling edge zero crossing was detected on "A" (even channel)`
- `DQ_VR608_AIN_STS_ABD (1L<<16) // - "A" and "B" data is ready`

For reliable ZC detection, UEI recommends that you set the ADC ZC detection threshold slightly above 0v to avoid false positives. This is setup in the `DqAdv608SetCfg` API and programmed in the extended config structure (`excfg.cfg_flags` has `DQ_VR608_CFGVLD_ZC_LEVEL` ORed in; and `excfg.zc_level` can be 0.1f).
Available in DAQLib revision 4.10.0.14+

### 4.48.5 *DqAdv608ReadADCStatus*

**Syntax:**

```
int DqAdv608ReadADCStatus(int hd, int devn, int chnl, pVR608_ADC_STS pSts)
```

**Input**

| | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int chnl | Size of the following channel list |
| pVR608_ADC_STS pSts | Data structure which holds the values from the VR-608. |

**Output**

**Returns**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by devn does not exist or is not a CT-608 |
| `DQ_BAD_PARAMETER` | configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description**

This function returns the status of the ADC for the channel pair that contains the channel specified in chnl. For example setting chnl = 2 or chnl = 3 will return the status for both channels 2 and 3. This is returned in the structure pSts. The values for the fields of pSts are listed below.

```
typedef struct {
    uint32 status;
    uint32 last_0;   // last calibrated ADC value for the even channel
    uint32 last_1;   // last calibrated ADC value for the odd channel
    uint32 max_0;    // maximum level registered for the even channel
    uint32 max_1;    // maximum level registered for the odd channel
    uint32 min_0;    // minimum level registered for the even channel
    uint32 min_1;    // minimum level registered for the odd channel

} VR608_ADC_STS, *pVR608_ADC_STS;
```

### *4.48.6 DqAdv608ReadADCFifo*

**Syntax:**

```
int DqAdv608ReadADCFifo(int hd, int devn, int channel, int sizemax, int
*size, int* avail, float* data_a, float* data_b, uint32* binary)
```

**Input**

| | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int `channel` | Size of the following channel list |
| int sizemax | Maximum size to read |
| int *size | Size of data in data_a and data_b |
| int* avail | Amout of data still available in the fifo (in uint32's) |
| float* data_a | Data from the even channel (e.g. 0,2,4,6) or NULL if not required |
| float* data_b | Data from the odd channel (e.g. 1,3,5,7) or NULL if not required |
| uint32* binary | Raw data or NULL if not required. The even channel is in the upper 2 bytes; the odd channel is in the lower 2 bytes. |

**Output**

**Returns**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by devn does not exist or is not a CT-608 |
| `DQ_BAD_PARAMETER` | configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description**

This function reads data stored in the FIFO for an ADC channel. This allows the user to "peek" into the scan of data on a selected channel. Data is taken from both channel pairs simultaneously. For example if channel is set to 2 data from channel 2 will be in data_a and data from channel 3 will be in data_b. If it is channel is set to 3 the same data will be received.

The first call will enable the ADC writes to the FIFO. To disable ADC write to the FIFO call with sizemax = 0.

**Note:**

The data returned is not calibrated (<1% error in hardware) and there is not guarantee that it is gapless.

### *4.48.7 DqAdv608ReadFifo*

**Syntax:**

`int DqAdv608ReadFifo(int hd, int devn, uint32 channel, uint32 sizemax, uint32* size, uint32* data)`

**Input**

| | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int `channel` | Size of the following channel list |
| int sizemax | Maximum size to read |
| int *size | Size of the data returned |
| uint32* data | Pointer to the array where to store the data |

**Output**

**Returns**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by devn does not exist or is not a CT-608 |
| `DQ_BAD_PARAMETER` | configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description**

This function reads data stored in the FIFO for the channel. FIFO mode has to be selected in DqAdv608SetConfig() and enabled with DqAdv608Enable().

The following information can be stored in the FIFO:

1. Current teeth count (+ optional timestamp)

- or -

2. Current position (+ optional timestamp) - requires Z-tooth
3. Current velocity (+ optional timestamp)
4. Timestamp and CRL/CRH - N-periods for each tooth or N teeth

The table below summarizes what events will cause data to be stored in the FIFO for each data point. The data will consist of various counter/timer registers and (optionally) the timestamp. Each of these will take up one uint32 and will return all of them in each call. The parameters can be used to get the velocity, position and number of teeth depending on the event configured.

| **Event** | **Data returned** |
|---|---|
| On timebase selected (Timed): | CRH,{CRR[15:0],CRL[15:0]},CR[,TS] |
| N-teeth encountered: | CRR,CRL[,TS] |
| Z-tooth found | CRH,{CRR[15:0],CRL[15:0]},CR[,TS] |

The meaning of the data registers above is as follows:

CRH – time

CRL – number of teeth over time in CRH

CRR – position count (reset by z-tooth detection
CR – total tooth count
TS – Timestamp

### 4.48.8 DqAdv608SetWatermark

**Syntax:**
`int DqAdv608SetWatermark(int hd, int devn, uint32 chnl, uint32 len)`

**Input**

| | |
|---|---|
| int hd | Handle to the IOM received from `DqOpenIOM()` |
| int devn | Device number |
| int chnl | Size of the following channel list |
| uint32 len | Maximum size to read (0-256) |

**Output**

**Returns**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by devn does not exist or is not a CT-608 |
| `DQ_BAD_PARAMETER` | configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description**
This function allows the user to configure the buffer watermark to better control dataflow from the IOM. The parameter `len` is the FIFO half full interrupt level for the FIFO buffer.

## 4.49 DNA-IRIG-650 layer

### 4.49.1 DqAdv650ConfigTimekeeper

**Syntax:**

`int DAQLIB DqAdv650ConfigTimekeeper(int hd, int devn, uint32 mode, uint32 flags)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 mode` | Mode of operation |
| `uint32 flags` | Operation flags |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets up timekeeper configuration

<mode> of TK operation
CT650_TKPPS_IO(N) -- External 1PPS signal from I/O lines
CT650_TKPPS_RFIN -- External 1PPS signal from RFIn1
CT650_TKPPS_SYNC(N) -- One of the SyncX lines
CT650_TKPPS_1PPX -- External 1PPx
CT650_TKPPS_GPS -- GPS as a source of 1PPS
CT650_TKPPS_INTERNAL -- Rely on internal synchronization
CT650_TKPPS_TIMECODE -- Work with PPS derived from external time code (AM or MII or NRZ)
CT650_TKPPS_DISABLED -- Timekeeper disabled (stops running)

Additional configuration <flags>
CT650_TKFLG_AUTOFOLLOW -- if external PPS is missing use internal timebase to continue
CT650_TKFLG_USENOMINAL -- if external PPS is lost use internal timebase instead of measured one from the time when it was available
CT650_TKFLG_SUBPPS -- external timebase can be slower than 1PPS (1PPM/1PPH) (CT650_TKFLG_USENOMINAL should be used in conjunction)

**Note:**

None.

### 4.49.2 DqAdv650SetTimecodeInput

**Syntax:**

```
int DAQLIB DqAdv650SetTimecodeInput(int hd, int devn, uint32 mode, uint32
flags,
pCT650_IRIG_PRM_DEF pPrmDef, pCT650_IRIG_CODE_DEF pDataDef)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 mode` | Mode of operation |
| `uint32 flags` | Operation flags |
| `pCT650_IRIG_PRM_DEF pPrmDef` | Timecode parameter definition table |
| `pCT650_IRIG_CODE_DEF pDataDef` | Timecode data definition table |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets up time code input configuration

Input <mode>
CT650_IN_ENABLED -- enable input
CT650_IN_DISABLED -- disable input
CT650_IN_EXT10MHZ -- use external 10MHz connected to RFIn1
CT650_IN_P0_ONLY -- enable single Px character at the beginning
CT650_IN_IDLE_EN -- enable "idle" characters within time message
CT650_IN_DICON_TDTK -- disconnect TD from TK - disable "time ready" output strobe to TK

<input> definition: where time code is coming from
CT650_IN_AM -- Time code is on AM signal
CT650_IN_M2_RF0 -- Manchester II code on RFIn0 input
CT650_IN_M2_RF1 -- Manchester II code on RFIn1 input

CT650_IN_M2_IO0 -- Manchester II code on I/O 0 input
CT650_IN_M2_IO1 -- Manchester II code on I/O 1 input
CT650_IN_NRZ_RF0 -- NRZ code on RF0 input
CT650_IN_NRZ_RF1 -- NRZ code on RF1 input
CT650_IN_NRZ_IO0 -- NRZ code on I/O 0 input
CT650_IN_NRZ_IO1 -- NRZ code on I/O 1 input
CT650_IN_GPS -- Time code is taken from GPS NMEA message

<`pPrmDef`> and <`pDataDef`> define input timecode. Different versions of IRIG input timecodes are provided in included file DAQLibHLTimeCodes.h. If the timecode tables needs to be modified it is a good practice to copy them into the user allocated memory, perform modifications and then pass the pointers to these new locations.

**Notes:**

Signal and data definition tables are pre-packaged for most types on input signals. Use compatible pSigDef and pDataDef to define input time code

### *4.49.3 DqAdv650SetTimecodeInputEx*

**Syntax:**

```
int DAQLIB DqAdv650SetTimecodeInputEx(int hd, int devn, uint32 mode, uint32 
input, pCT650_IRIG_PRM_DEF pPrmDef, pCT650_IRIG_CODE_DEF pDataDef, 
pCT650_IRIG_SIG_DEF pSigDef, uint32 sigDefMask)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 mode` | Mode of operation |
| `uint32 flags` | Operation flags |
| `pCT650_IRIG_PRM_DEF pPrmDef` | Timecode parameter definition table |
| `pCT650_IRIG_CODE_DEF pDataDef` | Timecode data definition table |
| `pCT650_IRIG_SIG_DEF pSigDef` | Direct values for various registers to define input timecode signal |
| `uint32 sigDefMask` | bit specifies which parameters in pSigDef are valid |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |

| | |
|---|---|
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets up time code input configuration. This is an extended version of `DqAdv650SetTimecodeInput()`which allows direct assignment of various time code parameters.

Input <mode>
CT650_IN_ENABLED -- enable input
CT650_IN_DISABLED -- disable input
CT650_IN_EXT10MHZ -- use external 10MHz connected to RFIn1

Timecode input configuration
<input> definition: where time code is coming from
CT650_IN_AM -- Time code is on AM signal
CT650_IN_M2_RF0 -- Manchester II code on RFIn0 input
CT650_IN_M2_RF1 -- Manchester II code on RFIn1 input
CT650_IN_M2_IO0 -- Manchester II code on I/O 0 input
CT650_IN_M2_IO1 -- Manchester II code on I/O 1 input
CT650_IN_NRZ_RF0 -- NRZ code on RF0 input
CT650_IN_NRZ_RF1 -- NRZ code on RF1 input
CT650_IN_NRZ_IO0 -- NRZ code on I/O 0 input
CT650_IN_NRZ_IO1 -- NRZ code on I/O 1 input
CT650_IN_GPS -- Time code is taken from GPS NMEA message

<`pPrmDef`> and <`pDataDef`> define input timecode. Different versions of IRIG input timecodes are provided in included file DAQLibHLTimeCodes.h. If the timecode tables needs to be modified it is a good practice to copy them into the user allocated memory, perform modifications and then pass the pointers to these new locations.

Additional parameters can be supplied in <`pSigDef`> parameter. It should point to the following table:

```
typedef struct {
    // user-definied part
    double ppx;           // time between time codes (for most IRIG-x == 1pps)
    uint32 pph;           // pulses per hour, 1PPS=3600PPH
    uint32 in_freq;       // input carier frequency
    uint32 rx_ici;        // ct650_rx_ici - pulses per bit
    uint32 dbcnt;         // ct650_dbcnt (debouncing counter only in NRZ input mode in 10ns)
    uint32 rxmsg_len;     // ct650_rxmsg_len input message length
    uint32 pi_min;        // ct650_pi_min - minimum number of pulses in position identifier
    uint32 pi_max;        // ct650_pi_max - maximum number of pulses in position identifier
    uint32 l0_min;        // ct650_l0_min - minimum number of pulses in level 0 position
    uint32 l0_max;        // ct650_l0_max - maximum number of pulses in level 0 position
    uint32 l1_min;        // ct650_l1_min - minimum number of pulses in level 1 position
    uint32 l1_max;        // ct650_l1_max - maximum number of pulses in level 1 position
    // system-defined part
    uint32 cr_tol_min;    // ct650_cr_tol_min (signal tolerance - MAN2 mode only)
    uint32 ct_tol_max;    // ct650_ct_tol_max (signal tolerance - MAN2 mode only)
    uint32 mii_tol;       // ct650_mii_tol 50us @ 100Mhz base
    uint32 pps_min;       // ct650_pps_min - flywheel clock divider auto-correction low
limit
    uint32 pps_max;       // ct650_pps_max - flywheel clock divider auto-correction high
limit
    uint32 ppsmav;        // ct650_ppsmav 3 = 8 points
} CT650_IRIG_SIG_DEF, *pCT650_IRIG_SIG_DEF;
```

The upper part can be user-defined. When function is processed parameters are taken from the passed <`pPrmDef`> and <`pDataDef`> first and then some timecode parameters can be overwritten by the parameters passed in <`pSigDef`>. <`sigDefMask`> has to be used to tell firmware which parameter needs to be overwritten:

```
// constant flags to validate each member of CT650_IRIG_SIG_DEF
#define CT650_RX_ICIWR      (1L<<0)
#define CT650_DBCNTWR       (1L<<1)
#define CT650_RXMSG_LENWR   (1L<<2)
#define CT650_PI_MINWR      (1L<<3)
#define CT650_PI_MAXWR      (1L<<4)
#define CT650_L0_MINWR      (1L<<5)
#define CT650_L0_MAXWR      (1L<<6)
#define CT650_L1_MINWR      (1L<<7)
#define CT650_L1_MAXWR      (1L<<8)
#define CT650_CR_TOL_MINWR  (1L<<9)
#define CT650_CR_TOL_MAXWR  (1L<<10)
#define CT650_MII_TOLWR     (1L<<11)
#define CT650_PPS_MINWR     (1L<<12)
#define CT650_PPS_MAXWR     (1L<<13)
#define CT650_PPSMAVWR      (1L<<14)
```

Notes:

Signal and data definition tables are pre-packaged for most types on input signals.
Use compatible pSigDef and pDataDef to define input time code

### 4.49.4 DqAdv650SetTimecodeOutput

**Syntax:**

```
int DAQLIB DqAdv650SetTimecodeOutput(int hd, int devn, uint32 mode, uint32
output, pCT650_IRIG_PRM_DEF pPrmDef, pCT650_IRIG_CODE_DEF pDataDef)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 mode` | Mode of operation |
| `uint32 output` | Output configuration |
| `pCT650_IRIG_PRM_DEF pPrmDef` | Timecode parameter definition table |
| `pCT650_IRIG_CODE_DEF pDataDef` | Timecode data definition table |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets up time code input configuration

Output <mode>
CT650_OUT_ENABLED -- enable output
CT650_OUT_DISABLED -- disable output

Notes:

Signal and data definition tables are pre-packaged for most types on input signals.
Use compatible pSigDef and pDataDef to define input time code

### *4.49.5 DqAdv650AssignTTLOutputs*

**Syntax:**

```
int DAQLIB DqAdv650AssignTTLOutputs(int hd, int devn, uint32 mode, uint32
output, uint32 ttl0, uint32 ttl1, uint32 ttl2, uint32 ttl3)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 mode` | Mode of operation |
| `uint32 output` | Output configuration |
| `uint32 ttl0` | Assignment for TTL output line 0 |
| `uint32 ttl1` | Assignment for TTL output line 1 |
| `uint32 ttl2` | Assignment for TTL output line 2 |
| `uint32 ttl3` | Assignment for TTL output line 3 |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

IRIG-650 layer has four independent TTL output lines <ttl0..ttl3> each of the lines can carry one of the following signals:

```
#define CT650_OUT_CFG_AM_NRZ      31    // AM->NRZ output
#define CT650_OUT_CFG_GPS_FIXV    30    // GPS Fix Valid output
#define CT650_OUT_CFG_GPS_ANSH    29    // GPS Antenna Shorted output
#define CT650_OUT_CFG_GPS_ANOK    28    // GPS Antenna Ok output
#define CT650_OUT_CFG_GPS_TXD1    27    // GPS TXD1 (COM1) output
#define CT650_OUT_CFG_GPS_TXD0    26    // GPS TXD0 (COM0) output
#define CT650_OUT_CFG_MII_CR      25    // Recovered Manchester II carrier
#define CT650_OUT_CFG_MII_NRZ     24    // Decoded Manchester II -> NRZ sequence
#define CT650_OUT_CFG_EVENT3      23    // Event 3
#define CT650_OUT_CFG_EVENT2      22    // Event 2
#define CT650_OUT_CFG_EVENT1      21    // Event 1
#define CT650_OUT_CFG_EVENT0      20    // Event 0
#define CT650_OUT_CFG_SRC_SYNC3   19    // Drive output from sync[3]
#define CT650_OUT_CFG_SRC_SYNC2   18    // Drive output from sync[2]
```

```
#define CT650_OUT_CFG_SRC_SYNC1   17    // Drive output from sync[1]
#define CT650_OUT_CFG_SRC_SYNC0   16    // Drive output from sync[0]
#define CT650_OUT_CFG_CR_OUT      15    // Output carrier frequency
#define CT650_OUT_CFG_SRC_CR      14    // Custom frequency output (from PLL)
#define CT650_OUT_CFG_SRC_10MHZ   13    // Precision 10MHz
#define CT650_OUT_CFG_SRC_5MHZ    12    // Precision 5MHz
#define CT650_OUT_CFG_SRC_1MHZ    11    // Precision 1MHz
#define CT650_OUT_CFG_SRC_NRZS    10    // Output NRZ start strobe
#define CT650_OUT_CFG_SRC_MII      9    // Manchester II output time code
#define CT650_OUT_CFG_SRC_NRZ      8    // NRZ output time code
#define CT650_OUT_CFG_SRC_1GPS     7    // Re-route GPS 1PPS pulse
#define CT650_OUT_CFG_SRC_1PPH     6    // 1PPH pulse
#define CT650_OUT_CFG_SRC_1PPM     5    // 1PPM pulse
#define CT650_OUT_CFG_SRC_1PPS     4    // 1PPS pulse
#define CT650_OUT_CFG_SRC_0_1S     3    // 0.1sec pulse
#define CT650_OUT_CFG_SRC_0_01S    2    // 0.01sec pulse
#define CT650_OUT_CFG_SRC_1        1    // Drive output with 1
#define CT650_OUT_CFG_SRC_0        0    // Drive output with 0
```

<mode> allows to select the following options:

1. Enable/disable output. Each output has two line drivers. At least one driver has to be enabled for output to operate. Second driver might be nessesary for longer line to compensate for its impedance

```
#define CT650_OUT_TTLEN1    (1L<<21)        // =1 - Enable buffer 1
#define CT650_OUT_TTLEN0    (1L<<20)        // =1 - Enable buffer 0
```

2. Pulse length. By default TTL puls is 10us, however at frequencies >10kHz it is nessesary to use shorter pulses. Following constants allow to switch to shorter pulses:

```
#define CT650_OUT_TTL3_40nS     (1L<<31)    // If any of these bits is set - corresponding
TTL output pulse
#define CT650_OUT_TTL2_40nS     (1L<<30)    // will NOT be extended to at least 60uS, any
pulse longer, then 60uS
#define CT650_OUT_TTL1_40nS     (1L<<29)    // will not be extended
#define CT650_OUT_TTL0_40nS     (1L<<28)    //
#define CT650_OUT_TTL_40nS      (0xf<<28)   // all pulses are 60 us length
```

3. Control line termination:

```
#define CT650_OUT_RF1TERM   (1L<<24)        // =1 - Enable 50 Ohm termination for the TTL
RF1 (TRIGIN) input
#define CT650_OUT_RF0TERM   (1L<<23)        // =1 - Enable 50 Ohm termination for the TTL
RF0 (CLKIN) input
#define CT650_OUT_AMTERM    (1L<<22)        // =1 - Enable 50 Ohm termination for the AM
input
```

4. Use debug output mode (used to debug non-standard time codes):

```
#define CT650_OUT_DBG       (1L<<25)        // switch TTL outputs to debug mode
```

If this bit is set output is re-assigned to the debug output mode and <ttlx> value has a different meaning:

```
// TTL output assingments for debugging
#define CT650_OUT_CFG_TD_PPX      30    // Detected 1PPx from the time decoder
#define CT650_OUT_CFG_TD_RDY      29    // Time decoder validation complete
```

```
#define CT650_OUT_CFG_TD_1PPSD    28    // TK: internal delayed 1PPS strobe (i.e. 1pps applied delay)
                                        // DELAY = TKPERIOD - TKDLY - (w_tkcfg_pdc ? in_dly : 0)
#define CT650_OUT_CFG_TA_CR       27    // Carrier clock for time assembler

#define CT650_OUT_CFG_FW_RLD      26    // TK: =1 if flywheel counter should be reloaded
(r_flywheel_reload)
#define CT650_OUT_CFG_FW_RST      25    // TK: =1 if flywheel counter (and 1ppS delay) should be
cleared (r_flywheel_reset)
#define CT650_OUT_CFG_FW_DONE     24    // TK: Flywheel counter expired or early 1ppS (r_fw_cnt_done)
#define CT650_OUT_CFG_FW_EXP      23    // TK: =1 if flywheel counter expired and reloaded by itself
#define CT650_OUT_CFG_TD_CR       22    // TD: Carrier pulses (start of each pulse indicate start of
the carrier period)
#define CT650_OUT_CFG_TT_RDY      21    // TTL2TIME: Character ready pulse
#define CT650_OUT_CFG_TA_IDLE     20    // TA: Idle state of the TTL NRZ output state machine
#define CT650_OUT_CFG_TA_RESTART  19    // TA: Restart request by delayed external PPx generated by the
time keeper
```

**Notes:**

<output> parameter can be used to assign SYNCx line to one of the signals described above suing `DQ_CT650_TTL_SYNCX` constant.
<mode> field then selects which of the SYNCx lines are enabled for output:

```
DqAdv650AssignTTLOutputs(hd, devn,
CT650_OUT_SYNCEN0|CT650_OUT_SYNCEN1|CT650_OUT_SYNCEN2|CT650_OUT_SYNCEN3,
                                           DQ_CT650_TTL_SYNCX,
                                           CT650_OUT_CFG_SRC_1PPS,
                                           CT650_OUT_CFG_SRC_1PPS,
                                           CT650_OUT_CFG_SRC_1PPS,
                                           CT650_OUT_CFG_SRC_1PPS);
```

### *4.49.6 DqAdv650SetAMOutputLevels*

**Syntax:**

```
int DAQLIB DqAdv650SetAMOutputLevels(int hd, int devn, uint32 flags, double
hi_gain, double low_gain, double offset)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 flags` | <reserved> |
| `uint32 output` | Output configuration |
| `double hi_gain` | "logic 1" gain [0..2]V |
| `double low_gain` | "logic 0" gain [0..2]V |
| `double offset` | output offset [-0.5..+0.5]V |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |

| | |
|---|---|
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets up AM output levels

### 4.49.7 *DqAdv650SetPropDelay*

**Syntax:**

```
int DAQLIB DqAdv650SetPropDelay(int hd, int devn, uint32 flags, int delay)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 flags` | <reserved> |
| `int delay` | Propagation delay in 10ns intervals |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Function sets propagation delay in 10ns resolution to compensate for input timecode signal delay in wiring. It is used to fine-tune input time to adjust for the beginning of the time frame (PP symbols in most time codes).

Zero disables propagation delay.

### 4.49.8 DqAdv650SetAMZCMode

**Syntax:**

```
int DAQLIB DqAdv650SetAMZCMode(int hd, int devn, uint32 flags, int
zc_adjust)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 flags` | <reserved> |
| `int zc_adjust` | Zero crossing averaging parameters |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

AM zero-crossing parameters

Following <flags> can be or-red:
CT650_ZCCFG_ZAV(N) -- number of samples in average for ADC data
CT650_ZCCFG_ZCAZ -- =1-use "auto zero" if it auto-calculated zero level is within of the 1/8 of "zero_level" (ZCL_Dx)
CT650_ZCCFG_ZCZD -- Select direction of the crossing for min/max detection (1=rising)
CT650_ZCCFG_ZCAS(N) -- 0=1, 1=2, 2=4, 3=8, 4=16, 5=32, 6=64, 7=128, 8=256
CT650_ZCCFG_ZCSC(N) -- 0=1, 1=2, 2=4, 3=8, 4=16, 5=32, 6=64, 7=128, 8=256,

**Note:**

This function is intended for fine-tuning of the input. Zero crossing is used as a marker for the beginning of the time frame. If the signal is noisy a larger value might be required to precisely decode the beginning of the time frame.

### *4.49.9 DqAdv650SetLocalOffset*

**Syntax:**

```
int DAQLIB DqAdv650SetLocalOffset(int hd, int devn, int hour_offset, int
min_offset)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int hour_offset` | Local time zone offset relatively to UTC, hours |
| `int min_offset` | Local time zone offset relatively to UTC, minutes |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Function sets location delay - timezone relative to UTC. Most time zones are defined in +/- hours relatively to UTC. For example Boston (Eastern Standard Time) is -5 hours behind the UTC. If daylight saving time is used the difference is -4 hours.

In a rare case some localities have their local time set to half-hour boundary.

**Note:**

Timekeeper always operates with the timecode is receives which is UTC. When local time offset is set it is applied to convert user-supplied time to UTC to adjust timekeeper and back to the local time when read from the timekeeper. However, if TREGs or timecode are read directly (i.e. in BCD mode) this offset does not apply.

## *4.49.10 DqAdv650GetInputTimecode*

**Syntax:**

```
int DAQLIB DqAdv650GetInputTimecode(int hd, int devn, uint32 size, uint32*
block, uint32* decoded_code)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 size` | How many characters to return (depends on the lenght of time code). Each character is returned in uint32 format |
| `uint32* block` | Which timecode input block was active at this moment |
| `uint32* decoded_code` | Received characters |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Retrieves processed block of decoded time code data.

Each decoded time code character occupies 32-bit words with the following structure:

```
CT650_CHAR_TSTAMP(N)   ((N)>>2) // 10us timstamp
CT650_CHAR_IDLE         0       // Idle characters
CT650_CHAR_POS          1       // 01 - Position Identifier
CT650_CHAR_ZERO         2       // 10 - "0"
CT650_CHAR_ONE          3       // 11 - "1"
```

**Notes:**

First two characters are dedicated for the timestamp and phase delay of the input signal (in 10ns resolution).

Time decoder processes incoming timecode serial data stream and looking for the single or dual position identifier as a start of the message. After message start is detected – each character is copied into the message memory until the number of characters reaches the number programmed.

First two locations of the message memory are dedicated for the timestamp and phase delay of the input signal.

There are two memory blocks to store incoming timecode characters: one is actively storing incoming characters and the second one holding recently decoded time frame. Then they switch.

### *4.49.11 DqAdv650GetTimeRegisters*

**Syntax:**

```
int DAQLIB DqAdv650GetTimeRegisters(int hd, int devn, uint32* time_regs)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32* time_regs` | Pointer to an array to store content of time registers (of 16*sizeof(uint32) size) |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Function retrieves timekeeper time register (sixteen registers)

**Notes:**

Use the following macros to extract information from delivered data. Pass <time_regs> as a parameter for these macros
CT650_GET_BCD_SEC100(A)
CT650_GET_BCD_SEC10(A)
CT650_GET_BCD_SEC(A)
CT650_GET_BCD_MIN(A)
CT650_GET_BCD_HOUR(A)
CT650_GET_BCD_DAYS(A)

CT650_GET_BCD_YEAR(A)
CT650_GET_SBS_SEC(A)
CT650_GET_PHASE_DLY(A)
CT650_GET_TIMESTAMP(A)

### *4.49.12 DqAdv650ProgramPLL*

**Syntax:**

```
int DAQLIB DqAdv650ProgramPLL(int hd, int devn, double duty_cycle, double
frequency, double* f_actual)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `double duty_cycle` | Signal duty cycle from 0 to 1 (non-inclusive) |
| `double frequency` | Required PLL frequency |
| `double* f_actual` | Actual PLL frequency |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Function sets up PLL frequency and duty cycle. Build-in PLL can generate any frequency from 1Hz to 1MHz accurate to four digits. Since PLL might not be able to generate accurate frequency for all requested frequencies the actual generated frequency is calculated based on the internal multiplicators and divisors used and retuned in <f_actual>.

**Notes:**

Use DqAdv650SetAssignTTLOutputs with CT650_OUT_CFG_SRC_CR to route frequence out to the desired TTL output

### *4.49.13 DqAdv650GetTimeSBS, DqCmd650GetTimeSBS*

**Syntax:**

```
int DAQLIB DqAdv650GetTimeSBS(int hd, int devn, uint32* seconds, uint32*
micro, uint32* dayofyear, uint32* year, uint32* tkstatus)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32* seconds` | Number of seconds in Unix format |
| `uint32* micro` | Number of microsecinds within a second |
| `uint32* dayofyear` | Day of the current year |
| `uint32* year` | Current year |
| `uint32* tkstatus` | Current time keeper status (error flags) |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Latches and returns time captured at the moment of latching it in binary format

Timekeeper status is a bitfield which can have following bits:

```
// Timekeeper status bits
CT650_TKSTS_BIN2BCD_ERR   (1L<<13) // sticky, =1 if binary time code is invalid (logic error)
CT650_TKSTS_BCD2BIN_ERR   (1L<<12) // sticky, =1 if BCD time code is invalid (serial stream error)
CT650_TKSTS_TK_ERR        (1L<<11) // sticky, =1 if time code received mismatch current time
CT650_TKSTS_1PPS_GEN      (1L<<10) // sticky, =1 if 1PPS was generated
CT650_TKSTS_1PPS_RCV      (1L<<9)  // sticky, =1 if selected 1PPS was received
CT650_TKSTS_TC_RCV        (1L<<8)  // sticky, =1 if external time code was received and applied
CT650_TKSTS_1PPS_ADVAL    (1L<<3)  // =1 if 1ppS output of adaptive PLL in TK is valid
CT650_TKSTS_PPS_LOST      (1L<<2)  // =1 1ppS was not detected within last validation interval
CT650_TKSTS_AVG_INVAL     (1L<<1)  // =1 if averaged 1ppS does not pass validation criteria
CT650_TKSTS_1PPS_INVAL    (1L<<0)  // =1 if last detected external 1ppS was invalid
```

**Note:**

This function provides correction for the local time zone settings

The sister function DqCmd650GetTimeSBS() provides the same interface as DqAdv650GetTimeSBS(), however its processing is done in the Ethernet handler directly instead of queueing requests in the message queue. Thus it might affect response time for other high-priority requests such as mapping modes or asynchronous events.

### *4.49.14 DqAdv650GetTimeBCD, DqCmd650GetTimeBCD*

**Syntax:**

```
int DAQLIB DqAdv650GetTimeBCD(int hd, int devn, pCT650_BCD_TIME time,
uint32* tkstatus)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `pCT650_BCD_TIME time` | Pointer to the structure to store BCD time |
| `uint32* tkstatus` | Current time keeper status (error flags) |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Function latches and returns time captured at the moment of latching it in BCD format:

```
typedef struct {
     uint32 s100;     // 100's of a second (4 bits)
     uint32 s10;      // 10's of a second (4 bits)
     uint32 sec;      // seconds (7 bits)
     uint32 min;      // minutes (7 bits)
     uint32 hours;    // hours (7 bits)
     uint32 days;     // days
     uint32 year;     // year
} CT650_BCD_TIME, *pCT650_BCD_TIME;
```

**Notes:**

BCD time is read directly from TREGs. Timekeeper decodes received time and stores it into TREGs when timecode signal is available and time decoder is running.
This function does not provide correction for the local time zone settings

The sister function DqCmd650GetTimeBCD() provides the same interface as DqAdv650GetTimeBCD(), however its processing is done in the Ethernet handler directly instead of queueing requests in the message queue. Thus it might affect response time for other high-priority requests such as mapping modes or asynchronous events.

### *4.49.15 DqAdv650ResetTimestampsGetBCD*

**Syntax:**

```
int DAQLIB DqAdv650ResetTimestampsGetBCD(int hd, int devn, int mask,
pCT650_BCD_TIME time, uint32* tkstatus)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int mask` | Device mask for devices which timestamps has to be reset |
| `pCT650_BCD_TIME time` | Pointer to the structure to store BCD time |
| `uint32* tkstatus` | Current time keeper status (error flags) |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function latches and returns time captured at the moment of latching it in BCD format then it resets timestmaps on the layers specified by the mask.

**Notes:**

See `DqAdv650GetTimeBCD()` for time representation structure

### *4.49.16 DqAdv650GetTimeANSI*

**Syntax:**

```
int DAQLIB DqAdv650GetTimeANSI(int hd, int devn, struct tm * time, uint32*
micro, uint32* tkstatus)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `struct tm * time` | Pointer to the structure to store ANSI time |
| `uint32* micro` | Microseconds at the moment of retrieving time |
| `uint32* tkstatus` | Current time keeper status (error flags) |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function latches and returns time captured at the moment of latching it in ANSI/System V format:

```
struct tm {
       int tm_sec;     /* seconds after the minute - [0,59] */
       int tm_min;     /* minutes after the hour - [0,59] */
       int tm_hour;    /* hours since midnight - [0,23] */
       int tm_mday;    /* day of the month - [1,31] */
       int tm_mon;     /* months since January - [0,11] */
       int tm_year;    /* years since 1900 */
       int tm_wday;    /* days since Sunday - [0,6] */
       int tm_yday;    /* days since January 1 - [0,365] */
       int tm_isdst;   /* daylight savings time flag */
      };
```

Note:

This function provides correction for the local time zone settings

### *4.49.17 DqAdv650SetTimeSBS*

**Syntax:**

```
int DAQLIB DqAdv650SetTimeSBS(int hd, int devn, uint32 flags, uint32
seconds, uint32 days, uint32 year, uint32* tkstatus)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 flags` | Flags specify what part of the time information has to be set |
| `uint32 seconds` | Number of seconds in Unix format |
| `uint32 days` | Day-of-the-year (if flag enables it) |
| `uint32 year` | Year (1900-2499 if flag enables it) |
| `uint32* tkstatus` | Current time keeper status (error flags) |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets time in binary format
Following <flags> can be used:
CT650_TIME_TM_SEC - seconds are valid
CT650_TIME_TM_YEAR - year is valid to set up
CT650_TIME_TM_YDAY - day of the year is valid to set up

Returns time keeper status

**Notes:**

This function DOES provide correction for the local time zone settings. Set bit (1<<31) in <seconds> to wait for the next 1PPS instead of applying seconds data immediately. If applied immediately while providing external 1PPS this operation can cause one of 1PPS pulse to be missed

### 4.49.18 DqAdv650SetTimeANSI

**Syntax:**

```
int DAQLIB DqAdv650SetTimeANSI(int hd, int devn, uint32 flags, struct tm*
time, uint32* tkstatus)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 flags` | Flags specify what part of the time information has to be set |
| `struct tm* time` | Time in ANSI format |
| `uint32* tkstatus` | Current time keeper status (error flags) |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets time in ANSI/System V format:

```
struct tm {
       int tm_sec;     /* seconds after the minute - [0,59] */
       int tm_min;     /* minutes after the hour - [0,59] */
       int tm_hour;    /* hours since midnight - [0,23] */
       int tm_mday;    /* day of the month - [1,31] */
       int tm_mon;     /* months since January - [0,11] */
       int tm_year;    /* years since 1900 */
       int tm_wday;    /* days since Sunday - [0,6] */
       int tm_yday;    /* days since January 1 - [0,365] */
       int tm_isdst;   /* daylight savings time flag */
       };
```

<flags> defines what part of the structure to set (0 to se all fields)

```
   #define CT650_TIME_TM_SEC   (1L<<0)    // seconds after the minute - [0,59]
   #define CT650_TIME_TM_MIN   (1L<<1)    // minutes after the hour - [0,59]
   #define CT650_TIME_TM_HOUR  (1L<<2)    //  hours since midnight - [0,23]
   #define CT650_TIME_TM_MDAY  (1L<<3)    // day of the month - [1,31]
   #define CT650_TIME_TM_MON   (1L<<4)    // months since January - [0,11]
```

```
#define CT650_TIME_TM_YEAR  (1L<<5)    // years since 1900
#define CT650_TIME_TM_WDAY  (1L<<6)    // days since Sunday - [0,6]
#define CT650_TIME_TM_YDAY  (1L<<7)    // days since January 1 - [0,365]
#define CT650_TIME_TM_ISDST (1L<<8)    // daylight savings time flag
#define CT650_TIME_TM_ALL   (0x1ff)    // everything
```

Notes:

### 4.49.19 DqAdv650ReadGPS

**Syntax:**

```
int DAQLIB DqAdv650ReadGPS(int hd, int devn, uint32 mode, int rq_size,
char* data, int* ret_size)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 mode` | <reserved> |
| `int rq_size` | Requested size in bytes |
| `char* data` | Pointer to the array of rq_size size to store data |
| `int* ret_size` | Actual number of bytes returned |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads data from GPS serial FIFO. GPS data is store in the FIFO in NMEA format. Since the FIFO size is limited data is constantly pushed out of the FIFO to make sure that it contains the recent data. Thus to ensure data delivery this function needs to be called on a periodic basis to avoid FIFO overruns.

**Note:**

Data format is defined in DNx-IRIG-650 user manual.

### *4.49.20 DqAdv650WriteGPS*

**Syntax:**

```
int DAQLIB DqAdv650WriteGPS(int hd, int devn, uint32 mode, int rq_size,
char* data, int* copied)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 mode` | <reserved> |
| `int rq_size` | Requested size in bytes |
| `char* data` | Pointer to the array of rq_size size with the data |
| `int* ret_size` | Actual number of bytes stored |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

The function writes data into GPS module serial interface.

**Note:**

Data format is defined in DNx-IRIG-650 user manual.

### *4.49.21 DqAdv650GetGPSStatus*

**Syntax:**

```
int DAQLIB DqAdv650GetGPSStatus(int hd, int devn, uint32 mode, uint32*
gps_status, uint32* status, uint32* time, uint32* date)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 mode` | <reserved> |
| `uint32* gps_status` | GPS status |
| `uint32* status` | Synchrnonization status |
| `uint32* time` | <reserved> |
| `uint32* date` | <reserved> |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads status word from GPS and if DqAdv650EnableGPSTracking() is enabled returns <status>, <date> and <time> received from GPS as well

<gps_status> can a combination of the follows bits:

```
CT650_GPS_ACC_GPSANTSHRT (1L<<9)    // =1 if antenna is shorted or passive antenna is used
CT650_GPS_ACC_GPSANTOK   (1L<<8)    // =1 if antenna detected by the GPS
CT650_GPS_ACC_GPSRXD1    (1L<<7)    // Current value of GPS RXD1 pin
CT650_GPS_ACC_GPSTXD1    (1L<<6)    // Current value of GPS TXD1 pin
CT650_GPS_ACC_GPSRXD0    (1L<<5)    // Current value of GPS RXD0 pin
CT650_GPS_ACC_GPSTXD0    (1L<<4)    // Current value of GPS TXD0 pin
CT650_GPS_ACC_GPSFIXV    (1L<<3)    // Fix valid output from GPS. Carries 0.5Hz square wave
                                    // when GPS fixes it's position
```

<status> is a status bits used to synchronize timekeeper with GPS time

```
CT650_GPS_ACC_GPSAPPLIED (1L<<18)   // Time is set to GPS time
CT650_GPS_ACC_SATNUM     (1L<<17)   // number of satellites tracked
CT650_GPS_ACC_ACTIVE     (1L<<16)   // GPS tracking is activated
```

### *4.49.22 DqAdv650ReadEventFifo*

**Syntax:**

```
int DAQLIB DqAdv650ReadEventFifo(int hd, int devn, uint32 mode, int
rq_size, uint32* data, int* ret_size)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 mode` | <reserved> |
| `int rq_size` | Requested number of uint32s |
| `uint32* data` | Pointer to the array of uint32s to store FIFO data (must be of rq_size size) |
| `int* ret_size` | Actual number of words copied into the array |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Function reads data from the event FIFO

### *4.49.23 DqAdv650ConfigEvents*

**Syntax:**

```
int DAQLIB DqAdv650ConfigEvents(int hd, int devn, int channel, int flags,
event650_t event, uint32* param)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int channel` | Event channel |
| `int flags` | Configuration flags |
| `event650_t event` | Event type |
| `uint32* param` | An array of parameters of any. Size depends on the event configured |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function configures asynchronous events to be sent from the layer.
<event_type>

```
typedef enum {
    EV650_CLEAR = 0x1000,     // clear all events
    EV650_EVENT = 0x101,      // event has happened
    EV650_PPS_CLK = 0x102,    // 1PPS pacer clock event
    EV650_TIMERDY = 0x103,    // Time received and processed (based on FPS rate)
    EV650_GPSRX = 0x104,      // GPS data was received (reserved)
    EV650_ERROR = 0x110       // multiple errors in TK or TA

} event650_t;
```

For cases EV650_CLEAR, EV650_EVENT, EV650_PPS_CLK, EV650_TIMERDY and EV650_ERROR parameter size is 1, there is no parameters required for other events

Event packets are sent asynchronously in the following structure:

```
typedef struct {
    uint32 chan;          // channel information
    uint32 evtype;        // type of the event
    uint32 evtmask;       // subype of reported events (mask)
    uint32 status;        // event status register
    uint32 tstamp;        // timestamp of event taken from layer TS generator
    uint32 size;          // size of the following data in bytes
    uint32 avail;         // number of bytes available
    uint32 data[];        // data to follow (byte pointer - ntohx() and convert
properly)
} EV650_ID, *pEV650_ID;
```

Please find more information on using asynchronous events in the DNx-IRIG-650 user manual.

### *4.49.24 DqAdv650GetEventStatus*

**Syntax:**

```
int DAQLIB DqAdv650GetEventStatus(int hd, int devn, int evt_chan, int
flags, pEV650_STS event)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int evt_chan` | Event channel |
| `int flags` | Configuration flags <reserved> |
| `pEV650_STS event` | Event status |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads event status along with event capture registers

```
// Event status
typedef struct {
    uint32 event_sts;          // event status
    uint32 event_tstamp;       // event timestamp
    uint32 event_ad;           // event address/data (debug only)
} EV650_STS, *pEV650_STS;
```

### 4.49.25 DqAdv650Enable

**Syntax:**

```
int DAQLIB DqAdv650Enable(int hd, int devn, int enable)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int enable` | TRUE to enable and FALSE to disable |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function enables/disables configured subsystems

### 4.49.26 DqAdv650EnableGPSTracking

**Syntax:**

```
int DAQLIB DqAdv650EnableGPSTracking(int hd, int devn, int enable, uint32*
status)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int enable` | TRUE to enable and FALSE to disable GPS tracking |
| `uint32* status` | Current GPS status |

**Output:**

None

**Return:**

| | |
|---|---|
| DQ_ILLEGAL_HANDLE | illegal IOM Descriptor or communication wasn't established |
| DQ_BAD_DEVN | device indicated by `devn` does not exist or is not a CT-650 |
| DQ_BAD_PARAMETER | counter number is invalid |
| DQ_SEND_ERROR | unable to send the Command to IOM |
| DQ_TIMEOUT_ERROR | nothing is heard from the IOM for Time out duration |
| DQ_IOM_ERROR | error occurred at the IOM when performing this command |
| DQ_SUCCESS | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function enables storing pattern of GPS messages and extracts day and time from them to set up time keeper with this information upon `DqAdv650SetGPSTime()` call.

### 4.49.27 DqAdv650SetGPSTime

**Syntax:**

```
int DAQLIB DqAdv650SetGPSTime(int hd, int devn, int flags, uint32* status)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Device number |
| `int flags` | <reserved> |
| `uint32* status` | Current GPS status |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets timekeeper time from GPS module time

**Note:**

To set timekeeper time from GPS source do the following

1. Eanble GPS time tracking with DqAdv650EnableGPSTracking(,,TRUE,)
2. Call DqAdv650GetGPSStatus() function until CT650_GPS_ACC_ACTIVE bit is set
3. Call DqAdv650SetGPSTime() to set timekeeper
4. Call DqAdv650EnableGPSTracking(,,FALSE,) to stop tracking

### *4.49.28 DqAdv650ClockCalibration*

**Syntax:**

```
int DAQLIB DqAdv650ClockCalibration(int hd, int devn, uint32 mode, uint32
indata, uint32* outdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Device number |
| `uint32 indata` | Data to write |
| `uint32* outdata` | Data received |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-650 |

| | |
|---|---|
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads/writes/enable/disable calibration counter,

**Note:**

The following sequence must be used for clock calibration using external atomic 10MHz clock connected to TTL input 0:

```
------------------------------------------------------------------------------
             mode                                  indata        outdata
------------------------------------------------------------------------------
-Enable counter DQL_IOCTL650_CAL_CT_ENABLE          -              -
-Disable counter DQL_IOCTL650_CAL_CT_DISABLE        -              -
-Calibration counter
data is ready (DQL_IOCTL650_CAL_DATA_READY)        -           counter status
-Read counter value (DQL_IOCTL650_CAL_CT_RD)        -           current counter value
-Write value to counter (DQL_IOCTL650_CAL_CT_WR)  initial
                                                 counter value
-Read state of TTL inputs
(DQL_IOCTL650_CAL_TTL_IN_RD)                        -            TTL inputs state
------------------------------------------------------------------------------
```

## 4.50 DNA-CT-651 layer

### 4.50.1 DqAdv651GetRegister

**Syntax:**

```
int DqAdv651GetRegister(int hd, int devn, uint32 reg, uint32* value)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 reg` | Register to get |

**Output:**

| | |
|---|---|
| `uint32 *value` | Value of register |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-651 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads (gets) a configuration or status register from a CT-651 layer.

The input `reg` must be one of the following defined constants:

| | |
|---|---|
| DQL_IOCTL651_GET_STS | // General Status Register |
| DQL_IOCTL651_GET_FWCFG | // Output mode config register |
| DQL_IOCTL651_GET_FWDC | // Flywheel duty cycle register |
| DQL_IOCTL651_GET_FWDIV | // Output period Register |
| DQL_IOCTL651_GET_FWCLK_MIN | // Flywheel clock divider auto-correction low limit, 32 bits |
| DQL_IOCTL651_GET_FWCLK_MAX | // Flywheel clock divider auto-correction high limit, 32 bits |
| DQL_IOCTL651_GET_FWCRH | // Input clock "high" count in 100MHz or 160MHz clocks |
| DQL_IOCTL651_GET_FWCRP | // Input clock period count in 100MHz or 160MHz clocks |
| DQL_IOCTL651_GET_FWCNT | // Current value of the flywheel counter, debug only |

**Note:**

See powerdna.h file for details about bit definitions of register contents.

### *4.50.2 DqAdv651SetRegister*

**Syntax:**

```
int DqAdv651SetRegister(int hd, int devn, uint32 reg, uint32* value)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 reg` | Register to set |
| `uint32 value` | Value to put in register |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a CT-651 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function writes (sets) a register in a CT-651 layer.

The input `reg` must be one of the following defined constants:

| | |
|---|---|
| DQL_IOCTL651_SET_LCR | // set LCR special bits, 5 bits |
| DQL_IOCTL651_SET_FWCFG | // Output mode config register, 20 bits |
| DQL_IOCTL651_SET_DACW | // DAC write register, 22 bits |
| DQL_IOCTL651_SET_FWDC | // Flywheel duty cycle register, 32 bits |
| DQL_IOCTL651_SET_FWDIV | // Output period Register, 32 bits |
| DQL_IOCTL651_SET_FWCLK_MIN | // Flywheel clock divider auto-correction low limit, 32 bits |
| DQL_IOCTL651_SET_FWCLK_MAX | // Flywheel clock divider auto-correction high limit, 32bits |

**Note:**

See powerdna.h file for details about bit definitions of register contents.

## *4.51 DNA-ARINC-664 layer*

Commands should be used in this order; please see extensive examples for proper call order.

Configuration calls:

- a. DqAdv664ClearConfig
- b. DqAdv664GetDeviceInfo
- c. DqAdv664SetConfig (or DqAdv664BusControl, DqAdv664ValidateVlPortCfg, DqAdv664AddVL, DqAdv664AddPort)
- d. If using VMap, see examples for correct DqAddIOMPort, DqRtVmapInitEx, DqRtVmapAddChannel, DqRtSetDefaultFlags, DqRtVmapStart call sequence.
- e. If using Asynchronous Events: DqCmdSetReplyMaxSize, DqAddIOMPort, DqAdv664ConfigEvents before DqCmdReceiveEvent.
- f. If using high-performance transmission: DqAdv664SendScheduleTable
- g. DqAdv664Enable

Run-time calls:

- h. Immediate-mode: DqAdv664SendMessage and/or DqAdv664RecvMessage
- i. VMap-mode:
    - a. DqRtVmapInitOutputPacket
    - b. DqRtVmapWriteOutput and/or DqRtVmapRequestInput
    - c. DqRtVmapRefresh; or DqRtVmapRefreshOutputs & DqRtXmapRefreshInputs
    - d. DqRtVmapReadInput
- j. Asynchronous Events may be received with DqCmdReceiveEvent.
- k. On-the-fly re-configuration, status, and statistics:
    - a. DqAdv664EnableVLPort
    - b. DqAdv664VLPortStatus, DqAdv664PortMsgStatus, DqAdv664GetBusStat

Cleanup calls (can also be called on startup if exiting uncleanly):

- l. DqAdv664ClearConfig, DqAdv664Enable
- m. If using AsyncEvents, DqAdv664ConfigEvents, DqRtAsyncEnableEvents calls.
- n. If using VMap, see examples for correct DqRtVmapStop/ DqRtVmapClose call sequence.

### *4.51.1 DqAdv664AddPort*

**Syntax:**

```
int DqAdv664AddPort(int hd, int devn, int vl_handle,
pAR664_PORT_CFG pPort, int* port_handle)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int vl_handle` | Handle to the VL (to add ports, control, etc) |
| `pAR664_PORT_CFG pPort` | Port Configuration Parameters |

**Output:**

| | |
|---|---|
| `int *port_handle` | Handle to this port for use with the firmware (must be >0); |

handle is zero if there is no more room in the port table.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not DNA-664 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Adds port to the channel configuration

**Note:**

Call DqAdv664ValidateVlPortCfg before calling this function.

### *4.51.2 DqAdv664AddVL*

**Syntax:**

```
int DqAdv664AddVL(int hd, int devn, int chan, int direction,
pAR664_VL_CFG pVL, int* vl_handle)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chan` | Channel of ARINC-664 board (A, B or dual) |
| `int direction` | TRUE for TX |
| `pAR664_VL_CFG pVL` | virtual link configuration |

**Output:**

| | |
|---|---|
| `int* vl_handle` | Handle to this VL for use with the firmware (must be >0); handle is invalid if zero, (-1) is this VL already exists on the bus |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not DNA-664 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Adds VL to the channel configuration

**Note:**

Call DqAdv664ValidateVlPortCfg before calling this function.

### 4.51.3 DqAdv664BusControl

**Syntax:**

`int DqAdv664BusControl(int hd, int devn, pAR664_CTRL pCtrl)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `pAR664_CTRL pCtrl` | Bus CTRL parameters |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not DNA-664 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Controls bus parameters for ARINC bus.

**Note:**

None

### *4.51.4 DqAdv664ClearConfig*

**Syntax:**

```
int DqAdv664ClearConfig(int hd, int devn, int actions)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int actions` | Set to `AR664_VL_DUAL`. |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not DNA-664 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Clears ARINC-664 configuration including all stored tables for `AR664_VL_DUAL` channels.

**Note:**

None

### *4.51.5 DqAdv664ConfigEvents*

**Syntax:**

`int DqAdv664ConfigEvents(int hd, int devn, int flags, event664_t event, uint32 eparam, int list_sz, uint* h_list);` **Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int flags` | Reserved. Set to 0. |
| `event664_t event` | Event type as EV664_ enum. |
| `uint32 eparam` | Event parameters (see enum). |
| `int list_sz` | Number of elements in the h_list array. |
| `uint* h_list` | List of 16-bit ARINC-664 handles. |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not DNA-664 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Configures Asynchronous Events for the DNA-664.

**Note:**

Events can be enabled with DqRtAsyncEnableEvents function call as shown in the example. Events received with `DqCmdReceiveEvent` contain a `DQEVENT` struct containing a `EV664_ID` struct with multiple `EV664_MSG` structs.

### *4.51.6 DqAdv664Enable*

**Syntax:**

```
int DqAdv664Enable(int hd, int devn, uint32 actions)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 action` | Enable (`AR664_VL_DUAL`) or disable (`FALSE`) |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not DNA-664 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| `DQ_VERSION_MISMATCH` | The IOM version is mismatched because the IOM firmware or DNA-664 firmware are not compatible. |
| Other negative values | low level IOM error |

**Description:**

Enables and disables operations on both channels.

**Note:**

None

### *4.51.7 DqAdv664EnableVLPort*

**Syntax:**

`int DqAdv664EnableVLPort(int hd, int devn, int enable, int arr_sz, vl_port_handles)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int enable` | Enable (TRUE) or disable (FALSE) |
| `int arr_sz` | Number of vl_port_handles in the array |
| `vl_port_handles` | Array of virtual-link or port handle(s) |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not DNA-664 |
| `DQ_NOT_ENOUGH_ROM` | not enough room in the packet |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Enable or disable VLs and Ports.

**Note:**

VL and ports are enabled/disabled separately, so if VL has multiple ports and both VL and ports are disabled first, ports need to be enabled in the array and then VL.
To override this behavior and force associated VL to be enabled pass (`TRUE|AR664_EN_VL_4_PORT`) instead of (`TRUE`).

### *4.51.8 DqAdv664GetBusStat*

**Syntax:**

```
int DqAdv664GetBusStat(int hd, int devn, int chan,
pAR664_BUS_STAT pBusStat, int clear)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chan` | Channel of ARINC-664 board (A, B or dual) |
| `pAR664_BUS_STAT pBusStat` | Pointer where to store the results of one (if A or B) or two (if dual) `AR664_BUS_STAT` |
| `int clear` | Clear statistics after get if not `FALSE`. |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not DNA-664 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Gets statistics from the layer and stores them into a `pAR664_BUS_STAT` structure.

**Note:**

Refer to powerdna.h for the elements included in the `pAR664_BUS_STAT` structure.

### *4.51.9 DqAdv664GetDeviceInfo*

**Syntax:**

`int DqAdv664GetDeviceInfo(int hd, int devn, pAR664_DEV_INFO pDeviceInfo)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `pAR664_DEV_INFO pDeviceInfo` | Device information structure, including incompatibility flags. |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not DNA-664 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| `DQ_VERSION_ MISMATCH` | The IOM version is mismatched because the IOM firmware or DNA-664 firmware are not compatible. |

**Description:**

Gets the device information for this DNA-664 device in the `pAR664_DEV_INFO` struct.

**Note:**

This function is may be called by DqAdv664SetConfig and DqAdv664Enable.

### *4.51.10 DqAdv664RecvMessage*

**Syntax:**

```
int DqAdv664RecvMessage(int hd, int devn, int port_handle,
uint32 flags, int msg_sz, uint8* message, int* received, uint32*
available, uint32* status)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int port_handle` | Port handle to get `message` from |
| `int flags` | Modifier flags |
| `int msg_sz` | Size of `message` buffer |
| `uint8* message` | Message buffer to receive, 1500 bytes max for sampling ports, 8192 for queuing |

**Output:**

| | |
|---|---|
| `int* received` | Number of bytes read from the port |
| `uint32* available` | Number of free messages in the Queuing port queue |
| `uint32* status` | Port status information |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not DNA-664 |
| `DQ_NOT_ENOUGH_ROM` | not enough room in the packet |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |

**Description:**

Read a message from the Sampling or Queuing port.

**Note:**

Function doesn't block until a message is received; just polls for whatever may be available.
A sampling port’s message stays in the port and `status` reports whether this message has been already read (the `AR664_PORT_NEW_DATA` bit is set if port is not yet read).
Queuing port messages are removed from the queue upon read; if the user did not allocate large enough buffer the remainder of the message will be lost. The `status` informs user about this condition, however `received` will contain the size of the message retrieved from the port.
Use `DQL_WR664_SMPL_OFFS(`offset`)` in `flags` to introduce offset in the message of the sampling port (i.e. read sampling port data from the defined offset of msg_size).
The function can timeout if the cube is busy; it is user's responsibility to retry.

Refer to powerdna.h for a list of status flags that can be returned with the `status` parameter.

### *4.51.11 DqAdv664RecvMessageHdr*

**Syntax:**

```
int DqAdv664RecvMessageHdr(int hd, int devn, int port_handle,
uint32 flags, int msg_sz, uint8* message, int* read,
pAR664_HEADERS pHDR, uint32* available, uint32* status)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int port_handle` | Port handle to get `message` from |
| `int flags` | Modifier flags |
| `int msg_sz` | Size of `message` buffer |
| `uint8* message` | Message buffer to receive, 1500 bytes max for sampling ports, 8192 for queuing |
| `pAR664_HEADERS pHDR` | Pointer to store message header information |

**Output:**

| | |
|---|---|
| `int* received` | Number of bytes read from the port |
| `uint32* available` | Number of free messages in the SAP queue |
| `uint32* status` | Port status information |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not DNA-664 |
| `DQ_NOT_ENOUGH_ROM` | not enough room in the packet |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |

**Description:**

Read a message from a service access port (SAP) and include ARINC-664 network header information.

**Note:**

This function works exactly like `DqAdv664RecvMessage` but also returns header information in a `AR664_HEADERS` structure which can include the Ethernet, IP, UDP headers, Ethernet CRC, ARINC-664 sequence number.

Refer to powerdna.h for a list of status flags that can be returned with the `status` parameter.

### *4.51.12 DqAdv664SendMessage*

**Syntax:**

```
int DqAdv664SendMessage(int hd, int devn, int port_handle,
uint32 flags, int msg_sz, uint8* message, int* written, uint32*
available, uint32* status)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int port_handle` | Port handle to get `message` from |
| `int flags` | Modifier flags (including offset for sampling ports) |
| `int msg_sz` | Size of `message` buffer in bytes (octets) |
| `uint8* message` | Queueing port msg_sz is always up to `d_size`=8192 bytes; take care to not exceed the queue `depth`.<br>Sampling port data beyond port's `d_size` will be silently truncated; e.g. d_size must be lower than 1471 for `Lmax`=1518.<br>Once in the port buffer, the on-card transmit 'soft' or 'bin-based' scheduler will take care of sending the messages.<br>Queueing messages may be split into fragments of LMax-47 octets, which takes longer than one BAG. Sampling messages will automatically be resent when `period` is not 0. Details in ARINC-664 User Manual. |

**Output:**

| | |
|---|---|
| `int* written` | Number of bytes written to the port |
| `uint32* available` | Number of free messages in the Queuing port queue |
| `uint32* status` | Port status information |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not DNA-664 |
| `DQ_NOT_ENOUGH_ROM` | not enough room in the packet |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |

**Description:**

Write a message to a Sampling or Queuing port.

**Note:**

Characteristics for Sampling ports:

- A sampling port set to *periodic* will repeat the message at the selected rate; any subsequent write to the periodic port will replace the message which will be sent to ARINC-664 bus upon the next period.

- A sampling port not set to *periodic*, the message is scheduled for immediate delivery once that the BAG (of the VL this port belongs to) expires.
- A sampling port cannot fragment packets, so only the first *MTU* bytes are sent; any additional bytes are not sent (error is returned).

Characteristics for Queuing (and SAP) ports:

- A Queuing Port's message appears on the bus only once.
- Sends up to 8192 bytes; messages larger than MTU bytes are sent as IP fragments.

The function doesn't block, but does return an error if the messaging queue is full. The `status` informs the user of errors; for example, the `AR664_PORT_OVF` bit is set if an overflow occurs. The `written` will contain the number of bytes written to the port.
Use `DQL_WR664_SMPL_OFFS(offset)` in `flags` to introduce offset in the message of the sampling port (i.e. read sampling port data from the defined offset of msg_size).
The function can timeout if the cube is busy; it is user's responsibility to retry.

Refer to powerdna.h for a list of status flags that can be returned with the `status` parameter.

If the ARINC-664 board is not enabled when calling this function then the data is written to the port buffer (as allocated with AddPort or the XML file) and sent immediately once that the transmit scheduler is enabled with DqAdv664Enable / DqAdv664BusControl. The same principle applies if the VL or ARINC-664 port is not enabled.

### *4.51.13 DqAdv664SendMessageHdr*

**Syntax:**

```
     int DqAdv664SendMessageHdr(int hd, int devn, int port_handle,
uint32 flags, int msg_sz, uint8* message, pAR664_HEADERS pHDR, int*
written, uint32* stored, uint32* available, uint32* status);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int port_handle` | Port handle to get `message` from |
| `int flags` | Modifier flags (including offset for sampling ports) |
| `int msg_sz` | Size of `message` buffer |
| `uint8* message` | Message buffer to receive, 1500 bytes max for sampling ports, 8192 for queuing. Any sampling value larger than MTU is truncated; any queuing value larger than MTU is fragmented |
| `pAR664_HEADERS pHDR` | Content of header(s) for the packet |

**Output:**

| | |
|---|---|
| `int* written` | Number of bytes written to the port |
| `uint32* available` | Number of free messages in the Service Access Port queue |
| `uint32* status` | Port status information |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not DNA-664 |
| `DQ_NOT_ENOUGH_ROM` | not enough room in the packet |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Send message to the Service Access Port and include ARINC-664 network header information.

**Note:**

This function works exactly like `DqAdv664SendMessage` but also writes header information in a `AR664_HEADERS` structure which can include the Ethernet, IP, UDP headers, Ethernet CRC, ARINC-664 sequence number, Boeing EDE CRCs and headers.
The additional functionality of DqAdv664SendMessageHdr() is if Ethernet, IP or UDP addresses are set in the `AR664_HEADERS` structure current addresses will be:

- For sampling or queuing ports: will send a one-off transmission and then addresses are reverted to the originally programmed ones.
- For SAP: will modify selected addresses for future use until subsequent call to this function changes them again.

Refer to powerdna.h for a list of status flags that can be returned with the `status` parameter.

### *4.51.14 DqAdv664SetConfig*

**Syntax:**

```
DqAdv664SetConfig(int hd, int devn, int, uint32 action,
pAR664_ARCFG pCfg, char* xml_filename, int* size_tbl,
AR664_CFG_HANDLES** handle_tbl)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int chan` | Reserved. Set to 0. |

| | |
|---|---|
| `uint32 action` | Reserved. Set to 0. |
| `pAR664_ARCFG pCfg` | ARINC-664 channel configuration (see DqAdv664BusControl) |
| `char* xml_filename` | String with filename for pre-created XML file with configuration or NULL if VL/Port configuration is to be created dynamically. |

**Output:**

| | |
|---|---|
| `int* size_tbl` | Size of the resulting handle_tbl |
| `AR664_CFG_HANDLES ** handle_tbl` | Array containing virtual-link and port handles |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not DNA-664 |
| `DQ_BAD_PARAMETER` | see notes. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| `DQ_VERSION_ MISMATCH` | The IOM version is mismatched because the IOM firmware or DNA-664 firmware are not compatible. |
| Other negative values | low level IOM error |

**Description:**

Performs ARINC-664 bus (speed, channels) configuration and VL/Port configuration.

**Note:**

If there are problems with configuration or XML file it will be returned in the handle_tbl; `status` field of the last returned item would contain error code of what has happened.
If the file cannot be opened at all and no VL or port is configured handle_tbl would contain at least one entry with the corresponding `status` field in `AR664_CFG_HANDLES`.

The following cases are checked:

1. Error code in the function call to IOM - `DqAdv664SetConfig()` returns corresponding error code and last entry in `handle_tbl[size_tbl].Line` contains the line number of the last entry from the XML file when error occurs.
2. Error code while validating VL or port parameters from XML file - returns `DQ_BAD_PARAMETER_2` and last entry in `handle_tbl[size_tbl]` contains the line number of the last entry from the XML file when error occurs. The `handle_tbl[size_tbl].Line` contains line in XML file while `handle_tbl[size_tbl].Status` contains `AR664_VL_ERR` flag and parameter in the `.Line` that caused failure.
3. Error code on parsing XML file for VL/Port: `DQ_BAD_PARAMETER` is returned,
   ```
   .Status = err_code|AR664_VL_ERR;
   .Line = err_line;
   ```

4. Error code on parsing XML file for version: `DQ_BAD_PARAMETER` is returned,
   `.Status = PARSE_ERR_BAD_VERSION|AR664_VL_ERR`;
   `.Line = 0`;

### *4.51.15 DqAdv664SendScheduleTable*

**Syntax:**

`DqAdv664SendScheduleTable(int hd, int devn, int flags)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int flags` | Reserved for future use. Set to 0. |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not DNA-664 |
| `DQ_BAD_PARAMETER` | see notes. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function call switches the ARINC-664 card’s transmitter periodic scheduler from ‘soft’ to ‘bin-based’. By default the ‘soft’ scheduling checks as frequently as possible if there is a message and transmits it as soon as the BAG allows another transmission. This function call gives the ARINC-664 card a scheduling table of slots or bins when messages can be transmitted. ‘Soft’ scheduling is fine-grained but quickly becomes overwhelmingly CPU-intensive for more than a few dozen periodic ARINC-664 sampling ports, which can result in dropped transmissions. If your CPU usage rises above 50% you should use this function call to reduce the CPU usage.

**Note:**

This function call forces the API to calculate a scheduler table from the port information that was created from the XML file provided to the `DqAdv664SetConfig` call. Any additional ports added or removed (but not en/disabled) afterwards requires calling this function again. The transmit scheduler operation is set by the `DqAdv664SetEnabled()` function call.

### *4.51.16 DqAdv664ValidateVlPortCfg*

**Syntax:**

```
DqAdv664ValidateVlPortCfg(AR664_VL_CFG* vl, AR664_PORT_CFG*
port, uint32* param_num, int strict);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `AR664_VL_CFG* vl` | Virtual Link configuration struct to check, or NULL. |
| `AR664_PORT_CFG* port` | ARINC-664 Port configuration struct to check, or NULL. |
| `uint32* param_num` | Parameter number that failed, or 0 if values in bounds. |
| `int strict` | `TRUE` to use strict checking of parameters. |

**Return:**

| | |
|---|---|
| `DQ_BAD_PARAMETER_1` | Out-of-bounds parameters in `AR664_VL_CFG` are listed in `param_num`. |
| `DQ_BAD_PARAMETER_2` | Out-of-bounds parameters in `AR664_PORT_CFG` are listed in `param_num`. |
| `DQ_SUCCESS` | successful completion |

**Description:**

Bounds-checking function that checks parameters in a `AR664_VL_CFG` (if `AR664_VL_CFG` is `NULL`) or `AR664_PORT_CFG` (with additional checks if is `AR664_VL_CFG` not `NULL`). The position in the structure of out-of-bounds values are returned as a bit map, with 0 is the first parameter in the `AR664_VL/PORT_CFG` struct and `DQ_BAD_PARAMETER_1/2` as the struct index.

**Note:**

Call DqAdv664ValidateVlPortCfg before every DqAdv664AddVl/Port function call. This function is called by `SetConfig` when configuring from an XML file.

### 4.51.17 DqAdv664VLPortStatus

**Syntax:**

```
int DqAdv664VLPortStatus(int hd, int devn, int arr_sz, int*
vl_port_handles, uint32* status)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int arr_sz` | Number of vl_port_handles and status in the array |
| `int* vl_port_handles` | Array of virtual-link or port handle(s) |

**Output:**

| | |
|---|---|
| `uint32* status` | Array of size `arr_sz` to store statuses for vl_port_handles |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not ARINC-664 |
| `DQ_BAD_PARAMETER_3` | if `arr_sz` is not 0 to `DQ_AR664_LISTSZ` |
| `DQ_NOT_ENOUGH_ROOM` | if there are more arinc664_handles than will fit in a packet |
| `DQ_SEND_ERROR` | unable to send the command to IOM |
| `DQ_DEVICE_BUSY` | if the ARINC-664 layer is busy |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command/had a transfer error |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Returns the current status of each virtual-link or port listed in `vl_port_handles`. This is a batch call of the status from DqAdv664RecvMessage or DqAdv664SendMessage, and is used when running the ARINC-664 in VMAP mode. When using VMAP, there is no special status channel; so you call this function to get the status of many channels.

**Note:**

The `status` bits are defined for each virtual-link (e.g. `AR664_VL_ERR`) or port (e.g. `AR664_PORT_NEW_DATA_OVR` or `AR664_PORT_ERR`) as defined in powerdna.h.

Refer to powerdna.h for a list of status flags that can be returned with the `status` parameter.

## *4.52 DNx-ARINC-708 / DNx-ARINC-708-453 / WXPD layer*

### *4.52.1 DqAdv708Config / DqAdv553ConfigA708*

**Syntax:**

```
DqAdv708Config(int hd, int devn, int channel, uint32 busmode, uint32
f_size, uint32 if_delay_us);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number inside the IOM |
| `int chan` | Channel (0 or 1) |
| `uint32 busmode` | What to monitor/where to transmit: bus A, bus B or both buses |
| `uint32 f_size` | Size of the frame to transmit/receive |
| `uint32 if_delay_us` | Delay between frames, in us |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not DNA-708 |
| `DQ_BAD_PARAMETER` | see notes. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function configures the bus monitor channel.

<`busmode`> defines what bus to listen to or transmit on. OR in one or more of the following constants:

```
#define DQ_L553_BM_LSTN_A    // listen bus A
#define DQ_L553_BM_LSTN_B    // listen bus B
#define DQ_L553_BM_TX_A      // enable transmission on bus A
#define DQ_L553_BM_TX_B      // enable transmission on bus B
#define DQ_L553_A708_TX_BE   // enable bit-swap on ARINC-708 TX data
                             (reverses all bits in all fields)
#define DQ_L553_A708_RX_BE   // enable bit-swap on ARINC-708 RX data
                             (reverses all bits in all fields)
#define DQ_L553_STORE_TS     // store timestamp information
#define DQ_L553_STORE_FLAGS  // store bus flags/status information
```

**Note:**

### *4.52.2 DqAdv708ConfigWXPD*

**Syntax:**

```
int DqAdv708ConfigWXPD(int hd, int devn, int channel,
uint32 busmode, uint32 frame_size_bits, uint32 if_delay_us);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number inside the IOM |
| `int chan` | Channel (0 or 1) |
| `uint32 busmode` | What to monitor/where to transmit: bus A, bus B or both buses |
| `uint32 frame_size_bits` | Reference size of the frame to transmit/receive |
| `uint32 if_delay_us` | Delay between frames, in us |

**Output:**

`None`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not DNA-708 |
| `DQ_BAD_PARAMETER` | see notes. |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function configures for straight ARINC-708/453 communications and also for WXPD operations.

<`busmode`> defines what bus to listen at:

```
#define DQ_L553_BM_LSTN_A    // listen bus A
#define DQ_L553_BM_LSTN_B    // listen bus B
#define DQ_L553_BM_TX_A      // enable transmission on bus A
#define DQ_L553_BM_TX_B      // enable transmission on bus B
#define DQ_L553_STORE_TS     // store timestamp information
#define DQ_L553_STORE_FLAGS  // store bus flags/status information
#define DQ_L553_A708_WXPD    // configure WXPD mode
```

<`frame_size_bits`> defines the reference frame size. It is used to advise firmware of the expected packet size that will be received or transmitted. This size is used to calculate how many frames would fit the packet. For example if the minimum size of the packet expected is 1280 bits and maximum 1800 bits, this parameter will need to be set to 1800 bits to ensure the largest packet will fit.

**Note:**

WXPD is an extension of the ARINC-708/453 protocol. The difference between the two is that ARINC-708/453 uses 1600-bit Manchester-encoded frames staring with a 3-bit SYNC sequence and WXPD uses a variable size frame which contains additional line information for the weather radar. To distinguish between standard 708/453 frames and WXPD frames, a synchronization sequence of eight "0" bits is added before the regular SYNC sequence.

Many constants are derived from the DNx-1553-553. The ARINC-708-453 is based upon `frame_size_bits,` which defines the reference frame size. The WXPD frame size can differ each time the new frame is sent.

The WXPD protocol is supported in PowerDNA ARINC-708/453 boards with logic version 11.AE and software 4.9.1.70 and 4.80.50.

### *4.52.3 DqAdv708Enable*

**Syntax:**

```
int DAQLIB DqAdv708Enable(int hd, int devn, uint32 actions);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number inside the IOM |
| `int actions` | Enable (1) or Disable (0) |

**Output:**

None

**Returns:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 708 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function enables or disables 708 operations for both channels.

**Note:**

This `DqAdv708Enable()` is the same function as `DqAdv553Enable().`

### *4.52.4 DqAdv708SetMode*

**Syntax:**

```
int DqAdv708SetMode(int hd, int devn, int channel, uint32 mode,
uint32 flags);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number inside the IOM |
| `int channel` | Channel (0 or 1) |
| `uint32 mode` | Mode of operation |
| `uint32 flags` | Additional initialization flags |

**Output:**

None

**Returns:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 708 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

<`mode`> - mode of operation:

```
#define DQ_L553_MODE_A708   // Recommended for ARINC-708 network
```

<`flags`> - additional initialization flags:

```
#define DQ_L553_DISCONNECT       // disconnect from the bus (default)
#define DQ_L553_TRANSFORMER      // transformer-coupled, Recommended
                                 //     for ARINC-708 network
                                 //     (including WXPD)
#define DQ_L553_COUPLE_DIRECTLY  // direct-coupling
                                 //    (potential isolation issues)
#define DQ_L553_FORCE_A          // make I/O compliant with MIL-1553A
                                 //    standard (default is MIL-1553B)
```

**Note:**

This `DqAdv708SetMode()` is the same function as `DqAdv553SetMode()`.

### 4.52.5 DqAdv708WriteTxFifo / DqAdv553WriteTxFifoA708

**Syntax:**

```
int DAQLIB DqAdv708WriteTxFifo(int hd, int devn, int channel, uint32
tx_size, uint32* written, uint32* remains,
uint16* tx_data);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number inside the IOM |
| `int channel` | Channel (0 or 1) |
| `uint32 tx_size` | Number of words for transmission |
| `uint32* written` | Pointer to number of words actually written |
| `uint32* remain` | Pointer to available room in the FIFO after those words were written |
| `uint32* tx_data` | Pointer to data to transmit |

**Output:**

| | |
|---|---|
| `uint32* written` | Number of words actually written |
| `uint32* remain` | Available room in the FIFO after those words were written |
| `uint32* tx_data` | Data to transmit |

**Returns:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 708 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_FIFO_OVERFLOW` | device FIFO overflowed |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function writes data to the Tx FIFO for the ARINC-708 version of the layer.

**Note:**

The function either completes the data write if there is enough room in the FIFO or fails, returning the `DQ_FIFO_OVERFLOW` flag.

### *4.52.6 DqAdv708ReadRxFifo / DqAdv553ReadRxFifoA708*

**Syntax:**

```
int DAQLIB DqAdv708ReadRxFifo(int hd, int devn, int channel, uint32
rq_size, uint32* received, uint32* remains,
uint16* rx_data);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number inside the IOM |
| `int channel` | Channel (0 or 1) |
| `uint32 rq_size` | Number of words for requested |
| `uint32* received` | Pointer to number of words received |
| `uint32* remains` | Pointer to the number of words that remain in the FIFO |
| `uint32* rx_data` | Pointer to the received data |

**Output:**

| | |
|---|---|
| `uint32* written` | Number of words received |
| `uint32* remain` | Number of words that remain in the FIFO |
| `uint32* tx_data` | Received data |

**Returns:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 708 |
| `DQ_BAD_PARAMETER` | Configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads data from the Rx FIFO for the ARINC-708 version of the layer.

**Note:**

The function returns up to "`rq_size`" words.

### *4.52.7 DqAdv708WriteTxFifoWXPD*

**Syntax:**

```
int DAQLIB DqAdv708WriteTxFifoWXPD(int hd, int devn, int channel,
uint32 tx_size_bytes, uint32 last_byte_bits, uint32 sync_size, uint32
delay_us, uint32* written_bytes, uint32* remain_bytes, uint8*
tx_data);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number inside the IOM |
| `int channel` | Channel (0 or 1) |
| `uint32 tx_size_bytes` | Number of bytes including an incomplete one for transmission |
| `uint32 last_byte_bits` | Number of bits in the last byte (0 == 8) |
| `uint32 sync_size` | Send WXPD SYNC pulses (0 == no SYNC, 8 == use SYNC) |
| `uint32 delay_us` | Delay before the beginning of the frame (or between auto frame retransmissions) in 100 us increments |
| `uint32* written_bytes` | Pointer to number of bytes actually written |
| `uint32* remain_bytes` | Pointer to available room in the FIFO after those words were written (in bytes) |
| `uint8* tx_data` | Pointer to data to transmit |

**Output:**

| | |
|---|---|
| `uint32* written_bytes` | Number of bytes actually written |
| `uint32* remain_bytes` | Available room in the FIFO after those words were written (in bytes) |
| `uint8* tx_data` | Data to transmit |

**Returns:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 708 |
| `DQ_BAD_PARAMETER` | configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_FIFO_OVERFLOW` | device FIFO overflowed |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function writes data to the Tx FIFO for the ARINC-708-453 and WXPD versions of the layer. WXPD requires use of eight synchronization "zeroes" before the 3-bit SYNC sequence.

**Note:**

The function either completes the data write if there is enough room in the FIFO or fails, returning the `DQ_FIFO_OVERFLOW` flag.

Important: Please note, unlike `DqAdv553WriteTxFifoA708()` this function can write a single WXPD frame due to the variable size configuration.

### *4.52.8 DqAdv708ReadRxFifoWXPD*

**Syntax:**

```
int DAQLIB DqAdv708ReadRxFifoWXPD(int hd, int devn, int channel,
uint32 rq_max_size_bytes, uint32* received_bytes,
uint32* last_byte_bits, uint32* frame_delay, uint32* remains,
uint8* rx_data);
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Device number inside the IOM |
| `int channel` | Channel (0 or 1) |
| `uint32 rq_max_size_bytes` | Maximum number of bytes allocated in rx_data |
| `uint32* received_bytes` | Pointer to number of bytes received, including incomplete one |
| `uint32* last_byte_bits` | Pointer to number of bits in the last byte (0 == 8) |
| `uint32* frame_delay` | Pointer to time since last received frame |
| `uint32* remains` | Pointer to the number of bytes that remain in the FIFO |
| `uint8* rx_data` | Pointer to the received data |

**Output:**

| | |
|---|---|
| `uint32* received_bytes` | Number of bytes received, including incomplete one |
| `uint32* last_byte_bits` | Number of bits in the last byte (0 == 8) |
| `uint32* frame_delay` | Time since last received frame |
| `uint32* remains` | Number of words that remain in the FIFO |
| `uint8* rx_data` | Received data |

**Returns:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not an 708 |
| `DQ_BAD_PARAMETER` | configuration parameters are incorrect |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |

| | |
|---|---|
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**
This function reads data from the Rx FIFO for the ARINC-708-453 and WXPD versions of the layer. It returns a single frame stored in the FIFO along with the number of received bytes, number of valid bits available in the last byte (zero if all are valid), delay from the previous frame, and number of bytes still stored in the FIFO. The receive FIFO is limited to 512 words.

**Notes:**

This function by default waits until a full WXPD frame is detected. If the configured frame size (set in `DqAdv708ConfigWXPD)` does not match the actual frame size, the end of frame may be incorrectly detected and cause the function to return too much data. To enable receiving partial frames up to `rq_max_size_bytes`, logically OR `rq_max_size_bytes` with `DQ_L533_WXPD_ENABLE_PARTFRAMES`.

- IMPORTANT: Unlike `DqAdv708ReadRxFifo(),` this function can read a single WXPD frame due to the variable size configuration.

## 4.53 DNx-PL-820 Layer

### 4.53.1 DqAdv820SetCfg

**Syntax:**

```
int DqAdv820SetCfg(int hd, int devn, uint32 cfg_mask, pPL820CFG pCfg, uint32*
status)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from DqOpenIOM() |
| `int devn` | Layer inside the IOM |
| `uint32 cfg_mask` | <reserved> |
| `pPL820CFG pCfg` | Pointer to the configuration structure |

**Output:**

| | |
|---|---|
| `uint32* status` | content of the status register PL820_STS |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by devn does not exist or is not a PL-820 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| `DQ_NO_MEMORY` | insufficient memory to perform command |
| Other negative values | low level IOM error |

**Description:**

Sets up the NIS/Port pin direction, as well as the SPI configuration.

Configuration is passed as the following structure:

```
typedef struct {
  uint32 prm_flags;  // flags to validate the following parameters

  // Control for external pins:
  // DQ_L820_PORTS is set to 4 (4*32-bit param to cover 105 pins)
  uint32 port_dir[DQ_L820_PORTS];    // direction of the user pins on
                                     //     DB62 (input:0 or output:1)
  uint32 port_out[DQ_L820_PORTS];    // output values

  // PLL section
  // DQ_L820_NIS_PORTS is set to 2 (2*32-bit param to cover 64 NIS-IO)
  uint32 pll_freq[DQ_L820_NIS_PORTS]; // requested PLL frequency
                                      //   in Hz
  uint32 pll_div[DQ_L820_NIS_PORTS];  // post-divider
// SPI configuration
uint32 spi_cfg_fpga[DQ_L820_NIS_PORTS]; // SPIx configuration
uint32 spi_wm[DQ_L820_NIS_PORTS];       // SPIx watermark register
uint32 spi_acfg[DQ_L820_NIS_PORTS];  // SPIx address
                                     //  configuration register
uint32 spi_div[DQ_L820_NIS_PORTS];   // SPIx clock divider
                                     // from PLL0 (24 MHz default)
uint32 spi_cfg_cpld[DQ_L820_NIS_PORTS]; // SPIx configuration

// NIS connects FPGA and CPLD. There are 64 pins total with the
//     exception of pins 2,4,8,12,16 and 39
// When direction is set to input on FPGA a command is sent
//     mirror FPGA configuration
// <nis_dir> and <nis_out> is from FPGA side.
uint32 nis_dir[DQ_L820_NIS_PORTS];   // direction of NIS pins
                                         //  (input:0 or output:1)
uint32 nis_out[DQ_L820_NIS_PORTS];   // output values: FPGA
uint32 nis_out_cpld[DQ_L820_NIS_PORTS]; // output values: CPLD

} PL820CFG, *pPL820CFG;
```

`<prm_flags>` defines which parameters are valid, i.e. which subsystems needs to be configured. It is possible to call `DqAdv820SetCfg()` for each subsystem individually. To configure more than one (or all) subsystems at the same time logically OR flags together:

The following flags are to be used:

```
#define PL608_PRMFPGA_SPI1 (1L<<8)     // configure SPI1, FPGA side
#define PL608_PRMFPGA_SPI0 (1L<<7)     // configure SPI0, FPGA side

#define PL608_PRMCPLD_SPI1 (1L<<6)     // configure SPI1, CPLD side
#define PL608_PRMCPLD_SPI0  (1L<<5)     // configure SPI0, CPLD side

#define PL608_PRMFPGA_PLL1  (1L<<4)     // configure PLL1
#define PL608_PRMFPGA_PLL0  (1L<<3)     // configure PLL0
                                        //   (CycloneIII required)

#define PL608_PRMCPLD_PORT  (1L<<2)     // configure port side
#define PL608_PRMCPLD_NIS  (1L<<1)     // configure NIS side
#define PL608_PRMFPGA_NIS  (1L<<0)     // configure NIS side
```

*Configuring SPI:* The SPI ports must be configured before use.
`<spi_cfg_fpga>` and `<spi_cfg_cpld>` are used to configure the SPIs:
For example, use the following bits and macros:

```
cfg.spi_cfg_fpga[spi_port] =
     PL820_SPI_CFG_CSI |
     PL820_SPI_CFG_ABE(ADDRM_START+ADDRM_SIZE-1) |
     PL820_SPI_CFG_ABS(ADDRM_START) |
     PL820_SPI_CFG_PSW |
     PL820_SPI_CFG_DBE(DATAM_START+DATAM_SIZE-1) |
     PL820_SPI_CFG_MWE |
     PL820_SPI_CFG_DBS(DATAM_START);
```

| | |
|---|---|
| PL820_SPI_CFG_CSI | CS is inverted (i.e. negative) |
| PL820_SPI_CFG_ABE(ADDRM_START+ADDRM_SIZE-1) | Last bit of address |
| PL820_SPI_CFG_ABS(ADDRM_START) | Address starts at this bit |
| PL820_SPI_CFG_PSR | Receive data is valid on the rising edge |
| PL820_SPI_CFG_PSW | Transmit data is valid on the rising edge |
| PL820_SPI_CFG_DBE(DATAM_START+DATAM_SIZE-1) | Data ends at this bit |
| PL820_SPI_CFG_MWE | Multi-word w/o address in every word enabled |
| PL820_SPI_CFG_DBS(DATAM_START) | Data starts at bit zero |

Note that for proper operations, both FPGA and CPLD must receive the same configuration for the exception of `PL820_SPI_CFG_PSR` bit. The PSR bit defines the clock edge when data becomes valid and needs to be set into opposite states.
`PL820_SPI_CFG_DBS, _DBE, _ABS, _ABE` constants define bit numbers where the data word starts and ends, and where the address part starts and ends. For example, for a 32-bit word with an 8-bit address and 24 bits of data, these constants have to be set to 0, 23, 24, 31 respectively.

`<spi_div>` needs to be programmed with a value of the 24 MHz base clock divider minus one. This field defines frequency on the clock line of SPIx interface. Each SPI interface may have its own clock rate.

`<spi_acfg>` is the address configuration register:

```
PL820IS_SPI_ACFG_REN |              // CPLD->FPGA enable
PL820IS_SPI_ACFG_WEN |              // FPGA->CPLD enable
PL820IS_SPI_ACFG_RAD(SPIcommand) |  // 8-bit read address start,
                                    //    CPLD->FPGA address
PL820IS_SPI_ACFG_WAD(SPIcommand);   // 8-bit write address start,
                                    //    FPGA->CPLD address
```

`<spi_wm>` is the watermark register. Sets watermark level in FIFO to generate an interrupt. Both TX and RX FIFOs are 1024 words. Interrupt Service Routine (ISR) implementation for aVMap mode reserved for future implementation.

*Configuring External IO Ports:*
`<port_dir>` controls whether the pin is an input (0) or output (1). Each IO pin is mapped to a bit number and port number. The bit number is the position in the 32-bit word, and the port number is the array index value (`DQ_L820_PORTS` or 4 ports).

```
// Port0 - CPLD - bits 0..31 = I/O 0..31
// Port1 - CPLD - bits 0..19 = I/O 32..50
// Port2 - FPGA - bits 0..31 = I/O 51..82
// Port3 - FPGA - bits 0..21 = I/O 83..104
```

`<port_out>` sets the output value (for DIO configured as outputs).

*Configuring Internal NIS-IO:*
`<nis_dir>` controls whether NIS I/O pin is an input (0) or output (1). The internal NIS connector is 64 bits; therefore, the first 32 bits are set using array index 0, and the 2nd are set using index 1.
`<nis_out>` sets the output value for the NIS pins on the base-board (FPGA) that are configured as outputs.
`<nis_out_cpld>` sets the output value for the NIS pins on the daughter-board (CPLD) that are configured as outputs.

**Note:**

1. For the NIS interface, the firmware has safeguards to prevent driving the same pin from both the CPLD and FPGA side, in case of user error.
2. The NIS interface has some bits reserved for CLI SPI to access registers from FPGA on CPLD side and clock lines:

```
// bits 2, 4, 8, 12, 16, 39 should be always written with 0s
#define PL820_NIS0_MASK (~0x00011114L)
#define PL820_NIS1_MASK (~0x00000080L)
```

3. Registers are described in the PL-820 user manual, logic document and in powerdna.h file starting with the `DQ_L820_LINES` define.

### 4.53.2 DqAdv820ReadReg

**Syntax:**

```
int DqAdv820ReadReg(int hd, int devn, uint32 reg, uint32* value)
```

**Command:**

DaqBIOS

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from DqOpenIOM() |
| `int devn` | Layer inside the IOM |
| `uint32 reg` | register offset |

**Output:**

| | |
|---|---|
| `uint32* value` | returned value of the register |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by devn does not exist or is not a PL-820 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

The function reads a register from either CPLD or FPGA. FPGA address range is 0x0 to 0x2ffc and CPLD address range is from 0x3000 to 0x31fc.

**Notes:**

1. Registers are described in the PL-820 user manual, logic document and in powerdna.h file starting with the `DQ_L820_LINES` define.
2. Write register cannot be used to transfer multi-word (i.e. when address comes in the first SPI word only) sequence because logic resets transmission every time output FIFO becomes empty. Use `DqAdv820ReadSPI()` instead.

### *4.53.3 DqAdv820WriteReg*

**Syntax:**

```
int DqAdv820WriteReg(int hd, int devn, uint32 reg, uint32 value)
```

**Command:**

DaqBIOS

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from DqOpenIOM() |
| `int devn` | Layer inside the IOM |
| `uint32 reg` | register offset |
| `uint32 value` | value to write to the register |

**Output:**

none

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by devn does not exist or is not a PL-820 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

The function writes a register on either CPLD or FPGA. FPGA address range is 0x0 to 0x2ffc and CPLD address range is from 0x3000 to 0x31fc.

**Notes:**

1. Registers are described in the PL-820 user manual, logic document and in powerdna.h file starting with `DQ_L820_LINES` define.

### *4.53.4 DqAdv820WriteSPI*

**Syntax:**

```
int DqAdv820WriteSPI(int hd, int devn, uint32 spi, uint32 size32, uint32* data,
uint32* written, uint32* avail)
```

**Command:**

DaqBIOS

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from DqOpenIOM() |
| `int devn` | Layer inside the IOM |
| `uint32 spi` | SPI interface 0 or 1 |
| `uint32 size32` | number of uint32 words to write |
| `uint32* data` | array of uint32s the size of <size32> |

**Output:**

| | |
|---|---|
| `uint32* written` | actual number of words written |
| `uint32* available` | remained capacity of FIFO after words were written |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by devn does not exist or is not a PL-820 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

The function writes words into the SPI FIFO (to be transmitted from FPGA to CPLD side). If the word size is from 8 to 32 bit then one write takes one uint32 to complete. If the word size is from 33 to 64 it takes two uint32s to complete one write. In the later mode the write will not proceed if you write odd number of words and either return zero <`written`> or ignore the extra word. <`written`> returns the actual number of uint32s written to the SPI FIFO

SPI must be configured before use. To do that `DqAdv820SetCfg()` must be called.
As an example, use the following to configure the FPGA side use the following bits and macros:

```
PL820_SPI_CFG_CSI |                 // CS is inverted (i.e. negative)
PL820_SPI_CFG_ABE(ADDRM_START+ADDRM_SIZE-1) |  // last bit of address
PL820_SPI_CFG_ABS(ADDRM_START) |         // address starts at this bit
PL820_SPI_CFG_PSW |      // Transmit data is valid on the rising edge
PL820_SPI_CFG_DBE(DATAM_START+DATAM_SIZE-1) | //Data ends at this bit
PL820_SPI_CFG_MWE |   // multi-word w/o address in every word enabled
PL820_SPI_CFG_DBS(DATAM_START)             // Data starts at bit zero
```

Notice, that for proper operations, both FPGA and CPLD must receive the same configuration for the exception of `PL820_SPI_CFG_PSR` bit. The PSR bit defines the clock edge when data becomes valid and needs to be set into opposite states.
`PL820_SPI_CFG_DBS, _DBE, _ABS, _ABE` constants define bit numbers where the data word starts and ends, and where the address part starts and ends. For example, for a 32-bit word with an 8-bit address and 24 bits of data, these constants have to be set to 0, 23, 24, 31 respectively.

<`spi_div`> needs to be programmed with a value of the 24 MHz base clock divider minus one. This field defines frequency on the clock line of SPIx interface. Each SPI interface may have its own clock rate.

**Notes:**

In multi-word mode the minimum number of uint32 to write is two for 8-32-bit words and four for 33-64 bit words. If the amount of data is less than that no words will be written. Also, multi-word transaction must be written in a single call to `DqAdv820WriteSPI()`.

### *4.53.5 DqAdv820ReadSPI*

**Syntax:**

```
int DqAdv820ReadSPI(int hd, int devn, uint32 spi, uint32 size32, uint32* data,
uint32* read, uint32* avail)
```

**Command type:**

DaqBIOS

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from DqOpenIOM() |
| `int devn` | Layer inside the IOM |
| `uint32 spi` | SPI interface 0 or 1 |
| `uint32 size32` | number of uint32 words to write |

**Output:**

| | |
|---|---|
| `uint32* data` | array of uint32s the size of <size32> |
| `uint32* read` | actual number of words read |
| `uint32* avail` | number of words available in the FIFO |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by devn does not exist or is not a CT-604 |
| `DQ_BAD_PARAMETER` | counter number is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads words from the SPI FIFO (to be transmitted from CPLD to FPGA side). If the word size is from 8 to 32 bits then each word is represented in one uint32. If the word size is from 33 to 64 it takes two uint32s to store the word.

**Notes:**

Only the data part of the word is stored, not the address one. Address bits are replaced with zeroes.

## 4.54 DNA-PC-910/911/912/913 layers

### 4.54.1 DqAdv91xRead

**Syntax:**

```
int DqAdv91xRead(int hd, int devn, uint32* status, uint32*
bdata, double* fdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `uint32 *status` | Returns status value , 1 uint32 |
| `uint32 *bdata` | Raw binary data, an array of 5 values, (`NULL` if not required) |
| `double *fdata` | Converted data, an array of 5 values, (`NULL` if not required) |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a PC-91x |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function returns the status and ADC readings from a 91x series layer.

PC-91X ADC channel assignments, indices for `bdata` [] and `fdata` []:

```
DQ_PC91X_CH_EXT_V           (0)        // volts,   external JIO connector
DQ_PC91X_CH_INPUT_I         (1)        // amps,    current used by DC/DC converters
DQ_PC91X_CH_INT_V           (2)        // volts,   internal DNx system power
DQ_PC91X_CH_DCDC_INPUT_V    (3)        // volts,   voltage at input to DC/DC
DQ_PC91X_CH_THERM           (4)        // degrees C,    temperature of the ADC IC
```

bit defines for status information returned to `*status`
DQ_91X_STS_JMAIN_ON (1L<<2) // = 1 if JMAIN (DNA) is used as a power source
DQ_91X_STS_JIO_ON (1L<<1) // = 1 if JIO (Front connector) is used as a power source
DQ_91X_STS_JIO (1L<<0) // JIO Input Power status, 0=fault, no power

**Note:**

None.

### 4.54.2 DqAdv91xSetConfig

**Syntax:**

`int DqAdv91xSetConfig(int hd, int devn, uint32 src, uint32 vsel)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int src` | Voltage source selector, internal, JIO or JIO w/autoswitch |
| `int vsel` | Select the voltage options or turn power off |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a PC-91x |
| `DQ_BAD_PARAMETER` | Src or vsel are not one of the specified constants |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets the operating state of the PC91x.

Use one of the following #defined constants for `src:`
DQ_91X_INTERNAL_POWER // use internal power from DNx system
DQ_91X_EXT_JIO_POWER // use power from DB-37 (JIO) connector
DQ_91X_EXT_JIO_AUTOSWITCH // use power from DB-37 (JIO) connector but switch to internal power when JIO voltage is too low ( approx 9V).

Use one of the following #defined constants for `vsel.` Use the constants that apply to your particular PC-91x model.

For PC-910:
DQ_91X_POWER_OFF // set power off
DQ_910_P10_ 15W // +10V 15Watts
DQ_910_P10_N10_30W //+-10V 30Watts

DQ_910_N10_15W // -10V 15Watts

For PC-911:
DQ_91X_POWER_OFF // set power off
DQ_911_P5_N5_12W // +-5V 12Watts
DQ_911_P10_N10_24W //+-10V 24Watts
DQ_911_P15_N15_36W // +-15V 36Watts
DQ_911_P15_N5_24W // +15V,-5V 24Watts
DQ_911_P5_N15_24W // +5V,-15V 24Watts

For PC-912:
DQ_91X_POWER_OFF // set power off
DQ_912_P12_20W // +12V 20Watts
DQ_912_P24_40W // +24V 40Watts

For PC-913:
DQ_91X_POWER_OFF // set power off
DQ_913_P15_N15_12W // +-15V 12Watts
DQ_913_P30_N30_24W // +-30V 24Watts
DQ_913_P45_N45_36W // +-45V 36Watts
DQ_913_P45_N15_24W // +45V,-15V 24Watts
DQ_913_P15_N45_24W // +15V,-45V 24Watts

**Note:**

None.

## 4.55 DNA-PC-921 layer

### 4.55.1 DqAdv921Read

**Syntax:**

```
int DqAdv921Read(int hd, int devn, uint32* status, uint32*
bdata, double* fdata)
```

**Command:**

Read back diagnostic data.

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `uint32 *status` | Returns status information in 1 uint32 (`NULL` if not required) |
| `uint32 *bdata` | Raw binary data, an array of 5 values (`NULL` if not required) |
| `double *fdata` | Converted data, an array of 5 values (`NULL` if not required) |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a PC-921 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |

| | |
|---|---|
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function returns the status and ADC readings from a PC-921 layer.

Bit defines for status information returned to `status`:

```
DQ_PC921_STS_PFAIL  (1L<<6)  // = 0 - power failure is detected
DQ_PC921_STS_IN3    (1L<<5)  // reserved
DQ_PC921_STS_IN2    (1L<<4)  // reserved
DQ_PC921_STS_IN1    (1L<<3)  // reserved
DQ_PC921_STS_IN0    (1L<<2)  // reserved
DQ_PC921_STS_CCHRG  (1L<<1)  // =0 - hold-up capacitor is charged
DQ_PC921_STS_CDCHRG (1L<<0)  // =0 - hold-up capacitor is discharged
```

ADC channel assignments in `bdata` [] and `fdata` []:

```
DQ_PC921_CH_HOLD_V    (0)   // volts; voltage on hold-up capacitors
                               (typically 38V)
DQ_PC921_CH_INPUT_I   (1)   // amps; current supplied to Cube
DQ_PC921_CH_INT_V     (2)   // volts; input voltage on DB-37 connector
DQ_PC921_CH_3_3_V     (3)   // volts;  internal 3.3V supply
DQ_PC921_CH_THERM     (4)   // degrees C; temperature inside the ADC IC
```

### 4.55.2 DqAdv921SetConfig

**Syntax:**

```
int DqAdv921SetConfig(int hd, int devn, uint32 sync, uint32
power)
```

**Command:**

Turn on/off power to Cube.

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 sync` | reserved |
| `uint32 power` | One of the following power configuration constants: |

```
DQ_PC921_CFG_PWR_ON       //leave power ON
DQ_PC921_CFG_POFF_EN      //Turn power OFF
DQ_PC921_CFG_IN_0_PWR_OFF //reserved
DQ_PC921_CFG_IN_1_PWR_OFF //reserved
DQ_PC921_CFG_IN_2_PWR_OFF //reserved
DQ_PC921_CFG_IN_3_PWR_OFF //reserved
```

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a PC-921 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |

| | |
|---|---|
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**
This function controls the layer's power on/off behavior.

## *4.56 DNA-PC-925 layer*

### *4.56.1 DqAdv925Read*

**Syntax:**

```
int DqAdv925Read(int hd, int devn, uint32* status, uint32* rpm,
uint32* bdata, double* fdata)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |

**Output:**

| | |
|---|---|
| `uint32 *status` | Returns status value , 1 uint32 (`NULL` if not required) |
| `uint32 *rpm` | Returns the RPM of the fan (`NULL` if not required) |
| `uint32 *bdata` | Raw binary data, an array of 4 values, (`NULL` if not required) |
| `double *fdata` | Converted data, an array of 4 values, (`NULL` if not required) |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a PC-925 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function returns the status, rpm and ADC readings from a 925 layer.

PC-925 ADC channel assignments, indices for `bdata` [] and `fdata` []:

| | | |
|---|---|---|
| DQ_PC925_CH_FAN_I | (0) | // amps, used by the fan |
| DQ_PC925_CH_INPUT_V | (1) | // volts, at input to PC925 layer |
| DQ_PC925_CH_FAN_V | (2) | // volts, voltage at input to fan |
| DQ_PC925_CH_THERM | (3) | // degrees C, temperature of the ADC IC |

bit define for status information returned to `*status`

DQ_91X_STS_JIO (1L<<0) // Fan is enabled

**Note:**

None.

## 4.57 DNR-PWR layer

### 4.57.1 DqAdvDnrpRead

**Syntax:**

`int DqAdvDnrpRead(int hd, int devn, int CLSize, uint32 *cl, uint32 *bData, double *fData)`

**Command:**

Read out parameters from the DNR Power layer

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | number of channels |
| `uint32 *cl` | pointer to channel list |
| `uint32 *bData` | pointer to raw data received from device |
| `double *fData` | pointer to store converted data (NULL if not required) |

**Output:**

| | |
|---|---|
| `uint32 *bData` | received binary data |
| `double *fData` | received converted data |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a power layer |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, `bData` is NULL, or a channel number in `cl` is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads voltage, current, and temperature parameters from a DNR-POWER-DC layer. The layer uses a 24- bit converter that reads data once a second.

The following channels are defined (as well as macros to convert data):

```
DQ_L2_ADC_TEMP2     (12)       // Temperature 2 (Code-0x800000)*149nV)/0.00295K***
DQ_L2_ADC_TEMP1     (11)       // Temperature 1 (Code-0x800000)*149nV)/0.00295K***
DQ_L2_ADC_I_IN      (10)       // Input current (Code-0x800000)*1.788uA***
```

```
DQ_L2_ADC_V_FAN     (9)        // Fan voltage (Code-0x800000)*149nV*23.1V
DQ_L2_ADC_V_1_2     (8)        // 1.2V source (Code-0x800000)*149nV*23.1V
DQ_L2_ADC_V_1_5     (7)        // 1.5V source (Code-0x800000)*149nV*23.1V
DQ_L2_ADC_V_IN      (6)        // Input voltage (Code-0x800000)*149nV*45.3V
DQ_L2_ADC_V_24NIC   (5)        //  24V NIC (Code-0x800000)*149nV*23.1V
DQ_L2_ADC_V_24DNR   (4)        //  24V DNR (Code-0x800000)*149nV*23.1V
DQ_L2_ADC_V_3_3NIC  (3)        // 3.3V NIC (Code-0x800000)*149nV*23.1V
DQ_L2_ADC_V_3_3DNR  (2)        // 3.3V DNR (Code-0x800000)*149nV*23.1V
DQ_L2_ADC_V_2_5NIC  (1)        // 2.5V NIC (Code-0x800000)*149nV*23.1V
DQ_L2_ADC_V_2_5DNR  (0)        // 2.5V DNR (Code-0x800000)*149nV*23.1V
```

- Channel 0x30 is a direct write/read to/from LED register (i.e., write occurs upon command execution).
- Channel 0x31 is a direct write/read into configuration register.
- Channels 0x32 and 0x33 have similar functionality; however, the actual read/write is performed by a periodic supervisor routine.

```
#define DQ_L2_DNRP_LED_CH   0x30    // direct R/W to LED register
#define DQ_L2_DNRP_FAN_CH   0x31    // direct R/W to CFG register
#define DQ_L2_DNRP_LED_MNGD 0x32    // managed R/W to LED register
#define DQ_L2_DNRP_FAN_MNGD 0x33    // managed R/W to CFG register
```

**Notes:**

Channels 10-12 are calibrated values. Using the given equations above on the raw data will give close approximations of the actual value, but it is best to use the calibrated data returned in `fData.`

### *4.57.2 DqAdvDnrpSetConfig*

**Syntax:**

```
int  DqAdvDnrpSetConfig(int hd, int devn, uint32 action, uint32* config)
```

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 action` | What to set:<br>`DQ_L2_SET_CONFIG - set configuration (Fan off and 24V off) (uint32)`<br>`DQ_L2_SET_LED - set LEDs (uint32)`<br>`DQ_L2_SET_LIMITS - set over and under limits (16*uint32 + 16*uint32)` |
| `uint32 *config` | pointer to the data array |

**Output:**

**None**

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a power |

| | |
|---|---|
| | layer |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, `bData` is NULL, or a channel number in `cl` is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up the following parameters:

`DQ_L2_SET_CONFIG –` this option allows turning off fans (only in case the controller is cooler than 45C) and 24V DC/DC (powers isolated sides of most analog input and output layers). Use constants DQ_L2_STS_FANOFF and DQ_L2_STS_DC24OFF to do that.

`DQ_L2_SET_LED –` allows you to turn on/turn off user defined LEDs: (DQ_L2_LED_USR, DQ_L2_LED_IO, DQ_L2_LED_ATT). The rest of LEDs are controlled by the periodic routine, which is executed once a second. To shut automatic update off and get compete access to LEDs, use DQ_L2_LED_STOP_UPDATE flag along with LED flags. A write without this flag restores periodic status indication.

```
// LED allocation
#define DQ_L2_LED_VIN       (1L<<0)    // input voltage (on if exists, blinks – over
the limit, off - failed)
#define DQ_L2_LED_IIN       (1L<<1)    // input current
#define DQ_L2_LED_1_5       (1L<<2)    // 1.5V
#define DQ_L2_LED_FAN       (1L<<3)    // FAN is on
#define DQ_L2_LED_USR       (1L<<4)    // User controlled
#define DQ_L2_LED_IO        (1L<<5)    // I/O (user)
#define DQ_L2_LED_OVRT      (1L<<6)    // Overtemperature (red)
#define DQ_L2_LED_ATT       (1L<<7)    // Attention (red)
#define DQ_L2_LED_24_DNR    (1L<<8)    // DNR side 24V
#define DQ_L2_LED_24_NIC    (1L<<9)    // NIC side 24V
#define DQ_L2_LED_3_3_DNR   (1L<<10)   // DNR side 3.3V
#define DQ_L2_LED_3_3_NIC   (1L<<11)   // NIC side 3.3V
#define DQ_L2_LED_STOP_UPDATE (1L<<31) // stop LED updates in periodic routine

#define DQ_L2_LED_BLINK(N)  ((N<<16)|N) // up and blink
```

`DQ_L2_SET_LIMITS –` sets under and over the limit levels (used if interrupt detection is enabled, currently not supported). Format is straight binary 24-bit; use 32-channel 32-bit array (16 over limit and 16 under limit values).

**Note:**

None

## 4.58 DNx-POWER-1G layer

### 4.58.1 DqAdvDnxpRead

**Syntax:**

```
int  DqAdvDnxpRead(int hd, int devn, int CLSize, uint32 *cl,
uint32 *bData, double *fData)
```

**Command:**

Read out parameters from the CPU Power layer.

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int CLSize` | number of channels |
| `uint32 *cl` | pointer to channel list |
| `uint32 *bData` | pointer to raw data received from device |
| `double *fData` | pointer to store converted data (NULL if not required) |

**Output:**

| | |
|---|---|
| `uint32 *bData` | received binary data |
| `double *fData` | received converted data |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a power layer |
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, `bData` is NULL, or a channel number in `cl` is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function reads voltage, current and temperature parameters from the CPU Power layer. In a RACK chassis, this layer is part of the double board that is located at the CPU/NIC slot. In a cube chassis, this layer is directly below the CPU. The layer uses a 24 bit converter that reads data once a second.

The following channels are defined (as well as macros to convert data):

```
// ADC allocation 0x40
#define DQ_L4_ADC_TEMP2     (15)     // Temperature 2 (Code-0x800000)*2.5/1570)
#define DQ_L4_ADC_I_1_5     (14)     // 1.5V output current (Code-0x800000)*7.45uA
#define DQ_L4_ADC_TEMP1     (13)     // Temperature 1 (Code-0x800000)*2.5/1570)
#define DQ_L4_ADC_I_3_3     (12)     // 3.3V output current (Code-0x800000)*7.45uA
#define DQ_L4_ADC_GND_3     (11)     // internal reference ground
#define DQ_L4_ADC_I_IN      (10)     // Input current (Code-0x800000)*1.788uA
#define DQ_L4_ADC_V_FAN     (9)      // Fan voltage (Code-0x800000)*149nV*23.1V
#define DQ_L4_ADC_V_1_2     (8)      // 1.2V source (Code-0x800000)*149nV*23.1V
#define DQ_L4_ADC_V_1_5     (7)      // 1.5V source (Code-0x800000)*149nV*23.1V
#define DQ_L4_ADC_V_IN      (6)      // Input voltage (Code-0x800000)*149nV*45.3V
```

```
#define DQ_L4_ADC_GND_2     (5)      // internal reference ground
#define DQ_L4_ADC_V_24DNR   (4)      // 24V DNR (Code-0x800000)*149nV*23.1V
#define DQ_L4_ADC_V_CAP     (3)      // V capacitor
#define DQ_L4_ADC_V_3_3DNR  (2)      // 3.3V DNR (Code-0x800000)*149nV*23.1V
#define DQ_L4_ADC_GND       (1)      // internal reference ground
#define DQ_L4_ADC_V_2_5DNR  (0)      // 2.5V DNR (Code-0x800000)*149nV*23.1V
```

- Channels from 0x10 to 0x1c return over-the-limit settings for abovementioned channels.
- Channels from 0x20 to 0x2c return under-the-limit settings for abovementioned channels.
- Channel 0x30 is a direct write/read to/from LED register (i.e. write occurs upon command execution).
- Channel 0x31 is a direct write/read into configuration register.
- Channels 0x32 and 0x33 have similar functionality; however actual read/write is performed by periodic supervisor routine.

```
#define DQ_L2_DNRP_LED_CH   0x30    // direct R/W to LED register
#define DQ_L2_DNRP_FAN_CH   0x31    // direct R/W to CFG register
#define DQ_L2_DNRP_LED_MNGD 0x32    // managed R/W to LED register
#define DQ_L2_DNRP_FAN_MNGD 0x33    // managed R/W to CFG register
```

**Notes:**
Please refer to the “SampleDiagnostics” example code for correctly detecting the layer’s device number and retrieving the diagnostic information mentioned above.

### 4.58.2 DqAdvDnxpSetConfig

**Syntax:**

`int  DqAdvDnxpSetConfig(int hd, int devn, uint32 action, uint32* config)`

**Command:**

IOCTL

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint32 action` | What to set:<br>`DQ_L2_SET_CONFIG - set configuration (Fan off and 24V off) (uint32)`<br>`DQ_L2_SET_LED - set LEDs (uint32)`<br>`DQ_L2_SET_LIMITS - set over and under limits (16*uint32 + 16*uint32)` |
| `uint32 *config` | pointer to the data array |

**Output:**

**None**

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or is not a power layer |

| | |
|---|---|
| `DQ_BAD_PARAMETER` | `CLSize` is not between 1 and `DQ_MAXCLSIZE`, `bData` is NULL, or a channel number in `cl` is invalid |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function sets up following parameters:

`DQ_L4_SET_CONFIG` – this option allows turning off fan 24V DC/DC (powers isolated sides of most analog input and output layers). Use constant DQ_L4_STS_DC24OFF to do that

`DQ_L4_SET_LED` – allows you to turn on/turn off user defined LEDs: (DQ_L4_LED_USR, DQ_L4_LED_IO, DQ_L4_LED_ATT). The rest of LEDs are controlled by the periodic routine that is executed once a second. To shut automatic update off and get compete access to LEDs, use the DQ_L4_LED_STOP_UPDATE flag along with LEDs flags. A write without this flag restores periodic status indication.

```
#define DQ_L4_LED_OVRT      (1L<<0)    // Overtemperature (red)
#define DQ_L4_LED_ATT       (1L<<1)    // Attention (red)
#define DQ_L4_LED_RW        (1L<<2)    // Read/write
#define DQ_L4_LED_USR       (1L<<3)    // User controlled
#define DQ_L4_LED_IO        (1L<<4)   // Communication active
#define DQ_L4_LED_3_3_DNR   (1L<<5)   // DNR side 3.3V
#define DQ_L4_LED_PG        (1L<<6)   // Power good
#define DQ_L4_LED_24_DNR    (1L<<7)    // DNR side 24V

#define DQ_L4_LED_BLINK(N)  ((N<<16)|N) // up and blink
```

`DQ_L4_SET_LIMITS` – sets under and over the limit levels (used if interrupt detection is enabled, currently not supported). Format is straight binary 24-bit, use 32-channel 32-bit array (16 over limit and 16 under limit values).

**Note:**

None

## 4.59 PowerDNA 1PPS / PTP Synchronization

The synchronization interface consists of configuring one or more RACK or Cube chassis and any associated I/O boards to align clocks, triggers and/or timestamps with an external pulse-per-second (PPS) or Precision Time Protocol (PTP) reference.

1PPS synchronization capabilities are implemented on CPU logic 02.12.2D (2017) and later.
IEEE-1588 (PTP) synchronization capabilities are implemented on -02 and -03 versions of UEI Cube and RACK chassis with CPU Logic 02.12.46 (2018) or later.
Future implementations are noted as <Reserved>.

PTP server parameters are configured using the `DqSyncDefinePTP()` API.

1PPS / PTP synchronization options are defined in a sync structure (`DQ_SYNC_SCHEME`) for each chassis in the system. `DQ_SYNC_SCHEME` consists of 24 elements that configure chassis-wide hardware options and is set in hardware with the `DqSyncDefineSyncScheme()` API.

Structure elements are described the section below, and APIs that set up the structure in hardware, as well as other functions that setup clocks, triggers, and I/O boards, are described in the sections that follow. For more information, please refer to the *PowerDNx 1PPS Synchronization Manual*.

### 4.59.1 Overview of DQ_SYNC_SCHEME structure

The `DQ_SYNC_SCHEME` structure is organized into 5 sections:

- *Section A: IOM SYNC Source Configuration*
- *Section B: Master Server Configuration*
- *Section C: Clock Configuration*
- *Section D: Trigger Configuration*
- *Section E: SyncOut Configuration*

```
typedef struct {
    // ==== Section A ================================================================
    // IOM Sync Source Configuration
    uint32 sync_device;      // IOM CPU type (1588-capable 8347S, 8347 -1G Cube/RACK,or 5200 PPCx Cube)
    uint32 sync_source;      // where to get nPPS clock to synchronize system (NULL for PTP)
    uint32 sync_line;        // where to route sync clock (NULL for PTP)
    uint32 sync_mode;        // mode of synchronization
    uint32 nPPS;             // N - number of pulses per second for input nPPS clock
    uint32 nPPS_us;          // Expected accuracy of the nPPS clock in us, clocks outside
                             //   of the range will be ignored, 0=default

    // ==== Section B ================================================================
    // Master Configuration: Identifies & configures the IOM as the nPPS Master server
    //   (enter 0 for all items if IOM is PTP or a slave that receives 1PPS signal externally)
    uint32 sync_server;       // set IOM to generate 1PPS sync pulse (as Master)
    uint32 srv_param;         // set routing of the generated raw 1PPS pulse
                              //   options include internal SYNC bus line
                              //   and external SYNCOUT pin
    uint32 trig_server;       // <Reserved>
// ==== Section C ================================================================
    // Clock Configuration: select clock source for each SYNC line (0 thru 3)
    uint32 clock_src[DQL_SYNC_LINES]; // identify clock source(s) to connect
                                      //  internal SYNC bus lines
    uint32 clock_tmr[DQL_SYNC_LINES]; // <Reserved>
    uint32 clock_frq[DQL_SYNC_LINES]; // set clock frequency for EM
    uint32 clock_div[DQL_SYNC_LINES]; // set clock divider for EMx or PLL
                                      //   (0 == divide by 1, 2 by two, 3 by three etc.)

    // ==== Section D ================================================================
    // Trigger Configuration: identify (or generate) and route trigger signal
    uint32 trig_source;       // identify trigger source(s) to connect to
                              //   internal SYNC bus lines
    uint32 trig_line;         // <reserved>
    uint32 trig_start;        // select mode to start trigger
    uint32 trig_delay;        // set offset of the trigger pulse from nPPS clock
                              //   (microseconds)
    uint32 trig_period_ms;    // <reserved>
    uint32 trig_stop;         // select source for the stop trigger
    uint32 trig_stop_src;     // set stop source for stop trigger upon N-count
    uint32 trig_duration;     // set milliseconds before issuing stop trigger or N-count

    // ==== Section E ================================================================
    // SyncOut Configuration: from SYNC lines or to the outside SyncOut0/1
    uint32 clclk_dest[DQL_SYNC_LINES]; // set where to route CL clock
    uint32 pps_dest;                   // set where to route 1PPS clock to
    uint32 trig_dest;                  // set where to route start/stop trigger

} DQ_SYNC_SCHEME, *pDQ_SYNC_SCHEME;
```

## Detailed description:

***Section A: IOM SYNC Source Configuration***: parameters provide the IOM with synchronization source set up information. All IOMs that are PPS / PTP synchronized, whether they are the master 1PPS server or slave chassis, must initialize parameters in this section.

**Section A `DQ_SYNC_SCHEME` parameters:**

`sync_device` Set this to which chassis CPU type this structure is defining:

- `DQ_SYNC_8347:` standard -1G Cube or RACK chassis (8347 CPU)
- `DQ_SYNC_5200:` standard PPCx Cube (5200 CPU)
- `DQ_SYNC_8347S:` 8347 CPU with IEEE-1588 hardware timestamping (-02 or -03 Cube or RACK)
- `DQ_SYNC_5200S:` <Reserved>

`sync_source` Sets external input pin connected to nPPS pulse.
For -1G chassis (8347 CPU), there are two sync input pins. For PPCx chassis, there is only one option

- `0` for PTP synchronization
- `DQ_SYNCCLK_SYNCIN0:` Uses SyncIn 0 (5200 or 8347)
- `DQ_SYNCCLK_SYNCIN1:` Uses SyncIn 1 (8347 only)

Set `sync_source` to 0 if the chassis is the master 1PPS source and the 1PPS is generated and routed internally (see ***Section B: Master Server Configuration*** for configuring a chassis as the 1PPS master).

| | |
|---|---|
| `sync_line` | Sets internal bus SYNC line that routes the external PPS signal to the ADPLL.<br><br>Set `sync_line` to any of the following:<br>• `0` for PTP synchronization<br>• `DQ_SYNCCLK_SYNC0:` Sync0 line delivers nPPS clock<br>• `DQ_SYNCCLK_SYNC1:` Sync1 line delivers nPPS clock<br>• `DQ_SYNCCLK_SYNC2:` Sync2 line delivers nPPS clock<br>• `DQ_SYNCCLK_SYNC3:` Sync3 line delivers nPPS clock |
| `sync_mode` | Determines how (or which protocol is used) to synchronize the chassis.<br>• `DQ_SYNCCLK_SYNC`: synchronized with PPS pulse<br>• `DQ_SYNCCLK_NTP:` <Reserved><br>• `DQ_SYNCCLK_1588:` synchronized with IEEE-1588 PTP protocol.<br>- Logically OR with `DQ_SYNCCLK_ETH0` or `DQ_SYNCCLK_ETH1` to configure which NIC port transfers PTP packets (i.e., `DQ_SYNCCLK_1588 \| DQ_SYNCCLK_ETH0).`<br>The default is NIC1 (`DQ_SYNCCLK_ETH0).`<br>-Can only be used with `sync_device` set to `DQ_SYNC_8347S` and with -02 or -03 Cube or RACK chassis |
| `nPPS` | Determines the number of pulses per second in the synchronization clock.<br>Set `nPPS` to the number of pulses/sec. (1 is typical – it defines a 1PPS signal) |
| `nPPS_us` | Sets the expected accuracy of the synchronization clock input. The ADPLL uses this to validate the incoming nPPS pulse and ignores pulses outside this range. The ADPLL will maintain an internal nPPS signal if the reference nPPS input is out of the `nPPS_us` range.<br><br>Set `nPPS_us` to the accuracy of the device generating the clock. For example, an nPPS pulse produced by a PowerDNx chassis acting as the 1PPS master can stay within 100 µs accuracy; therefore, `nPPS_us` should be programmed with a value of 100.<br><br>As a general rule, program this number to the jitter value of your clock source + 100 µs. 0 will use the default value for the mode of operation. |

***Section B: Master Server Configuration:*** parameters set the chassis as the 1PPS server (Master server). When the following options are set, the 1PPS synchronization signal is generated internally.

**Section B** `DQ_SYNC_SCHEME` **parameters:**

| | |
|---|---|
| `sync_server` | Set `sync_server` to `DQ_SYNCSRV_1PPS` to configure the chassis to generate 1PPS internally and act as the 1PPS master.<br>Set `sync_server` to 0 to leave chassis as slave or for PTP synchronization. |
| `srv_param` | When `sync_server` is set to `DQ_SYNCSRV_1PPS,` this parameter controls where the internally generated 1PPS signal will be routed.<br>Set `srv_param` to any of the following:<br>• `DQ_SYNCSRV_SYNCOUT0:` Routes to SyncOut0 (5200 or 8347)<br>• `DQ_SYNCSRV_SYNCOUT1:` Routes to SyncOut1 (8347 only)<br>• `0:` set to 0 if chassis is a 1PPS slave or for PTP synchronization<br>Additionally, `srv_param` can be logically ORed with any of the following to route the 1PPS signal to the internal SYNC bus:<br>• `DQ_USE_SYNC0:` Raw 1PPS to SYNC0<br>• `DQ_USE_SYNC1:` Raw 1PPS to SYNC1<br>• `DQ_USE_SYNC2:` Raw 1PPS to SYNC2<br>• `DQ_USE_SYNC3:` Raw 1PPS to SYNC3<br>For example, the following routes the 1PPS externally to SyncOut0 on the 10-pin connector and internally to SYNC line 0:<br>`DQ_SYNCSRV_SYNCOUT0 \| DQ_USE_SYNC0` |
| `trig_server` | Reserved, set to 0. |

***Section C: Clock Configuration*** Clock Configuration defines clock routing via the internal SYNC lines and clock frequency settings.

NOTE: Typically the sync lines are also used to carry an nPPS signal and a trigger. The elements corresponding to these lines should be left as ‘0’.

Each of the parameters in this section is an array of 4 with 1 element mapped to each SYNC line. Array element 0 corresponds to SYNC line 0 while array element 3 corresponds to SYNC line three. This behavior can be overridden by bitwise ORing the DQ_USE_SYNC0 through DQ_USE_SYNC3 define with the values set.

**Section C `DQ_SYNC_SCHEME` parameters:**

`clock_src[4]` Routes a particular clock source to the corresponding SYNC line.

The SYNC line is determined by the element position of the array `{<SyncLine0>, <SyncLine1>, <SyncLine2>, <SyncLine3>}` or alternatively set by bitwise ORing a `DQ_USE_SYNCx` define value with the value set.

- `DQ_CLOCKSRC_EM0:` Clock is synthesized by Event Module. *Use to generate internal clocks that are synchronized to PPS / PTP.*
- `DQ_CLOCKSRC_EM1: EM0` Event Module divider 1 (use `clock_div` as divider)
- `DQ_CLOCKSRC_EM2: EM0` Event Module divider 2 (use `clock_div` as divider)
- `DQ_CLOCKSRC_SYNCIN0:` Clock input to pin SYNCIN0 (5200 or 8347 CPU)
- `DQ_CLOCKSRC_SYNCIN1:` Clock input to pin SYNCIN1 (8347 CPU board)
- `DQ_CLOCKSRC_PLL0:` PLL0 on CPU board (will not be locked to 1PPS)
- `DQ_CLOCKSRC_PLL0TMR0:` PLL0 on CPU board divided by TMR0 (will not be locked to 1PPS)
- `DQ_CLOCKSRC_PLL0TMR1:` PLL0 on CPU board divided by TMR1 (will not be locked to 1PPS)
- `DQ_CLOCKSRC_UNUSED:` Not connected to any of the following clock sources or connected elsewhere (or left unused)
- `<DQ_CLOCKSRC_ADPLL:` Clock generated by ADPLL on CPU board (for debug purposes only)>

NOTE: Only the Event Module and ADPLL are synchronized to the PPS / PTP.

Lines that have previously been set or will be set to something else should be written with 0, (i.e., `DQ_CLOCKSRC_UNUSED`).

For example, to program the EM on SYNC line 2:

- Set `clock_src` to `{0, 0, DQ_CLOCKSRC_EM0, 0}`

| | |
|---|---|
| `clock_tmr[4]` | <Reserved> |
| `clock_freq[4]` | Sets the frequency of the clock specified for each SYNC line.<br>For example, to set an 8 kHz EM clock on SYNC line 2:<br>`clock_src` would be set to `{0, 0, DQ_CLOCKSRC_EM0, 0}` and<br>`clock_freq` would be set to `{0, 0, 8000, 0}` |
| `clock_div[4]` | Used as an 8-bit clock divider when `clock_src` is set as `DQ_CLOCKSRC_EMx.` The maximum Event Module divider is 255.<br>Used as a 32-bit clock divider for the PLL when `clock_src` is set as `DQ_CLOCKSRC_PLL0TMRx.`<br>Set `clock_div` to 0 or 1 to divide by 1 (i.e., leave `clock_freq` as programmed), set to 2 to divide by 2, set to 3 to divide by 3, etc. |

***Section D: Trigger Configuration:*** Trigger Configuration describes the start and stop trigger.

**Section D `DQ_SYNC_SCHEME` parameters:**

| | |
|---|---|
| `trig_source` | Sets which SYNC lines the trigger is routed to.<br>Set `trig_source` to any of the following:<br>• `DQ_USE_SYNC0:` Feed trigger from SYNC0<br>• `DQ_USE_SYNC1:` Feed trigger from SYNC1<br>• `DQ_USE_SYNC2:` Feed trigger from SYNC2<br>• `DQ_USE_SYNC3:` Feed trigger from SYNC3<br>NOTE: To use an externally generated trigger, the `trig_start` parameter is set to `DQ_TRIGSTART_SYNC`; to route an externally generated trigger, `trig_source` must be set to one of the `DQ_USE_SYNCx` lines bitwise OR'ed with one of the following external inputs:<br>• `DQ_TRIGSTART_SYNCIN0:` Delivers trigger via SyncIn 0 (5200 or 8347) (pin 10)<br>• `DQ_TRIGSTART_SYNCIN1:` Delivers trigger via SyncIn 1 (8347 only) (pin 6) |
| `trig_line` | Reserved, set to 0 |
| `trig_start` | Describes the mode of operation of the start trigger. A chassis can be configured to use an externally generated trigger or an internally generated, synchronized trigger:<br>• `DQ_TRIGSTART_SYNC:` Use external start trigger.<br>• `DQ_TRIGSTART_NPPS:` Issue start trigger on the next PPS (plus <`trig_delay`>)<br>NOTE:<br>When using `DQ_TRIGSTART_SYNC`, the chassis routes the external trigger as described by `trig_source`. This means the I/O boards will start on the rising edge of the input signal and stop on the falling edge. This trigger will not be locked with the 1PPS reference.<br>When using `DQ_TRIGSTART_NPPS` the user must issue a command to "arm" the trigger on all the boards. To arm all chassis at once, the broadcast API (`DqSyncTrigOnNextPPSBrCast()` or `DqSyncTrigOnNextPPS())` can be used and once the broadcast message is received, chassis will issue a trigger on the next rising edge of the nPPS signal. |
| `trig_delay` | Used to issue a trigger after a time delay from the nPPS signal (in µs) when `trig_start` is configured as `DQ_TRIGSTART_NPPS`.<br>The minimum value of 0 causes the start trigger to be issued as soon as the nPPS signal is received. The maximum value is 1048575 (0xFFFFF), or approximately 1 second of delay. |

| | |
|---|---|
| `trig_period_ms` | Reserved |
| `trig_stop` | Defines the stop trigger for I/O boards to stop acquiring data:<br>• `DQ_TRIGSTOP_SYNC:` Derive stop trigger from the SYNC line<br>• `DQ_TRIGSTOP_DURATION:` Issue stop trigger after a delay of N ms<br>• `DQ_TRIGSTOP_NCLOCKS:` Issue stop trigger after a certain number of clocks seen (specify which clock with `trig_stop_src)`<br>• `0`: no stop trigger<br><br>`- DQ_TRIGSTOP_SYNC` is used when the start trigger is external. The stop trigger is the falling edge of the external trigger.<br><br>`- DQ_TRIGSTOP_DURATION` causes the chassis to issue an internally generated stop trigger after a programmed number of milliseconds have passed since the start trigger. Time is specified in the `trig_duration` element (in ms) described below.<br><br>`- DQ_TRIGSTOP_NCLOCKS` causes the chassis to issue an internally generated stop trigger once a certain number of user-defined clocks have passed (see `trig_stop_src` and `trig_duration` parameters below). |
| `trig_stop_src` | Used when `trig_stop` = `DQ_TRIGSTOP_NCLOCKS`. This parameter specifies which SYNC line provides the clock source used to determine the number of clocks.<br>• `DQ_USE_SYNC0:` Feed clock from SYNC0<br>• `DQ_USE_SYNC1:` Feed clock from SYNC1<br>• `DQ_USE_SYNC2:` Feed clock from SYNC2<br>• `DQ_USE_SYNC3:` Feed clock from SYNC3 |
| `trig_duration` | Sets when stop trigger will be asserted.<br>Used when `trig_stop` = `DQ_TRIGSTOP_DURATION` or `DQ_TRIGSTOP_NCLOCKS`.<br><br>When `trig_stop` is set as `DQ_TRIGSTOP_DURATION,` `trig_duration` is the time in milliseconds the stop trigger will be issued after the start trigger is detected. The maximum value is 1048575 (0xFFFFF), or approximately 17 minutes of acquisition.<br><br>When `trig_stop` is set as `DQ_TRIGSTOP_NCLOCKS,` `trig_duration` defines a number of clock cycles that will pass until the stop trigger is issued. The maximum value is 1048575 (0xFFFFF), or `DQ_TRIGSTOP_NCLOCKS` * 1/frequency of the `trig_stop_source`. |

***Section E: SyncOut Configuration:*** parameters for SyncOut Configuration describe where to route signals to the SYNC lines or to the outside SyncOut 0/1 pins.

**Section E** `DQ_SYNC_SCHEME` **parameters:**

| | |
|---|---|
| `clclk_dest[4]` | Routes the SYNC line for a particular clock source out via the 10-pin sync connector.<br>Each element takes a SYNC line bitwise ORed with the desired line of the sync connector:<br>• `DQ_USE_SYNC0:` Feed clock from SYNC0<br>• `DQ_USE_SYNC1:` Feed clock from SYNC1<br>• `DQ_USE_SYNC2:` Feed clock from SYNC2<br>• `DQ_USE_SYNC3:` Feed clock from SYNC3<br>Bitwise OR DQ_USE_SYNCx with either of the following:<br>• `DQ_CLKDEST_SYNCOUT0:` Delivers clock via SyncOut 0 (5200 or 8347) (pin 8)<br>• `DQ_CLKDEST_SYNCOUT1:` Delivers clock via SyncOut 1 (8347 only) (pin 4) |
| `pps_dest` | Routes the PPS signal out via the 10-pin sync connector.<br>The SYNC line routing the PPS signal is bitwise ORed with the desired line of the sync connector:<br>• `DQ_USE_SYNC0:` Feed clock from SYNC0<br>• `DQ_USE_SYNC1:` Feed clock from SYNC1<br>• `DQ_USE_SYNC2:` Feed clock from SYNC2<br>• `DQ_USE_SYNC3:` Feed clock from SYNC3<br>Bitwise OR `DQ_USE_SYNCx` with either of the following:<br>• `DQ_nPPSDEST_SYNCOUT0:` Delivers PPS via SyncOut 0 (5200 or 8347) (pin 8)<br>• `DQ_nPPSDEST_SYNCOUT1:` Delivers PPS via SyncOut 1 (8347 only) (pin 4) |
| `trig_dest` | Route the trigger out via the 10-pin sync connector.<br>The SYNC line routing the PPS signal is bitwise ORed with the desired line of the sync connector:<br>• `DQ_USE_SYNC0:` Feed clock from SYNC0<br>• `DQ_USE_SYNC1:` Feed clock from SYNC1<br>• `DQ_USE_SYNC2:` Feed clock from SYNC2<br>• `DQ_USE_SYNC3:` Feed clock from SYNC3<br>Bitwise OR DQ_USE_SYNCx with either of the following:<br>• `DQ_TRGDEST_SYNCOUT0:` Delivers trigger via SyncOut 0 (5200 or 8347) (pin 8)<br>• `DQ_TRGDEST_SYNCOUT1:` Delivers trigger via SyncOut 1 (8347 only) (pin 4) |

### 4.59.2 DqSyncDefineSyncScheme

**Syntax:**

```
int DqSyncDefineSyncScheme(int handle, pDQ_SYNC_SCHEME scheme, uint32*
status)
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `pDQ_SYNC_SCHEME scheme` | Synchronization scheme structure. Refer to previous section for parameters. |
| `uint32 *status` | Status information |

**Output:**

| | |
|---|---|
| `uint32 *status` | Status information. A value of 0 means a passed status. A non-zero value points to the element in the `DQ_SYNC_SCHEME` structure that caused the failure. |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_PARAMETER` | configuration parameters are not correct |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

`DqSyncDefineSyncScheme()`defines what method of synchronization is used, what clocks and triggers are generated, and where they are routed.

**Notes:**

`DqSyncDefineSyncScheme()` must be called after the following setup:

- After chassis I/O boards are initialized with board-specific configuration APIs (e.g., channel list selection, gains, etc)
- After chassis I/O boards have entered the OPS state (otherwise settings can be lost). The chassis transitions into the OPS state after `DqeEnable()` is called in ACB mode or after a `DqRtVmapStart*()` or `DqRtDmapStart()` function is called in VMAP/DMAP modes. Refer to the earlier data collection sections of this manual for more information about ACB/VMAP/DMAP/ASYNC functions.
- After configuring the PTP server if synchronizing to the IEEE-1588 standard (after calling `DqSyncDefinePTPServer()`).

Clocks, triggers, and/or timestamps for the individual I/O boards can be configured with `DqSyncDefineLayerClock()`, `DqSyncDefineLayerTrigger()`, and `DqSyncDefineLayerTimestamp()` after `DqSyncDefineSyncScheme()`is called.

### *4.59.3 DqSyncDefinePTPServer*

**Syntax:**

`int DqSyncDefinePTPServer(int handle, int mode, pDQ_SYNC_DEFPTP pPTP)`

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `int mode` | Reserved |
| `pDQ_SYNC_DEFPTP pPTP` | Structure of PTP interface parameters |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_PARAMETER` | configuration parameters are not correct |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Defines parameters for the PTP server if PTP is enabled on the IOM.

The `pDQ_SYNC_DEFPTP` is defined as follows:

```
typedef struct {
    uint32 cfg;             // bitmask for PTP configuration

    uint8 subdomain;        // PTP subdomain (Domain) number
    uint8 priority1;        // priority of the device
    uint8 priority2;
    int8  logSyncInterval;  // must be 0; log2(period of sync messages)

    int8  logMinDelayRequestInterval; // log2(minimum space between delay
requests)
    int8  logAnnounceInterval; // log2(period of announce msgs)
    uint8 announceTimeout;
    uint8 reserved;
    uint32 reserved[5];

    // nonstandard extensions
    uint32 static_master_ip;
} DQ_SYNC_DEFPTP, *pDQ_SYNC_DEFPTP;
```

| | |
|---|---|
| `uint32 cfg` | Bitmask for PTP configuration:<br>• Set to 0 for normal operation<br>• Set to `DQ_PTP_USE_STATIC_MASTER` to bypass BMCA and select the PTP master directly with `static_master_ip`.<br>*Default is 0.* |
| `uint8 subdomain` | Subdomain field that specifies the set of clocks in a multiple clock distribution system that are capable of synchronizing with each other.<br>*Default is 0.* |
| `uint8 priority1` | User-assigned, preemptive priority to the best master clock algorithm<br>(Smaller numbers indicate higher priority. This is automatically set to 255 when a chassis is configured in slave only mode.) |
| `uint8 priority2` | User-assigned priority2<br>(Smaller numbers indicate higher priority) |
| `int8 logAnnounceInterval` | Log2(period of announce messages)<br>How often the PTP master clock sends Announce messages. |
| `uint8 announceTimeout` | Number of announce intervals allowed to transpire without the slave receiving an Announce message from the master. After this delay, the slave will timeout: “ANNOUNCE_RECEIPT_TIMEOUT_EXPIRES” will occur, and the slave returns to the state of listening for a new master PTP source. |
| `int8 logSyncInterval` | Log2(period of sync messages)<br>How often the PTP master clock sends Sync messages in multicast mode.<br>**Required to be 0**, resulting in 1 second intervals |
| `int8 logMinDelayRequestInterval` | log2(minimum space between delay requests)<br>Minimum interval allowed between PTP delay-request messages.<br>Slave devices extract this from sync packets. (If chassis is master clock, this value must be set). |
| `uint32 static_master_ip` | IP address of master IEEE-1588 device<br>(will be set and reset automatically using the best-master clock algorithm)<br>Can be configured for debug. |

**Notes:**

Supported on CPU Logic version 12.46 and later. Call after I/O boards are in the Operation (OPS) state and before calling `DqSyncDefineSyncScheme()`.

### 4.59.4 DqSyncDefineLayerClock

**Syntax:**

`int DqSyncDefineLayerClock(int Iom, int devn, pDQ_SYNC_DEF_CLOCKS clocks);`

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Board number inside the IOM |
| `pDQ_SYNC_DEF_CLOCKS clocks` | Clock settings. See **Notes** below. |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by devn does not exist |
| `DQ_BAD_PARAMETER` | configuration parameters are not correct |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets clock source for I/O boards. This is a "wrapper" function that allows users to set clocking for any type of I/O board and hides differences in implementation.

**Notes:**

`DqSyncDefineLayerClock()` must be called after `DqSyncDefineSyncScheme()` is called.

The `DQ_SYNC_DEF_CLOCKS` structure contains the following parameters:

```
typedef struct {
    int clk_line;   // clock line to use. See mappings below
    int divider;    // 0=original clock,
                    //  1 thru n = divide clock by n
    int grp_delay;  // group delay in samples, -1 = auto define
    uint32 flags;   // flag to change mode of operation:
} DQ_SYNC_DEF_CLOCKS, *pDQ_SYNC_DEF_CLOCKS;
```

**`clk_line`:**

Supported `#define` values for `clk_line` parameter:

| | |
|---|---|
| `DQ_SYNCLCLK_SYNC0` | Use clock on SYNC0 line on internal sync bus |
| `DQ_SYNCCLLK_SYNC1` | Use clock on SYNC1 line on internal sync bus |

| | |
|---|---|
| `DQ_SYNCCLLK_SYNC2` | Use clock on SYNC2 line on internal sync bus |
| `DQ_SYNCCLLK_SYNC3` | Use clock on SYNC3 line on internal sync bus |
| `DQ_SYNCCLLK_SYNCIN0` | <Reserved> |
| `DQ_SYNCCLLK_SYNCIN1` | <Reserved> |
| `DQ_SYNCCLLK_DEFAULT` | Board programmed (disable using chassis-wide sync) |

**`divider`:**

Each I/O board has the capability of dividing down the incoming clock by an integer. Set `divider` to 0 or 1 to use the incoming clock supplied on `clk_line.` Set `divider` to an integer n value to divide the incoming clock by n, (i.e., if the incoming clock is 1000 Hz, setting `divider` to 10 will supply a 100 Hz clock for the I/O board.

**`grp_delay`:**

The `grp_delay` parameter represents the group delay in samples of AI board. The group delay is the number of input samples that need to feed through an FIR filter before the output samples represent filtered input samples. It can be thought of as an initialization period.

Setting this parameter to -1 allows the software to program the default group delay associated with the particular AI board you are programming. Allowing the software to compensate for the group delay will guarantee timestamp alignment. The software holds off incrementing the timestamp until the first filtered input sample is ready for output.

Default group delays for each AI board are as follows:

| Board | UEI FIR Group Delay | HW Related Delay | Total Group Delay |
|---|---|---|---|
| AI-205 | 64 samples | 0 samples | 64 samples |
| AI-211 | 127 samples | 38 samples | 165 samples |
| AI-217 | 64 samples | 38 samples | 102 samples |
| AI-218 | 64 samples<br>*when sample rate > 3.75 kHz* | 38 samples | 102 samples<br>*@ SR > 3.75 kHz* |
| | 128 samples<br>*when sample rate ≤ 3.75 kHz* | 38 samples | 166 samples<br>*@ SR ≤ 3.75 kHz* |
| AI-228 | 64 samples<br>when sample rate > 3.75 kHz | 38 samples | 102 samples<br>@ SR > 3.75 kHz |
| | 128 samples<br>when sample rate ≤ 3.75 kHz | 38 samples | 166 samples<br>@ SR ≤ 3.75 kHz |

Alternatively, users can program the group delay with a value of their choice, but timestamp alignment will not be compensated for. Not compensating for the group delay will result in seeing pre-trigger samples/missing scans in output data.

**`flags`:**

The **`flags`** parameter allows users to change the clocking operations of an I/O board.

- `0`: default operation
- `DQ_SYNCLCLK_CLOCK_PER_SCAN:` enables analog input boards to use a single clock per scan mode, which provides a sample on each clock.
  For example, `DQ_SYNCLCLK_CLOCK_PER_SCAN` allows AI boards with an 8x decimation rate to use a clock at the 1x rate instead of the 8x rate, (examples of boards that run at an 8x rate are AI-211, AI-217, AI-218, AI-228).
- `DQ_SYNCLCLK_ISSUE_IMMEDIATE:` configures clock dividers to start issuing clocks immediately after the trigger occurs. Otherwise, the first clock is issued once the divider counts to zero after the trigger.
  For example, if the sample rate is 8 kHz, when the `DQ_SYNCLCLK_ISSUE_IMMEDIATE` is set, the first 8 kHz clock to the I/O board will issue immediately after the trigger; otherwise, if `DQ_SYNCLCLK_ISSUE_IMMEDIATE` is not set, the first 8 kHz clock to the I/O board is issued approximately 125 microseconds (1 8 kHz cycle) after the trigger.

### *4.59.5 DqSyncDefineLayerTrigger*

**Syntax:**

```
int DqSyncDefineLayerTrigger(int Iom, int devn, int trig_line, int mode);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Board number inside the IOM |
| `int trig_line` | Hardware source for trigger |
| `int mode` | Reserved, set to 0 |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_BAD_PARAMETER` | configuration parameters are not correct |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets trigger sources for I/O boards. This is a "wrapper" function that allows users to set triggering for any type of I/O board and hides differences in implementation.

**Notes:**

`DqSyncDefineLayerTrigger()` must be called after `DqSyncDefineSyncScheme()` is called.

Supported `#define` values for `trig_line` parameter:

| | |
|---|---|
| `DQ_SYNCTRG_SYNC0` | Use trigger on SYNC0 line on internal sync bus |
| `DQ_SYNCTRG_SYNC1` | Use trigger on SYNC1 line on internal sync bus |
| `DQ_SYNCTRG_SYNC2` | Use trigger on SYNC2 line on internal sync bus |
| `DQ_SYNCTRG_SYNC3` | Use trigger on SYNC3 line on internal sync bus |
| `DQ_SYNCTRG_SYNCIN0` | <Reserved> |
| `DQ_SYNCTRG_SYNCIN1` | <Reserved> |
| `DQ_SYNCTRG_SOFT` | Use software trigger |
| `DQ_SYNCTRG_DEFAULT` | Board programmed (disable using chassis-wide sync) |

### *4.59.6 DqSyncDefineLayerTimestamp*

**Syntax:**

```
int DqSyncDefineLayerTimestamp(int Iom, int devn, int trig_line, int mode);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Board number inside the IOM |
| `int trig_line` | Hardware clock source for timestamp |
| `int mode` | Reserved, set to 0 |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_BAD_PARAMETER` | configuration parameters are not correct |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sets the timestamp clock source for I/O boards. This is a "wrapper" function that allows users to set the clock source to timestamping for any type of I/O board and hides differences in implementation.

**Notes:**

`DqSyncDefineLayerTimestamp()` must be called after `DqSyncDefineSyncScheme()` is called.

Supported `#define` values for `line` parameter:

| | |
|---|---|
| `DQ_SYNCTST_SYNC0` | Use SYNC0 line on internal sync bus as trigger clock |
| `DQ_SYNCTST_SYNC1` | Use SYNC1 line on internal sync bus as trigger clock |
| `DQ_SYNCTST_SYNC2` | Use SYNC2 line on internal sync bus as trigger clock |
| `DQ_SYNCTST_SYNC3` | Use SYNC3 line on internal sync bus as trigger clock |
| `DQ_SYNCTST_SYNCIN0` | <Reserved> |
| `DQ_SYNCTST_SYNCIN1` | <Reserved> |
| `DQ_SYNCTST_DEFAULT` | Board programmed (disable using chassis-wide sync) |

### *4.59.7 DqSyncGetUTCTimeFromPTP*

**Syntax:**

```
int DqSyncGetUTCTimeFromPTP(int handle, int mode, pDQ_SYNC_UTC_TIME ptpUTC)
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `int mode` | Reserved, set to 0 |
| `pDQ_SYNC_UTC_TIME ptpUTC` | Structure holding the PTP time |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_BAD_PARAMETER` | configuration parameters are not correct |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function returns the current UTC time from the PTP master.

**Notes:**

The structure `pDQ_SYNC_UTC_TIME ptpUTC` contains the following fields:

```
typedef struct {
   uint32 reserved;  // reserved
   uint32 sec;       // PTP time in seconds
   uint32 nsec;      // PTP time nanoseconds
   uint32 timestamp; // timestamp from the CPU layer
                     //   to the time above
   uint32 flags;     // status flags relating to the returned time
} DQ_SYNC_UTC_TIME, *pDQ_SYNC_UTC_TIME;
```

`ptpUTC->flags` will have the bit `DQ_SYNC_UTCTM_TIMEVALID` set if the returned time is valid. Additional `flags` bits can alert the user that the time may not actually be UTC time:

- `DQ_SYNC_UTCTM_ARB_PTPTSCALE:` The PTP master is using an arbitrary timescale.
- `DQ_SYNC_UTCTM_OFFSINVLAID:` This isn’t a valid UTC offset from the master.
- `DQ_SYNC_UTCTM_TIMEVALID:` Time is known. Either we are master or PTP slave.

### 4.59.8 DqSyncGetSyncStatus

**Syntax:**
```
int DAQLIB DqSyncGetSyncStatus(int Iom, int mode, pDQ_SYNC_STATUS status);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `int mode` | Reserved, set to 0 |
| `pDQ_SYNC_STATUS status` | Status of sync interface |

**Output:**

| | |
|---|---|
| `pDQ_SYNC_STATUS status` | Status of sync interface |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by devn does not exist |
| `DQ_BAD_PARAMETER` | configuration parameters are not correct |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Returns current status of synchronization scheme for chassis.

**Notes:**

The `DQ_SYNC_STATUS` structure consists of the following elements (descriptions are on the following page):

```
typedef struct {
    DQ_SYNC_ADPLL_STAT adpll_sts; // ADPLL status
    uint32 pll_stat[0];           // <reserved>
    uint32 pps_status;            // <reserved>
    uint32 gps_irig;              // <reserved>
    uint32 evm_stat;              // event module status
    uint32 sync_snap;             // snapshot of the SYNC lines
    uint32 sync_conn;             // snapshot of SyncIn/SyncOut lines
    uint32 reserved[4];           // reserved
    uint32 time_since_pps;        // how long since the last PPS
} DQ_SYNC_STATUS, *pDQ_SYNC_STATUS;
```

The following table provides bit descriptions of DQ_SYNC_STATUS elements:

| | | | |
|---|---|---|---|
| `adpll_sts` | Structure of type DQ_SYNC_ADPLL_STAT providing access to all ADPLL status information | See DQ_SYNC_ADPLL_STAT descriptions and bitmapping following this table. | |
| `pll_stat[0]` | Reserved, PLL lock status | N/A | |
| `pps_status` | Reserved, nPPS status | N/A | |
| `gps_irig` | Reserved, GPS/IRIG receiver status | N/A | |
| `evm_stat` | Status of Event Module | 28 | EVTMOD_STS_DPLLS: Reads (1) if the Event Module generated the correct number of clocks |
| | | 23:0 | Event counter: Specific count values are for reserved use; however, note that a non-incrementing counter can indicate an error in your 1PPS source or sync structure configuration. |
| `sync_snap` | Snapshot of SYNC lines on the internal SYNC bus. | 31:24 | <Reserved> |
| | | 23 | Current state of SYNC3 |
| | | 22 | Current state of SYNC2 |
| | | 21 | Current state of SYNC1 |
| | | 20 | Current state of SYNC0 |
| | | 19:0 | <Reserved> |
| `sync_conn` | Snapshot of the external SyncIn and SyncOut lines | 31:0 | <Reserved> |
| `reserved` | <Reserved> | N/A | |
| `time_since_pps` | Number of 66 MHz clocks since last 1PPS | 31:0 Number of cycles | |

The `DQ_SYNC_ADPLL_STAT` structure is the first element in the `DQ_SYNC_STATUS` structure and provides the following ADPLL status settings (descriptions are on the following page):

```
typedef struct {
    uint32 status;  // ADPLL status register
    uint32 min_per; // Minimum period length for the input clock
    uint32 avg_per; // Averaged detected length of the VALIDATED
                    //  input period
    uint32 max_per; // Maximum period length for the input clock
    uint32 lst_per; // Measured length of the last input period
    uint32 acc_err; // Accumulated pulse position error in system
                    //  clocks
} DQ_SYNC_ADPLL_STAT, *p DQ_SYNC_ADPLL_STAT;
```

Bit mapping of the status registers included in `DQ_SYNC_ADPLL_STAT` are:

| | | | |
|---|---|---|---|
| `status` | ADPLL status register.<br>*See **NOTE** on next page for `RESYNC`, `AV`, `CV`, and `CE` descriptions. | 31:17<br>16<br>15:3<br>2<br>1<br>0 | <Reserved><br>RESYNC*<br><Reserved><br>AV*<br>CV*<br>CE* |
| `min_per` | Minimum period length for the ADPLL input clock | 31:27<br>26:0 | <Reserved><br>Minimum period |
| `avg_per` | Measured average period of the ADPLL detected and validated input clock | 31:27<br>26:0 | <Reserved><br>Measured average period |
| `max_per` | Maximum period length for the ADPLL input clock | 31:27<br>26:0 | <Reserved><br>Maximum period |
| `lst_per` | Measured length of the last ADPLL input period | 31:27<br>26:0 | <Reserved><br>Measured last period |
| `acc_err` | Accumulated pulse position error between the ADPLL output clock and the input clock.<br>Sign differences between the last measured error and the accumulated position error result in the accumulated error zeroing out. | 31:27<br>26:0 | <Reserved><br>Accumulation error |

**NOTE**: The following are bit descriptions of the ADPLL `status` register:

| | |
|---|---|
| **`RESYNC:`** | Reads (1) when ADPLL and external clock are re-synchronized. This bit will be 1 after initial synchronization of the ADPLL and clock source. If it is set during normal ADPLL operation, a 1 indicates that the synchronization with the source clock was lost (and an error grew to > ¼ of the external clock period). Reading a 1 on `RESYNC` may indicate a drifting or unstable clock source. `RESYNC` is a sticky bit, auto-cleared. Reset state is 0. |
| **`AV:`** | Reads (1) when the moving average passes validation. Invalidated input clocks will not affect the moving average. Reset state is 0. |
| **`CV:`** | Reads (1) when the last input clock period passed validation. Reset state is 0. |
| **`CE:`** | Reads (1) when the last input clock period is too long. Reset state is 0.<br>This bit will read 1 if the 1PPS has been lost. |

### 4.59.9 DqSyncGetPTPStatus

**Syntax:**

```
int DqSyncGetPTPStatus(int handle, int mode, pDQ_SYNC_PTP_STAT pPTPstat);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `int mode` | Reserved, set to 0 |
| `pDQ_SYNC_PTP_STAT pPTPstat` | Structure holding the PTP status information |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_BAD_PARAMETER` | configuration parameters are not correct |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Defines and provides status parameters for the PTP server if PTP is enabled on the IOM.

**Notes:**

The structure `DQ_SYNC_PTP_STAT pPTPstat` contains the following fields:

```
typedef struct {
 uint32 reserved;
  uint32 state;              // Current PTP state (listed below)

  uint64 grandMasterClockID; // PTP clock ID of the grand master
                             //   of the system
  uint64 masterClockID;      // PTP clock ID of the current master.

  uint32 stepsFromGrandMaster;  // Value of the PTP data set steps
  uint32 grandMasterClockClass; // PTP grand master clock class taken
                                //   from the master's announce
                                //   messages

  uint32 reserved0[2];
  int32 meanPathDelay;       // Current calculated mean path delay
  int32 lastMeasuredOffset;  // Last calculated time offset from
                             //   master
  int32 maxMeasuredOffset;   // Maximum calculated time offset from
                             //   master
```

```
    int32 minMeasuredOffset;   // Minimum calculated time offset from
                               //   master
    int32 avgMeasuredOffset;   // Average calculated time offset from
                               //   master
    uint32 reserved1[8];

    // packet statistics
    uint32 totalPkts;          // Total PTP packets received on the
                               //   domain (multicast and for us)
    uint32 announceRcvd;        // The following parameters are counters
                               //   of packets sent or accepted by type
    uint32 announceSnt;         // The following counters won't increment if
                               //    a message is rejected
      uint32 syncRcvd;          //    because the IOM is in the wrong state.
      uint32 syncSnt;
      uint32 followUpRcvd;
      uint32 followUpSnt;
      uint32 delyReqRcvd;
      uint32 delyReqSnt;
      uint32 delyRspRcvd;
      uint32 delyRspSnt;
      uint32 signalingRcvd;
      uint32 signalingSnt;
} DQ_SYNC_PTP_STAT, *pDQ_SYNC_PTP_STAT;
```

The following are defined values for the port `state`:

```
      // defines for port state
      #define DQ_PTP_PORT_STATE_INIT         (1)
      #define DQ_PTP_PORT_STATE_FAULTY       (2)
      #define DQ_PTP_PORT_STATE_DISABLED     (3)
      #define DQ_PTP_PORT_STATE_LISTENING    (4)
      #define DQ_PTP_PORT_STATE_PRE_MASTER   (5)
      #define DQ_PTP_PORT_STATE_MASTER       (6)
      #define DQ_PTP_PORT_STATE_PASSIVE      (7)
      #define DQ_PTP_PORT_STATE_UNCALIBRATED (8)
      #define DQ_PTP_PORT_STATE_SLAVE        (9)
```

### 4.59.10 DqSyncTrigOnNextPPSBrCast

**Syntax:**

```
    int DqSyncTrigOnNextPPSBrCast(int handle, int nIOM, uint32 reserved,
int* handle_arr)
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `int nIOM` | Number of IOMs to trigger |
| `int reserved` | Reserved, set to 0 |
| `int* handle_arr` | List of handles of IOMs to broadcast to |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_PARAMETER` | configuration parameters are not correct |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sends a broadcast UDP command to trigger on the next PPS that is received by IOMs identified in the `handle_arr` array.

**Notes:**

none.

### 4.59.11 DqSyncTrigOnNextPPS

**Syntax:**

```
int DqSyncTrigOnNextPPS(int handle, uint32 line);
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `uint32 line` | <Reserved> |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_PARAMETER` | configuration parameters are not correct |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Trigger the specified IOM on the next PPS.

**Notes:**

none.

### *4.59.12 DqCmdResetTimestampBrCast*

**Syntax:**

```
int DqCmdResetTimestampBrCast(int handle, uint32 timestamp)
```

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `uint32 timestamp` | Value to all IOM timestamps to |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_PARAMETER` | configuration parameters are not correct |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Sends a broadcast UDP command to set/reset timestamp to `timestamp` value. Received by all IOMs.

**Notes:**

When using 1PPS sync, timestamps will automatically reset upon a trigger. This feature is implemented on CPU logic 02.12.2A and later.

### *4.59.13 DqSyncDisableSyncScheme*

**Syntax:**

`int DqSyncDisableSyncScheme(int handle, uint32* status)`

**Input:**

| | |
|---|---|
| `int handle` | Handle to the IOM received from `DqOpenIOM()` |
| `uint32* status` | Reserved |

**Output:**

`none`

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_PARAMETER` | configuration parameters are not correct |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

Disables synchronization scheme on Cube or RACK system:

- Disconnects SYNC lines
- Disconnects signals from sync connector
- Stops clocks in the Event Module/ADPLL
- Stops PTP handling I/O

**Notes:**

### 4.59.14 DqSyncConfigEvents

**Syntax:**

`int DAQLIB DqSyncConfigEvents(int hd, int xtra_prm, int flags, evsync_t event, uint32 param)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | handle to the IOM received from `DqOpenIOM()` |
| `int xtra_prm` | <Reserved> |
| `int flags` | <Reserved> |
| `evsync_t event` | type of event to monitor |
| `uint32 param` | additional parameters |

**Output:**

None

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_PARAMETER` | invalid parameter |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function configures synchronization events on the CPU board.

`evsync_t event` is defined as:

```
typedef enum {
    EVSYNC_CLEAR = 0x1000, // clear all events

    // sync events
    EVSYNC_1PPS  = 0x101  // 1PPS clock from ADPLL event
} evsync_t;
```

To clear all events use the following call:

```
ret = DqSyncConfigEvents(a_handle, 0, 0, EVSYNC_CLEAR, 0);
```

It clears all events that have previously been set along with runtime variables and accumulated events.

When an event happens (i.e., for `EVSYNC_1PPS` the event occurs on the rising edge of the 1PPS signal), the firmware sends back a packet of the following structure:

```
/* DQCMD_EVENT structure */
typedef struct {
    uint8   dev;                    // device (CPU board: 0xE)
    uint8   ss;                     // subsystem (SS0_IN)
    uint16  size;                   // data size
    uint32  event;                  // event type = any of EVSYNC* listed
    uint8   data[];                 // data
} DQEVENT, *pDQEVENT;
```

Where:

`<size>` field contains sizeof(`EVSYNC_ID`).

`<event>` contains the type of the event, e.g. `EVSYNC_1PPS`.

`<data[]>` if `DQEVENT` contains `EVSYNC*` flexible structure:

```
// Event data for sync layer
typedef struct {
    uint32 chan;                    // channel information
    uint32 evtype;                  // type of the event
    uint32 evtmask;                 // subtype of reported events (mask)
    uint32 status;                  // event status register
    uint32 tstamp;                  // timestamp of event taken from
                                    //    layer TS generator
    uint32 size;                    // size of the following data in bytes
    uint32 avail;                   // number of bytes available
    uint32 data[];                  // data to follow
                                    //   (byte pointer –
                                    //    ntohx() and convert properly)
} EVSYNC_ID, *pEVSYNC_ID;
```

Where DQEVENT contains

| | |
|---|---|
| `EVSYNC_1PPS:` | the event data[] consists of a uint32 timestamp stored on the rising edge of the 1PPS |

**Note:**

Note that the `DqCmdReceiveEvent()` API is used to wait for the event packets and that while DQEVENT is properly handled by `DqCmdReceiveEvent()` in terms of converting data fields from big endian (network data representation) to little endian if used with Intel-based host computer `EVSYNC_ID` needs to be converted by the user. The following code can be used:

```
// shuffle bytes big endian->little endian in place
void ntoh_pEvSync(pEVSYNC_ID pEvSYNC) {
    uint32 i;

    pEvSYNC->chan = ntohl(pEvSYNC->chan);
    pEvSYNC->evtype = ntohl(pEvSYNC->evtype);
    pEvSYNC->size = ntohl(pEvSYNC->size);
    pEvSYNC->tstamp = ntohl(pEvSYNC->tstamp);
    for (i = 0; i < pEvSYNC->size/sizeof(uint32); i++)
        pEvSYNC->data[i] = ntohl(pEvSYNC->data[i]);
    return;
}
```

The reason for that is that `DqCmdReceiveEvent()` returns all different types of layer-specific events and as implemented doesn’t do this conversion.

## *4.60 PowerDNA simplified layer signaling*

### *4.60.1 DqAdvRouteClockIn*

**Syntax:**

```
int DqAdvRouteClockIn(int hd, int devn, int line)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int line (all layers)` | `DQ_EXT_SYNC1`<br>`DQ_EXT_SYNC3`<br>`DQ_EXT_INT0`<br>`0 to release` |
| `int line (counter-timer)` | `DQ_EXT_SYNCx`<br>`0 to release` |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_BAD_PARAMETER` | `line` has a value other than the appropriate constants listed above |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function selects the clock source for the layer.

**Note:**

Clock comes from *Ext1* line on the layer (ClockIn). Our intent is if this line is not available and trigger does not use *Ext0* line to use *Ext0* line

Clock can be fed to the counter-timer inputs. Counter-timer 0 can received clock from SYNC0, etc.

### *4.60.2 DqAdvRouteClockOut*

**Syntax:**

```
int DqAdvRouteClockOut(int hd, int devn, int line)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int line (all layers)` | `DQ_EXT_SYNC1`<br>`DQ_EXT_SYNC3`<br>`0 to release` |

| | |
|---|---|
| `int line (counter-timer)` | `DQ_EXT_SYNCx`<br>`0 to release` |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_BAD_PARAMETER` | `line` has a value other than the appropriate constants listed above |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function feeds Clock Output from the layer to Sync1 or Sync3 line.

**Note:**

When using a CT-601 layer, the clock output of Counter-timer0 may be routed to any sync line with this function. To route the output of Counter-timer1, use the DqAdvRouteTrigOut() function.

### *4.60.3 DqAdvRoutePll*

**Syntax:**

```
int DqAdvRoutePll(int hd, double frequency, double* f_actual, int
line)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `double frequency` | Desired frequency |
| `double* f_actual` | Actual frequency |
| `int line` | Line to assign. Use one of the following:<br>*DQ_EXT_SYNC1*<br>*DQ_EXT_SYNC3*<br>*DQ_EXT_SYNC1* \| *DQ_EXT_IMMEDIATE*<br>*DQ_EXT_SYNC3* \| *DQ_EXT_IMMEDIATE*<br>*0* to release |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_PARAMETER` | `line` has a value other than the appropriate constants listed above |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |

| | |
|---|---|
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function programs and routes PLL clock output to one of the SYNCX lines. If the optional DQ_EXT_IMMEDIATE flag is used the PLL will be programmed and immediately connected to the SYNCX line when the DqCmdSetSyncRt() command is issued.

**Note:**

### *4.60.4 DqAdvRouteSyncIn*

**Syntax:**

```
int DqAdvRouteSyncIn(int hd, int line)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int line` | Line to assign. Use one of the following:<br>*DQ_EXT_SYNC0*<br>*DQ_EXT_SYNC1*<br>`0` to release |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_PARAMETER` | `line` has a value other than the appropriate constants listed above |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function routes SyncIn connector to one of the SYNCX lines. Call DqCmdSetSyncRt() after calling this function to make connection.

**Note:** For PPC cubes only.

### *4.60.5 DqAdvRouteSyncOut*

**Syntax:**

```
int DqAdvRouteSyncOut(int hd, int line)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int line` | Line to assign. Use one of the following:<br>*DQ_EXT_SYNC0*<br>*DQ_EXT_SYNC1*<br>*DQ_EXT_SYNC2*<br>*DQ_EXT_SYNC3*<br>`0` to release |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_PARAMETER` | `line` has a value other than the appropriate constants listed above |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function routes one of the SYNCX lines to the SyncOut connector . Call DqCmdSetSyncRt() after calling this function to make connection.

**Note:** For PPC cubes only.

### *4.60.6 DqAdvRouteSyncClockIn*

**Syntax:**

```
int DqAdvRouteSyncClockIn(int hd, int line)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int line` | Line to assign. Use one of the following:<br>*DQ_EXT_SYNC0*<br>*DQ_EXT_SYNC1*<br>*DQ_EXT_SYNC2*<br>*DQ_EXT_SYNC3*<br>*0* to release |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_PARAMETER` | `line` has a value other than the appropriate constants listed above |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function routes the clock input on the Sync connector to one of the SYNCX lines. Call DqCmdSetSyncRt() after calling this function to make connection.

**Note:** For DNR racks and 1GB cubes only.

### *4.60.7 DqAdvRouteSyncClockOut*

**Syntax:**

`int DqAdvRouteSyncClockOut(int hd, int line)`

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int line` | Signal to assign. Use one of the following:<br>*DQ_EXT_SYNC0*<br>*DQ_EXT_SYNC1*<br>*DQ_EXT_SYNC2*<br>*DQ_EXT_SYNC3*<br>*DQ_EXT_GPIO_LOGIC0*<br>*DQ_EXT_GPIO_LOGIC1*<br>*DQ_EXT_PUSH_BUTTON* (see **Note**)<br>*0* to release |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_PARAMETER` | `line` has a value other than the appropriate constants listed above |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function routes one of the selected signals (SYNCX lines, logic states or pushbutton) to the clock output on the Sync connector . Call DqCmdSetSyncRt() after calling this function to make connection.

**Note:** For DNR racks and 1GB cubes only.

Also note, setting `line` to `DQ_EXT_PUSH_BUTTON` uses the state of the Reset pushbutton located at the front of the chassis as the state of ClockOut on the sync connector. When Reset is used as ClockOut, the pushbutton reset functionality will still be implemented: Momentary presses of Reset can show as a ClockOut state change; however, pressing the button for 3 seconds or longer will reset the chassis.

### *4.60.8 DqAdvRouteSyncTrigIn*

**Syntax:**

```
int DqAdvRouteSyncTrigIn(int hd, int line)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int line` | Line to assign. Use one of the following:<br>*DQ_EXT_SYNC0*<br>*DQ_EXT_SYNC1*<br>*DQ_EXT_SYNC2*<br>*0* to release |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_PARAMETER` | `line` has a value other than the appropriate constants listed above |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function routes the trigger input line on the Sync connector to one of the SYNCX lines. Call DqCmdSetSyncRt() after calling this function to make connection.

**Note:** For DNR racks and 1GB cubes only.

### *4.60.9 DqAdvRouteSyncTrigOut*

**Syntax:**

```
int DqAdvRouteSyncTrigOut(int hd, int line)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int line` | Signal to assign. Use one of the following:<br>*DQ_EXT_SYNC0*<br>*DQ_EXT_SYNC1*<br>*DQ_EXT_SYNC2*<br>*DQ_EXT_SYNC3*<br>*DQ_EXT_GPIO_LOGIC0*<br>*DQ_EXT_GPIO_LOGIC1*<br>*DQ_EXT_PUSH_BUTTON* (see **Note**)<br>*0* to release |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_PARAMETER` | `line` has a value other than the appropriate constants listed above |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function routes one of the selected signals (SYNCX lines, logic states or pushbutton) to the trigger output of the Sync connector . Call DqCmdSetSyncRt() after calling this function to make connection.

**Note:** For DNR racks and 1GB cubes only.

Also note, setting `line` to `DQ_EXT_PUSH_BUTTON` uses the state of the Reset pushbutton located at the front of the chassis as the state of TrigOut on the sync connector. When Reset is used as a pushbutton trigger, reset functionality will still be implemented: Momentary presses of Reset can be used as a start trigger; however, pressing the button for 3 seconds or longer will reset the chassis.

### *4.60.10 DqAdvRouteTrigIn*

**Syntax:**

```
int DqAdvRouteTrigIn(int hd, int devn, int line)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int32 line` | `DQ_EXT_SYNC0`<br>`DQ_EXT_SYNC2`<br>`DQ_EXT_INT0` trigger source<br>`0` to release |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_BAD_PARAMETER` | `line` has a value other than the appropriate constants listed above |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function routes one of the SyncX lines as a Trigger Input for the layer (or selects external trigger).

**Note:**

### *4.60.11 DqAdvRouteTrigOut*

**Syntax:**

```
int DqAdvRouteTrigOut(int hd, int devn, int line)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int32 line (all layers)` | `DQ_EXT_SYNC2` trigger destination<br>`0` to release |
| `int line (counter-timer)` | `DQ_EXT_SYNCx`<br>`0` to release |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_BAD_PARAMETER` | `line` has a value other than the appropriate constants listed above |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function feeds Trigger Output from the layer to Sync2 line.

**Note:**

When using a CT-601 layer, the clock output of Counter-timer1 may be routed to any sync line with this function. To route the output of Counter-timer0, use the DqAdvRouteClockOut() function.

Like all commands in this series, call DqCmdSetSyncRt() after calling this function to make connection.

### *4.60.12 DqCmdSetSyncRt*

**Syntax:**

```
int DqCmdSetSyncRt(int hd, pSYNCPLL pSyncPll, pSYNCRT pSyncRt)
```

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `pSYNCPLL pSyncPll` | Pointer to the Pll interface specification. If this parameter is NULL, the function will use the default specification produced by the DqAdvRoutePll() function. |
| `pSYNCRT pSyncRt` | Pointer to the synchronization interface specification. If this parameter is NULL, the function will use the default specification automatically setup by the DqAdvRoute...() series of functions. |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn’t established |
| `DQ_BAD_PARAMETER` | `pSyncPll` or `pSyncRt` has an illegal value |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This command sets the sync and pll interface when using the simplified DqAdvRoute... command series. Run this command after all DqAdvRoute... commands are complete.

**Note:**

### *4.60.13 DqAdvLayerAccessDio*

**Syntax:**

```
int DqAdvLayerAccessDio(int hd, int devn, uint16 config, uint16
d_out, uint32 *d_in)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to the IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `uint16 config` | select bits to enable and their direction |
| `uint16 d_out` | data to be sent to dio output pins |

**Output:**

| | |
|---|---|
| `uint32 *d_in` | dio state |

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist or the layer does not support this operation |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

For use with layers AI-201, AI-202, AI-204, AI-225, AO-308 and DIO-403.
This function is used to provide access to the dio pins on db connector.

Configuration: The 'config' value enables the DIO lines and sets their direction ( in or out). Set config by logically ORing the following defines:

```
#define DQ_ACCESS_DIO_DIO0_ENB      0x1
 #define DQ_ACCESS_DIO_DIO1_ENB      0x2
 #define DQ_ACCESS_DIO_DIO2_ENB      0x4
 #define DQ_ACCESS_DIO_DIO3_ENB      0x8
 #define DQ_ACCESS_DIO_DIO0_OUT      0x10
 #define DQ_ACCESS_DIO_DIO0_IN       0       <-- default setting
 #define DQ_ACCESS_DIO_DIO1_OUT      0x20
 #define DQ_ACCESS_DIO_DIO1_IN       0       <-- default setting
 #define DQ_ACCESS_DIO_DIO2_OUT      0x40    <-- default setting
 #define DQ_ACCESS_DIO_DIO2_IN       0
 #define DQ_ACCESS_DIO_DIO3_OUT      0x80
 #define DQ_ACCESS_DIO_DIO3_IN       0       <-- default setting
```

Sending data: The data is sent using 'd_out' . Bit0 contains value for dio0, bit1 contains value for dio1, etc

Receiving data: The d_in value is always returned. Bit0 contains value for dio0, bit1 contains value for dio1, etc

## *4.61 PowerDNA layer signaling*

### *4.61.1 DqAdvSetClockSource*

**Syntax:**

```
int DqAdvSetClockSource(int hd, int devn, uint32 clock, uint32
source, uint32 edge)
```

**Input:**

| | | |
|---|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` | |
| `int devn` | Layer inside the IOM | |
| `uint32 clock` | `DQ_EXT_CLOUT` | command output (*CL* start) |
| | `DQ_EXT_CVOUT` | command output (*CV* start) |
| | `DQ_EXT_CLIN` | command input (*CL* start) |
| | `DQ_EXT_CVIN` | command input (*CV* start) |
| `uint32 source` | `DQ_EXT_CLKIN` | *EXT0* (*DIO0*, default setting) |
| | `DQ_EXT_EXT0` | *EXT0* line (source needs to be selected) |
| | `DQ_EXT_EXT1` | *EXT1* line (source needs to be selected) |
| | `DQ_EXT_SYNC0`<br>`DQ_EXT_SYNC1`<br>`DQ_EXT_SYNC2`<br>`DQ_EXT_SYNC3` | *SYNCx* interface line |
| `uint32 edge` | `DQ_EDGE_RISING`<br>`DQ_EDGE_FALLING` | |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_BAD_PARAMETER` | `clock`, `source`, or `edge` has a value other than the appropriate constants listed above |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function selects the external clock source for the *CL* or *CV* clock.

**Note:**

The external clock shall be selected in the configuration. If *EXTx* lines are used, you have to select the source signal for the lines by calling `DqAdvAssignIsoDio()`. By default, *EXT0* is assigned to *DIO0*, and *EXT1* is assigned to *DIO1*.

The function DqAdvWriteSignalRouting() must follow theDqAdvSetClockSource() in order to make the connection. For example:  DqAdvWriteSignalRouting(hd0, dev, NULL);

### *4.61.2 DqAdvSetTriggerSource*

**Syntax:**

```
int DqAdvSetTriggerSource(int hd, int devn, uint32 trigger,
uint32 source, uint32 edge)
```

**Input:**

| | | |
|---|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` | |
| `int devn` | Layer inside the IOM | |
| `uint32 trigger` | `DQ_EXT_START_TRIG` | select source for a start trigger |
| | `DQ_EXT_STOP_TRIG` | select source for a stop trigger |
| `uint32 source` | `DQ_EXT_TRIGIN` | default setting *EXT1* (DIO2) |
| | `DQ_EXT_BURST` | burst clock *TMR1* |
| | `DQ_EXT_EXT0` | line *EXT0* (source need to be selected) |
| | `DQ_EXT_EXT1` | line *EXT1* (source need to be selected) |
| | `DQ_EXT_SYNC0` | *SYNCx* interface line |
| | `DQ_EXT_SYNC1` | |
| | `DQ_EXT_SYNC2` | |
| | `DQ_EXT_SYNC3` | |
| `uint32 edge` | `DQ_EDGE_RISING` | |
| | `DQ_EDGE_FALLING` | |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_BAD_PARAMETER` | `trigger`, `source`, or `edge` has a value other than the appropriate constants listed above |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function selects an external clock source for start and stop trigger.

**Note:**

The external clock shall be selected in configuration. If EXTx lines are used, you have to select a source signal for this line by calling DqAdvAssignIsoDio(). By default, EXT0 is assigned to DIO0 and EXT1 is assigned to DIO1.

### *4.61.3 DqAdvAssignIsoDio*

**Syntax:**

```
int DqAdvAssignIsoDio(int hd, int devn, uint32 dio_line, uint32
direction, uint32 signal)
```

**Input:**

| | | |
|---|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` | |
| `int devn` | Layer inside the IOM | |
| `uint32 dio_line` | `DQ_EXT_DIO(0...3)_OFFS`<br>Number of available *DIO* lines depends on layer type. See the table below. | |
| `uint32 direction` | `DQ_EXT_DIO_INPUT` | select *DIO* as an input |
| | `DQ_EXT_DIO_OUTPUT` | select *DIO* as an output |
| | `DQ_EXT_DIO_INVERTED` | I/O signal is inverted – or with this bit to invert signal |
| `uint32 signal` | `DQ_EXT_ADCCVT` | ADC conversion clock |
| | `DQ_EXT_GPIO_LOGIC0` | Fixed level of logic 0 if pin is selected as output |
| | `DQ_EXT_GPIO_LOGIC1` | Fixed level of logic 1 if pin is selected as output |
| | `DQ_EXT_INT0` | Route *INT0* line to digital output |
| | `DQ_EXT_INT1` | Route *INT1* line to digital output |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_BAD_PARAMETER` | `dio_line`, `direction`, or `signal` has a value other than the appropriate constants listed above |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function selects the direction and signal assignment for external *DIO* line.

**Note:**

This function gives control on the direction, state, and routing of the *DIO* lines on the isolated side. This function can be used alone to select the digital line state and the direction or together with `DqAdvAssignIsoSync()` lines to assign signals to the *INTx* lines on the non-isolated side.

The default signal allocation depends on the number of external *DIO* lines available as shown in the following table.

| **Number of lines available on the layer** | **Clock-In** | **Trig-In (*DIO1*)** | **Clock-Out** | **Trig-Out (*DIO3*)** |
|---|---|---|---|---|
| 4 (AI-205) | *DIO0* | *DIO1* | *DIO2* | *DIO3* |
| 3 (AI-201, AI-225) | *DIO0* | *DIO1* | *DIO2* | |
| 2 (AO-302, DIO-403) | *DIO0* | *DIO1* | | |
| 1 (AI-208) | | *DIO0* | | |

The *EXT0* and *EXT1* lines are always assigned to *DIO0* and *DIO1*. If you select any of these *DIO* lines as an input signal from *DIO0,* it will be translated to *EXT0,* and so on.

### *4.61.4 DqAdvAssignIsoSync*

**Syntax:**

```
int DqAdvAssignIsoSync(int hd, int devn, uint32 sync_line,
uint32 signal)
```

**Input:**

| | | |
|---|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` | |
| `int devn` | Layer inside the IOM | |
| `uint32 sync_line` | `DQ_EXT_INT0` | *IS<-NIS* sync 0 |
| | `DQ_EXT_INT1` | *IS<-NIS* sync 1 |
| `uint32 signal` | `DQ_EXT_START_TRIG` | Start trigger clock (output, when started) |
| | `DQ_EXT_STOP_TRIG` | Start trigger clock (output, when started) |
| | `DQ_EXT_CLIN` | Input channel list clock (data from *IS*) |
| | `DQ_EXT_CLOUT` | Output channel list clock (data to *IS*) |
| | `DQ_EXT_CVIN` | Input conversion clock (data from *IS*) |
| | `DQ_EXT_CVOUT` | Output conversion clock (data to IS) |
| | `DQ_EXT_TSTD` | Time stamp timebase |
| | `DQ_EXT_CLOCK` | *TMR0* |
| | `DQ_EXT_BURST` | *TMR1* |
| | `DQ_EXT_SYNC0`<br>`DQ_EXT_SYNC1`<br>`DQ_EXT_SYNC2`<br>`DQ_EXT_SYNC3` | *SYNCx* line |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_BAD_PARAMETER` | `sync_line` or `signal` has a value other than the appropriate constants listed above |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |

| | |
|---|---|
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function selects the direction and signal assignment for the ISO line.

**Note:**

See the layer hardware description for a full description of the channel list implementations.

### *4.61.5 DqAdvAssignSyncx*

**Syntax:**

```
int DqAdvAssignSyncx(int hd, int devn, uint32 sync_line, uint32
signal)
```

**Input:**

| | | |
|---|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` | |
| `int devn` | Layer inside the IOM | |
| `uint32 sync_line` | `DQ_EXT_SYNC0`<br>`DQ_EXT_SYNC1`<br>`DQ_EXT_SYNC2`<br>`DQ_EXT_SYNC3` | |
| `uint32 signal` | `DQ_EXT_EXT0` | connect *EXT0* line to *SYNCx* line |
| | `DQ_EXT_EXT1` | connect *EXT1* line to *SYNCx* |
| | `DQ_EXT_START_TRIG` | start trigger pulse |
| | `DQ_EXT_STOP_TRIG` | stop trigger pulse |
| | `DQ_EXT_CLIN` | input channel list clock |
| | `DQ_EXT_CLOUT` | output channel list clock |
| | `DQ_EXT_CVIN` | input conversion clock |
| | `DQ_EXT_CVOUT` | output conversion clock |
| | `DQ_EXT_BURST` | *TMR1* |
| | `DQ_EXT_CLOCK` | *TMR0* |
| | `DQ_EXT_TSTD` | timestamp generator |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_BAD_PARAMETER` | `sync_line` or `signal` has a value other than the appropriate constants listed above |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function selects the output source to drive the *SYNCx* interface lines.

**Note:**

This function selects what to output to the *SYNCx* line. Input from the line is selected from appropriate functions.

Use `devn = 0xff` to access the CPU layer *SYNCx* lines.

When *EXTx* lines are in use, the user should select the source for these lines on the isolated side or use the default designation.

### *4.61.6 DqAdvWriteSignalRouting*

**Syntax:**

`int DqAdvWriteSignalRouting(int hd, int devn, pSGNLS psignals)`

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `pSGNLS psignals` | pointer to receive current configuration of signal routing on the layer; can be NULL if not required |

**Output:**

| | |
|---|---|
| `pSGNLS psignals` | current configuration of signal routing on the layer |

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function transfers signal assignments to IOM firmware.

**Note:**

None

## *4.62 PowerDNA buffer control*

### *4.62.1 DqAdvSetScansPerPkt*

**Syntax:**

`int DqAdvSetScansPerPkt(int hd, int dev, int ss, int scans)`

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ss` | Input subsystem number - e.g. `DQ_SS0IN` |
| `int scans` | Number of scans per packet |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_BAD_PARAMETER` | `scans` is less than 1 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function alters the number of scans required before a packet will be sent from the IOM to the host. It is used to reduce latency and is only applicable to input subsystems.

This function cannot be called when the layer is involved in any streaming or data mapping operations.

**Note:**

The maximum number of scans per packet is limited by the DaqBIOS packet size and varies with the number of channels and sample size.

### *4.62.2 DqAdvSetNetworkBuffers*

**Syntax:**

```
int DqAdvSetNetworkBuffers(int hd, int dev, int ss, int nbufs)
```

**Command:**

DQE

**Input:**

| | |
|---|---|
| `int hd` | Handle to IOM received from `DqOpenIOM()` |
| `int devn` | Layer inside the IOM |
| `int ss` | Subsystem - e.g. `DQ_SS0IN` |
| `int nbufs` | Number of IOM network buffers; must be at least 2 |

**Output:**

None.

**Return:**

| | |
|---|---|
| `DQ_NO_MEMORY` | error allocating buffer |
| `DQ_ILLEGAL_HANDLE` | illegal IOM Descriptor or communication wasn't established |
| `DQ_BAD_DEVN` | device indicated by `devn` does not exist |
| `DQ_BAD_PARAMETER` | `nbufs` is less than 2 |
| `DQ_SEND_ERROR` | unable to send the Command to IOM |
| `DQ_TIMEOUT_ERROR` | nothing is heard from the IOM for Time out duration |
| `DQ_IOM_ERROR` | error occurred at the IOM when performing this command |
| `DQ_SUCCESS` | successful completion |
| Other negative values | low level IOM error |

**Description:**

This function alters the number of network buffers allocated to each subsystem on the IOM. During high speed, low latency operations, it may be necessary to increase the number of network buffers to prevent data loss.

This function cannot be called when the layer is involved in any streaming or data mapping operations.

**Note:**

This function is only needed if the scans per packet have been altered via DqAdvSetScansPerPkt.