4.3.2 Example of aggregating multiple ports .............................................................. 28

4.4 Producer Stream API Reference ........................................................................ 28
4.4.1 Constant and Enum Definitions .................................................................. 28
4.4.2 Odi.Producers.Count .................................................................................. 30
4.4.3 Odi.Producers.Name(index) ........................................................................ 30
4.4.4 Odi.Producers.GetDataSources() ................................................................. 30
4.4.5 Odi.Producers.AddStream( name, dataSource, destinationPorts) ................. 30
4.4.6 Odi.Producers.RemoveStream( name) ......................................................... 31
4.4.7 Odi.Producers[Name].Name ....................................................................... 32
4.4.8 Odi.Producers[Name].GetCapability() ...................................................... 32
4.4.9 Odi.Producers[Name].IsFormatSupported( packetFormat, classId) ............... 33
4.4.10 Odi.Producers[Name].Activate( linkChannel, packetFormat, classId, contextClassId, streamId, ...) .............................................................. 34
4.4.11 Odi.Producers[Name].Deactivate() ......................................................... 35

4.5 Consumer Stream API Reference ...................................................................... 36
4.5.1 Constant and Enum Definitions .................................................................. 36
4.5.2 Odi. Consumers.Count .............................................................................. 36
4.5.3 Odi. Consumers.Name(index) .................................................................... 36
4.5.4 Odi. Consumers.GetDataDestinations() ...................................................... 36
4.5.5 Odi. Consumers.AddStream( name, dataDestination, sourcePorts) ............... 37
4.5.6 Odi. Consumers.RemoveStream( name) ..................................................... 38
4.5.7 Odi. Consumers[Name].Name ................................................................... 38
4.5.8 Odi. Consumers[Name].GetCapability() ................................................ 38
4.5.9 Odi. Consumers[Name].IsFormatSupported( packetFormat, classId) ............ 40
4.5.10 Odi. Consumers[Name].Activate( linkChannel, packetFormat, classId, ...) .......... 40
4.5.11 Odi. Consumers[Name].Deactivate() ..................................................... 41

Figures

Figure 1 - Single producer to single consumer ......................................................... 6
Figure 2 - Single stream using aggregated ports .................................................. 7
Figure 3 – Two streams over one link .................................................................... 7
Figure 4 - IVI Help ............................................................................................... 9
## Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-08-01</td>
<td>Add Directionality to Port.GetCapability and Port.Activate. Updated GetStatus bit definitions for flow control status. Updated ClassId list. Changed PacketFormat enum to avoid 0. Updated StreamId description. Changed FlowControl enum.</td>
</tr>
<tr>
<td>2017-09-06</td>
<td>Added ContextClassId to Producer.Activate. Updated Class ID enums. Change SCPI DIRectionality to DIRection to meet 12 character SCPI rule.</td>
</tr>
<tr>
<td>2017-11-27</td>
<td>Added DualUnidirectional to Directionality enum, and changed 3.1.1 by removing additional logical names for one physical port. Rename C structs. Added reference to VITA-49.2 Split FlowControl parameter into rxFlowControl and txFlowControl. Changed GetCapability structure field to use class id enums. Converted enums and structs to UpperCamelCase.</td>
</tr>
<tr>
<td>2018-07-18</td>
<td>Add SCPI query for ACTIVATE? Use strings for 64-bit SCPI enum values. Add additional ClassIds.</td>
</tr>
<tr>
<td>2019-01-02</td>
<td>Revision 2.0. Updated formatting of specification. Title page and font type updated. Changed CSTAtus to CSTatus.</td>
</tr>
<tr>
<td>2020-12-15</td>
<td>Revision 2.1. Added Port.Test().</td>
</tr>
</tbody>
</table>
1 Introduction

This standard establishes recommendations for implementing the programming interface of the Optical Data Interface (ODI). ODI is a high-speed data transfer interface suitable for point-to-point transfer of measurement data within a solution, using multi-gigabit FPGA transceivers, multi-lane optical interfaces, Interlaken protocol, interoperable data formats, and configured through a standardized API.

This document specifies the API of ODI. The physical, protocol, and data presentation layer (data format) are specified in other documents referenced below.

1.1 Scope

This document applies to any test and measurement product implementing ODI, and applies to the programming interface used by application software on a controller directed to an ODI-compliant device. It does not specify programming interfaces internal to a device.

1.1.1 RULE vs. RECOMMENDED

The standards are categorized as RULE or RECOMMENDATION. RULE indicates a requirement aimed at providing product compatibility and/or a common customer experience. RECOMMENDATION refers to best practices for consistency, interoperability and development efficiency.

1.2 References

ODI-1: Physical Layer  
http://www.axiestandard.org/odispecifications.html

ODI-2: Transport Layer  
http://www.axiestandard.org/odispecifications.html

ODI-2.1: High Speed Data Formats  
http://www.axiestandard.org/odispecifications.html

VITA-49.2 VITA Radio Transport (VRT) Standard,  
VITA-49A Spectrum Survey Interoperability Specification  
http://www.vita.com/
2 Overview

This API is partitioned into three areas, ODI Port Control, Data Producer Stream Control, and Data Consumer Stream Control.

- The ODI Port API methods configure and initialize a single optical interface (or port) of an ODI link. This API is common to all devices supporting ODI.

- The data Producer and Consumer Stream APIs configure the routing and formatting of data streams within a device. Although details of this API are device-specific, this standard provides a recommended pattern for APIs.

Devices may have multiple ports, and may support multiple producer streams and multiple consumer streams. The simplest case is a single producer, single port device sending data to a single port, single data consumer device.

![Diagram](image)

Figure 1 - Single producer to single consumer
More complex configurations include bonding multiple ports to accommodate higher data rates. The example below shows a single stream using two ports.

![Figure 2 - Single stream using aggregated ports](image)

Other applications may send multiple independent data streams through a single port. The example below shows how two streams might share one link.

![Figure 3 – Two streams over one link](image)
2.1 Supported Programming Environments

This API document describes interfaces for .NET, C, and SCPI programming environments. The .NET interface is also intended to be a pattern for a modern C++ interface. Individual instrument drivers can choose which of these interfaces to provide depending on the programming environments they support.

The .NET API definitions documented below assume a programming environment that supports hierarchical organization of members in interfaces, and collections of repeated capabilities. These definitions may need to be “flattened” for other environments (like native C).

In method names, Driver is a placeholder for an instrument-specific name.

Due to limitations in some SCPI implementations, ODI SCPI uses case-insensitive strings for parameters that are 64-bit enums in the C/C++ API. As command parameters, they are sent as a string matching the C/C++ enum name. Query responses return a string containing the enum name.

2.2 Example Implementation

An example IVI-NET and IVI-C implementation software package is available. This package was created using Pacific Mindworks Nimbus Driver Studio for creating instrument drivers.

This tool creates a Windows chm help file which can be used to browse the API. Figure 4 - IVI Help, shows how the ODI API appears as part of a hypothetical Keysight model Mxxxx instrument.
Contact AXIe to get a copy of these files.
3 Port API

The Port API is exposed as a collection of Port objects, each has methods to configure and initialize one optical interface (or port).

3.1 Port API Structure

`Driver.Odi.Ports. ...`

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int32 Count</td>
<td>Returns the number of ports</td>
</tr>
<tr>
<td>String Name(int index)</td>
<td>Returns name of port given 0-based index</td>
</tr>
<tr>
<td>Item[portName]</td>
<td>Ports[] is a read-only collection with an item for each physical optical interface (or port). Ports have predefined names.</td>
</tr>
<tr>
<td>String Name</td>
<td>Returns name. Read-only.</td>
</tr>
<tr>
<td>OdiPortCapability GetCapability()</td>
<td>Returns a list of capabilities of the optical port. Capability information includes lane rates supported, flow control methods supported, etc. Details below.</td>
</tr>
<tr>
<td>void Activate(LaneRate laneRate, Int32 transmitBurstMax, Directionality direction, FlowControl txFlowControl, FlowControl rxFlowControl, string options)</td>
<td>&quot;Lights up&quot; the optical port transmitters using the specified parameters. Enable the optical receivers to being clock synchronization.</td>
</tr>
<tr>
<td>void Deactivate()</td>
<td>Turns off the optical port</td>
</tr>
<tr>
<td>OdiPortStatus GetStatus()</td>
<td>Returns a status word containing bits for Active, receiver locked, CRC error, Flow Control.</td>
</tr>
<tr>
<td>OdiPortStatistics GetStatistics()</td>
<td>Returns error count and traffic statistics. Statistics are returned as an object (.NET) or structure (C).</td>
</tr>
<tr>
<td>StatusChangedEvent</td>
<td>Subscribe to an event for Status changes. Implementation is language dependent: .NET event mechanism or C callback.</td>
</tr>
<tr>
<td>Test</td>
<td>Verify operation of an ODI link by sending and receiving packets containing a standardized test pattern</td>
</tr>
</tbody>
</table>

3.1.1 Port Names

The Ports[] collection is indexed by a Port Name strings. Port names are case insensitive and should match the connector label. Recommended names are “ODI1”, “ODI2”, and so on.
3.2 Example Usage

3.2.1 Example calling sequence

<table>
<thead>
<tr>
<th>Data Producer (source)</th>
<th>Data Consumer (sink)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Select ports to be used, connect optical cable. Configure measurement settings.</td>
<td></td>
</tr>
<tr>
<td>2. Call Ports[&quot;ODI1&quot;].GetCapability() to get producer capabilities.</td>
<td>3. Call Ports[&quot;ODI1&quot;].GetCapability() to get consumer capabilities.</td>
</tr>
<tr>
<td>4. Choose common capabilities for lane rate, BurstMax, flow control.</td>
<td></td>
</tr>
<tr>
<td>7. Poll by calling Ports[&quot;ODI1&quot;].GetStatus() until link is up and ready.</td>
<td>8. Poll by calling Ports[&quot;ODI1&quot;].GetStatus() until link is up and ready.</td>
</tr>
<tr>
<td>9. Call data Producer and data Consumer methods described in the next section to configure data streams, then call instrument-specific methods to start measurement.</td>
<td></td>
</tr>
</tbody>
</table>
3.3 Port API Reference

The Port API provides methods to report capabilities, configure, and initialize the optical link. This API is common to all devices supporting ODI.

3.3.1 Constant and Enum Definitions

```java
public enum OdiLaneRate {
    _12R5G = 1,
    _14R1G = 2,
}

public enum OdiFlowControl {
    None = 1,
    InBand = 2,
    InBandPerChannel = 3,
    OutOfBand1Wire = 4,
    OutOfBandBackplane0 = 100, // PXI_TRIG0 or AXIe TRIG0
    OutOfBandBackplane1 = 101, // PXI_TRIG1 or AXIe TRIG1
 ...
    OutOfBandBackplane8 = 108, // PXI_TRIG8 or AXIe TRIG8
    OutOfBandBackplane9 = 109, // AXIe TRIG9
    OutOfBandBackplane10 = 110, // AXIe TRIG10
    OutOfBandBackplane11 = 111, // AXIe TRIG11
    OutOfBandBackplane12 = 112, // AXIe TRIG12
}

public enum OdiDirectionality {
    Bidirectional = 1, // Both TX and RX
    Producer = 2, // TX only
    Consumer = 3, // RX only
    DualUnidirectional = 4, // Independent RX and TX
}
```

C implementations may specify enum constants using #defines, with an appropriate name prefix.

3.3.2 Odi.Ports.Count

**Purpose**

Returns the number of Ports in the collection.

**SCPI Interface**

ODI:PORT:COUNt? -> <integer>

3.3.3 Odi.Ports.Name(index)

**Purpose**

Returns name of port given 0-based index.
### SCPI Interface

ODI:PORT[N]:NAME? -> "<name>"

#### 3.3.4 Odi.Ports[Name].Name

**Purpose**

Returns name of the port. Useful if a Port object passed by reference in programming environments that allow this.

#### 3.3.5 Odi.Ports[Name].GetCapability()

**Purpose**

Query capability of an ODI port.

### .Net Interface

```csharp
public struct OdiPortCapability
{
    public string Name;  // Version of the standard supported
    string Version;  // Version of the standard supported
    OdiLaneRate[] LaneRates;  // list of supported lane rates.
    Int32[] TxBurstMaxes;  // Transmitter BurstMax values supported (bytes)
    Int32 RxBurstMax;  // Receiver BurstMax in bytes.
    OdiFlowControl[] FlowControls;  // Flow control methods supported
    Int32 ChannelMax;  // Maximum Interlaken channel number supported
    OdiDirectionality Directions;  // list of Unidirectional/Bidirectional support
    Boolean TxRateMatching;  // Interlaken Sec 5.4.10 TX rate matching supported
}
```

### C Interface

```c
ViStatus Driver_OdiPortGetCapability(ViSession s, ViString port,
    ViInt32 structVersion,
    OdiPortCapability *capability);

#define ODI_PORT_CAPABILITY_VERSION 1

struct OdiPortCapability {
    ViChar name[64];
    ViChar version[64];
    OdiLaneRate laneRates[8];  // 0-terminated list of LaneRate enums
    ViUInt32 txBurstMaxes[8];  // list of Tx BurstMax values
    ViUInt32 rxBurstMax;  // Rx BurstMax
    OdiFlowControl flowControls[32];  // flow control methods supported
    ViUInt32 channelMax;  // Maximum Interlaken channel number supported
    OdiDirectionality directions[8];  // list of Unidirectional/bidirectional support
    ViBoolean txRateMatching;  // TX rate matching (limiting) supported
};
```
SCPI Interface

ODI:PORT[N]:CAPability:FCONtrols? -> <list of enum values>,
e.g. NONE, IBANd, IBPC, OOBWire, OOBBackplane0, OOBBackplane8, OOBBackplane9, OOBBackplane10, OOBBackplane11, OOBBackplane12

ODI:PORT[N]:CAPability:NAME? -> "<name>"

ODI:PORT[N]:CAPability:RATes? -> <list of enum values>, e.g. R125, R141

ODI:PORT[N]:CAPability:RBMax? -> <integer>

ODI:PORT[N]:CAPability:TBMax? -> <list of integers>, e.g. 4096, 8192

ODI:PORT[N]:CAPability:DIRection? -> <list of enum>, e.g. BIDirectional, PRODucer, CONSumer, DUAL

ODI:PORT[N]:CAPability:TRMatch? -> 0|1

ODI:PORT[N]:CAPability:VERSion? -> "<string>"

Remarks

This operation queries a capability of the optical interface port on the device.
3.3.6 Odi.Port[Name].Activate( laneRate, txBurstMax, direction, txFlowControl, rxFlowControl, options)

Purpose

Configure and activate an ODI port

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>laneRate</td>
<td>IN</td>
<td>OdiLaneRate</td>
<td>Enum value specifying lane rate.</td>
</tr>
<tr>
<td>txBurstMax</td>
<td>IN</td>
<td>Int32</td>
<td>Maximum size in bytes of Interlaken bursts</td>
</tr>
<tr>
<td>direction</td>
<td>IN</td>
<td>OdiDirectionality</td>
<td>Configure port for bidirectional or unidirectional operation.</td>
</tr>
<tr>
<td>txFlowControl</td>
<td>IN</td>
<td>OdiFlowControl</td>
<td>Enum value specifying type of transmit flow control</td>
</tr>
<tr>
<td>rxFlowControl</td>
<td>IN</td>
<td>OdiFlowControl</td>
<td>Enum value specifying type of receive flow control</td>
</tr>
<tr>
<td>options</td>
<td>IN</td>
<td>String</td>
<td>Additional instrument-specific settings</td>
</tr>
</tbody>
</table>

Exceptions/Errors

“Not Supported” if the hardware does not support the specified settings.

“In Use” if the port is already active.

.Net Interface

```c
void Activate(OdiLaneRate laneRate, int txBurstMax, OdiDirectionality direction,
              OdiFlowControl txFlowControl, OdiFlowControl rxFlowControl, string options);
```

C Interface

```c
ViStatus Driver_OdiPortActivate(ViSession s, ViString port, OdiLaneRate laneRate,
                                  int txBurstMax, OdiDirectionality direction, OdiFlowControl txFlowControl,
                                  OdiFlowControl rxFlowControl, ViString options);
```

SCPI Interface

```scpi
ODI:PORT[N]:ACTivate <laneRate>, <txBurstMax>, <direction>, <txFlowControl>, <rxFlowControl>, <options>
ODI:PORT[N]:ACTivate? ->  <laneRate>, <txBurstMax>, <direction>, <txFlowControl>, <rxFlowControl>, <options>
```

ODI-A Application Programming Interface 15  Optical Data Interface
Remarks

This method enables transmitters to transmit empty bursts, and initiates the receiver locking process. This method does not wait for receivers to complete clock synchronization.

Reset() and ResetWithDefaults() does not affect ODI Port activation. Port activation is excluded from instrument state reset to avoid resynchronization, which may be slow.

ODI hardware does not auto-negotiate settings for laneRate, txBurstMax, flowControl. Instead, the controlling software should query capabilities of both devices, select compatible settings, then Activate using those settings.

Other settings, such as PacketFormat are set on the Producer or Consumer stream, not on the Port.

3.3.7 Odi.Ports[Name].Deactivate ()

Purpose

Turn off the optical port

.Net Interface

void Deactivate();

C Interface

ViStatus Driver_OdiPortDeactivate(ViSession s, ViString port);

SCPI Interface

ODI:PORT[N]:DEACTivate

3.3.8 Odi.Ports[Name].GetStatus()

Purpose

Query status of an optical port.

.Net Interface

OdiPortStatus GetStatus();

C Interface

ViStatus Driver_OdiPortGetStatus(ViSession s, ViString port, ViUInt32* portStatus);

SCPI Interface

ODI:PORT[N]:CSTatus? -> <integer>
## Status Bits

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>0</td>
<td>Port activated by software. Actual readiness to send and receive will depend upon the opposite end of the link and flow control configuration.</td>
</tr>
<tr>
<td>TxReady</td>
<td>1</td>
<td>Ready to transmit and flow control allows. If flow control is disabled, transmit will always be ready to send. If flow control is enabled, the port will not be 'ready to send' until receive is ready and indicating XON via flow control. To troubleshoot TxReady not becoming set in this case, troubleshoot the receive path starting with RxSignalLoss.</td>
</tr>
<tr>
<td>RxReady</td>
<td>2</td>
<td>Receiver ready. All lanes synchronized and aligned.</td>
</tr>
<tr>
<td>RxLaneError</td>
<td>3</td>
<td>Error in one or more lanes since last GetStatus.</td>
</tr>
<tr>
<td>RxBurstMaxError</td>
<td>4</td>
<td>Received too large a burst since last GetStatus.</td>
</tr>
<tr>
<td>RxCRCError</td>
<td>5</td>
<td>Received bad burst CRC since last GetStatus.</td>
</tr>
<tr>
<td>RxOverrun</td>
<td>6</td>
<td>Receiver data overrun since last GetStatus</td>
</tr>
<tr>
<td>RxSignalLoss</td>
<td>7</td>
<td>Received signal loss. Optical power too low.</td>
</tr>
<tr>
<td>RxSyncPending</td>
<td>8</td>
<td>Receiver activated but has not achieved synchronization</td>
</tr>
<tr>
<td>9 to 15</td>
<td></td>
<td>Unused</td>
</tr>
<tr>
<td>RxFcStatus</td>
<td>16</td>
<td>Received link-level flow control status. 1 is XON, 0 is XOFF. From Interlaken idle/control word bit 55 or from an out-of-band flow control signal.</td>
</tr>
<tr>
<td>RxFcStatus0 to</td>
<td></td>
<td>Received per-channel flow control status bits. 1 is XON, 0 is XOFF. Bit 17 is channel 0 from bit 54 of the Interlaken idle/control word, bit 18 is channel 1 from bit 53 of the control word, and so on.</td>
</tr>
<tr>
<td>RxFcStatus14</td>
<td>17 to 31</td>
<td></td>
</tr>
</tbody>
</table>

Status bits described with “since last GetStatus” are cleared by GetStatus(). All status bits will be 0 on an inactive port.

```java
public enum OdiPortStatus {
    Active = 0x0001,     // port has been activated
    TxReady = 0x0002,     // Ready to transmit and flow control allows
    RxReady = 0x0004,     // Rx synchronized and aligned
    RxLaneError = 0x0008, // RX error in metaframe, scrambler, ...
    RxBurstMaxError = 0x0010, // Received too large of a burst.
    RxCRCError = 0x0020,   // CRC error in Burst
    RxOverrun = 0x0040,
    RxSignalLoss = 0x0080, // Received signal loss, optical power too low.
    RxSyncPending = 0x0100, // Receiver activated but has not achieved synchronization
```
3.3.9 Odi.Ports[Name].GetStatistics()

Purpose

Query statistics of an optical port.

.Net Interface

```csharp
OdiPortStatistics GetStatistics();

public struct OdiPortStatistics {
    public Int64 BytesSent; // cumulative bytes sent
    public Int64 BytesReceived; // cumulative bytes received
    public Int64 BadBurstsReceived; // number of bursts received with bad CRC
    public Int64 TxFlowControlHoldoffs; // number of clock cycles TX was held off
}
```

C Interface

```c
ViStatus Driver_OdiPortGetStatistics(ViSession s, ViString port,
    ViInt32 structVersion, OdiPortStatistics *statistics);

#define ODI_PORT_STATISTICS_VERSION 1
struct OdiPortStatistics {
    ViUInt64 bytesSent; // cumulative bytes sent
    ViUInt64 bytesReceived; // cumulative bytes received
    ViUInt64 badBurstsReceived; // number of bursts received with bad CRC
    ViUInt64 txFlowControlHoldoffs; // number of clock cycles TX was held off
};
```

SCPI Interface

```scpi
ODI:PORT[N]:PSTatistics:BBURrst? -> <integer>
```
Remarks

This operation queries the statistics of the optical interface port on the device. Statistics are cumulative since the port was Activated.

3.3.10 Odi.Ports[Name]. StatusChangedEvent

Purpose

Enable notification of port events by reporting changes to port status.

.Net Interface

```csharp
void StatusChangedHandler(object sender, OdiPortStatusChangedEventArgs e);
```

```csharp
event EventHandler<OdiPortStatusChangedEventArgs> StatusChangedEvent;
```

C Interface

```c
typedef void (*PORT_EVENT_CALLBACK)(ViString port, ViPBuf context, ViUInt32 portStatus);
```

```c
ViStatus Driver_OdiPortEnableEvent(ViSession s, ViString port, PORT_EVENT_CALLBACK handler, ViPBuf context);
```

SCPI Interface

This command is not available in SCPI.

Remarks

Enables reporting of status changes to the calling software. The provided Event (.NET) or callback function pointer (C language), is called when certain status bits change. Changes in Flow Control status bits are excluded.

To disable event reporting, call EnableEvent() with null for the handler.

Implementation of PortStatusChangedEvent () is optional. Some products may report Not Implemented error.
3.3.11 Odi.Ports[Name].Test()

Purpose

Verify operation of an ODI link by sending and receiving packets containing a standardized test pattern.

Three test modes:

- **Tx**: Instructs an ODI port to produce the test pattern data. This mode is used by a data producer ODI device.
- **Rx**: Instructs an ODI port to expect to receive the test pattern data. This mode is used by a data consumer ODI device.
- **Loopback**: Instructs an ODI port to send and receive. This mode requires a loopback optical cable adapter.

The TX and RX modes are intended to support verification testing in deployed systems without disturbing optical cabling.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestMode</td>
<td>IN</td>
<td>OdiTestMode</td>
<td>Enum value specifying Tx, Rx, or Loopback</td>
</tr>
<tr>
<td>PacketCount</td>
<td>IN</td>
<td>Int32</td>
<td>Number of packets to send and expect to receive.</td>
</tr>
<tr>
<td>Message</td>
<td>OUT</td>
<td>String</td>
<td>Returns additional information about test failures.</td>
</tr>
<tr>
<td>Return</td>
<td></td>
<td>OdiTestResult</td>
<td>Enum. 0 = Pass, -3 = Timeout, ...&lt;0 Other Fail,</td>
</tr>
</tbody>
</table>

Enum definitions

```java
public enum OdiTestMode
{
    Tx = 0,
    Rx = 1,
    Loopback = 2,
}
```
public enum OdiTestResult {
    Pass = 0,
    SetupError = -2,
    TxTimeout = -3,
    RxTimeout = -4,
    RxNotReadyError = -5,
    RxSignalError = -6,
    RxDataError = -7,
    OtherFail = -15,
}

.Net Interface

OdiTestResult Test ( OdiTestMode testMode, int packetCount, out String message);

C Interface

ViStatus Driver_OdiPortTest (ViSession s, ViString port, OdiTestMode testMode,
    int packetCount, ViInt32 MessageBufferSize, ViString message, ViInt32* testResult);

SCPI Interface

ODI:PORT[N]:TEST? <mode> [, <packetCount>]

Remarks

Before performing a test, the Port must first be Activate()-ed to set line rate and other parameters.

The test uses a packet size of 16384 bytes. The default PacketCount of 1048576 packets results in a test duration of about 1 second.

In TX mode, the test method returns successfully after sending the packets, or returns a timeout error if unable to send for more than 30 seconds. In Rx mode, the test method blocks until receiving packets with the expected pattern, within a timeout of 30 seconds plus the estimated transmission time. To test a link first start the Rx device test, then start the Tx device test.
### Example

<table>
<thead>
<tr>
<th>Data Producer Device</th>
<th>Data Consumer Device</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Activate Port, as Producer or as Bi-directional</strong></td>
<td></td>
</tr>
<tr>
<td><strong>SCPI:</strong> PORT:ACT R141, 2048, Bidirectional</td>
<td><strong>SCPI:</strong> PORT:ACT R141, 2048, Bidirectional</td>
</tr>
<tr>
<td><strong>C/Net:</strong> Port.Activate(R141, 2048, Bidirectional, ..);</td>
<td><strong>C/Net:</strong> Port.Activate(R141, 2048, Bidirectional, ..);</td>
</tr>
<tr>
<td><strong>Activate Port, as Consumer or as Bi-directional</strong></td>
<td></td>
</tr>
<tr>
<td><strong>SCPI:</strong> PORT:ACT R141, 2048, Bidirectional</td>
<td><strong>SCPI:</strong> PORT:ACT R141, 2048, Bidirectional</td>
</tr>
<tr>
<td><strong>C/Net:</strong> Port.Activate(R141, 2048, Bidirectional, ..);</td>
<td><strong>C/Net:</strong> Port.Activate(R141, 2048, Bidirectional, ..);</td>
</tr>
<tr>
<td><strong>Wait until Port status shows RxReady</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Wait until Port status shows TxReady</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Start receiving test</strong></td>
<td></td>
</tr>
<tr>
<td><strong>SCPI:</strong> PORT:TEST? &quot;RX&quot;</td>
<td><strong>SCPI:</strong> PORT:TEST? &quot;RX&quot;</td>
</tr>
<tr>
<td><strong>C/Net:</strong> Port.Test(Rx, ...);</td>
<td><strong>C/Net:</strong> Port.Test(Rx, ...);</td>
</tr>
<tr>
<td><strong>Start transmitting test</strong></td>
<td></td>
</tr>
<tr>
<td><strong>SCPI:</strong> PORT:TEST? &quot;TX&quot;</td>
<td></td>
</tr>
<tr>
<td><strong>C/Net:</strong> Port.Test(Tx, ...);</td>
<td></td>
</tr>
<tr>
<td><strong>Expect response like &quot;0, Pass&quot;, or &quot;-3, TxTimeout&quot;</strong></td>
<td>**Expect response like &quot;0, Pass&quot;, or &quot;-6, Failed. Bad packets: 2, Bad lanes: 2, 3&quot;</td>
</tr>
</tbody>
</table>

Optical Data Interface
Loopback Example

<table>
<thead>
<tr>
<th>Instrument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attach loopback dongle to port</td>
</tr>
<tr>
<td>Activate Port, as Bi-directional</td>
</tr>
<tr>
<td>Wait until Port status shows TxReady, RxRead</td>
</tr>
<tr>
<td>Start loopback test</td>
</tr>
<tr>
<td>SCPI: PORT:TEST? &quot;LOOPBACK&quot;</td>
</tr>
<tr>
<td>C/Net: Port.Test(Loopback, ...);</td>
</tr>
</tbody>
</table>

Expect response like "0, Pass", or 
"-4, RxTimeout"

Test Pattern
The test data is a counting pattern encoded in 4 byte (32 bit) words of the data stream. The pattern is specified in section 4.5 “ODI Test Pattern” of ODI-1 Physical Layer Specification, revision 3.1 or later.
4 Producer and Consumer Stream APIs

The Producer and Consumer Stream APIs configure the routing and formatting of data streams within a device.

In this API, *Producer* is short for *Data Producer Stream*. A digitizer may implement one or more Producer streams to route and format data from one or more ADC channels to one or more ODI ports.

*Consumer* is short for *Data Consumer Stream*. An AWG may implement one or more Consumer streams to route and format data from one or more ODI ports to one or more DAC channels.

4.1 Producer Stream API Structure

<table>
<thead>
<tr>
<th><strong>Driver.Odi.Producers</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int32 Count</td>
<td>Returns the number of producer streams.</td>
</tr>
<tr>
<td>String Name(int index)</td>
<td>Returns name of stream, given 0-based index</td>
</tr>
<tr>
<td>String GetDataSources()</td>
<td>(optional) Returns a comma separated list of valid DataSources for AddStream().</td>
</tr>
<tr>
<td>AddStream(</td>
<td>(optional) Dynamically add new Producer stream to the collection, connecting a data source to one or more ODI ports. Name is user-supplied, becomes key for Item[]. DataSource names are instrument specific, and can be queried with GetDataSources(). DestinationPorts is a comma separated list of Port names.</td>
</tr>
<tr>
<td>RemoveStream(String Name)</td>
<td>(optional) Disconnect a stream created by AddStream(). SCPI command: ODI:PRODucer:RSTReam &lt;name&gt;</td>
</tr>
<tr>
<td>Item[Name]</td>
<td>Collection of producer streams. May initially be empty, or pre-populated with pre-defined producer streams.</td>
</tr>
<tr>
<td><strong>Driver.Odi.Producers.Item[Name]</strong>,</td>
<td></td>
</tr>
<tr>
<td><strong>Driver.Odi.Producers[Name]</strong></td>
<td></td>
</tr>
<tr>
<td>String Name</td>
<td>Returns name. Read-only.</td>
</tr>
<tr>
<td>OdiProducerCapability GetCapability()</td>
<td>Returns a list of capabilities of the data producer, including packet formats and binary data formats.</td>
</tr>
<tr>
<td>IsFormatSupported(</td>
<td>Returns true if the combination of PacketFormat and Vita49ClassId is supported.</td>
</tr>
<tr>
<td>OdiPacketFormat packetFormat,</td>
<td></td>
</tr>
<tr>
<td>Vita49ClassId classId)</td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>`void Activate(int32 linkChannel,</td>
<td>Configure the data producer stream and activate it for use.</td>
</tr>
<tr>
<td>OdiPacketFormat packetFormat,</td>
<td>LinkChannel sets the Interlaken packet channel (not measurement channel).</td>
</tr>
<tr>
<td>Vita49ClassId ClassId,</td>
<td>PacketFormat specifies no header, or VITA-49, or other packet formats.</td>
</tr>
<tr>
<td>Vita49ClassId contextClassId,</td>
<td>ClassId specifies the binary format of the payload data, even if not using headers or VITA-49 packets.</td>
</tr>
<tr>
<td>int32 streamId,</td>
<td>ContextClassId specifies the format of context packets, or 0 if none.</td>
</tr>
<tr>
<td>OdiTimestampFormat timestamp,</td>
<td>Additional &lt;device params&gt; specify domain-dependent parameters such as packet size.</td>
</tr>
<tr>
<td>int32 packetSizeLimit,</td>
<td>&lt;device params&gt;…</td>
</tr>
<tr>
<td>&lt;device params&gt;…</td>
<td></td>
</tr>
<tr>
<td><code>Void Deactivate()</code></td>
<td>Deactivates the stream.</td>
</tr>
</tbody>
</table>

The ODI API does not include methods to start or stop the stream. These functions are expected to be implemented by other measurement-specific areas of the API.

### 4.1.1 Producer Stream Names

The Producers[] collection is indexed by a name strings. Names are case insensitive. Names are IVI repeated capability identifiers and so cannot contain space or ‘-‘ characters. The recommended format is

<source>_to_<destination>

where <source> is instrument-specific and should match channel or other names used in the product. <destination> is an “and” separated list of Port names. Some example names are

- “Ch1_to_ODI1"
- “Ch1DDC_to_ODI1”
- “Ch1_to_ODI1_and_ODI2”

### 4.2 Consumer Stream API Structure

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Driver.Odi.Consumers.Int32 Count</code></td>
<td>Returns the number of consumer streams.</td>
</tr>
<tr>
<td><code>Driver.Odi.Consumers.String Name(int index)</code></td>
<td>Returns name of stream, given 0-based index</td>
</tr>
<tr>
<td><code>Driver.Odi.Consumers.String GetDataDestinations()</code></td>
<td>(optional) Returns a comma separated list of valid DataDestinations for AddStream().</td>
</tr>
<tr>
<td>`Driver.Odi.Consumers.AddStream(String name,</td>
<td>(optional) Dynamically add new Consumer stream to the collection, connecting one or more ODI ports to a data destination. Name is user-supplied, becomes key for Item[]. DataDestination names are instrument specific, and can be queried with GetDataDestination(). Source Ports is a comma separated list of Port names.</td>
</tr>
<tr>
<td>String dataDestination,</td>
<td></td>
</tr>
<tr>
<td>String sourcePorts,</td>
<td></td>
</tr>
<tr>
<td>String options)`</td>
<td></td>
</tr>
</tbody>
</table>
The ODI API does not include methods to start or stop the stream. These functions are expected to be implemented by other measurement-specific areas of the API.

### 4.2.1 Consumer Stream Names

The Consumers[] collection is indexed by a name strings. Names are case insensitive. The recommended format is

<source>_to_<destination>

where <source> is an “and” separated list of Port names, and <destination> is instrument-specific and should match channel or other names used in the product. Some example names are

- “ODI1_to_Ch1”
- “ODI1_to_Ch1DUC”
- “ODI1_and_ODI2_to_Ch1”
4.3 Example Usage

4.3.1 Example calling sequence
The following sequence assumes the instruments have pre-defined streams, and the Ports involved are already activated.

<table>
<thead>
<tr>
<th>Data Producer (source)</th>
<th>Data Consumer (sink)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Select ports to be used, connect optical cable, Activate Ports, and check Port status. See Port Example calling sequence</td>
<td></td>
</tr>
<tr>
<td>2. Get Producer names and capabilities by consulting documentation, or by iterating through each Producer and calling .GetCapability().</td>
<td>3. Get Consumer names and capabilities by consulting documentation, or by iterating through each Consumer and calling .GetCapability().</td>
</tr>
<tr>
<td>4. Choose common PacketFormat and Vita49ClassId.</td>
<td>5. Call Consumers[“ODI1_to_Ch1”].Activate( 0, Vita49Data, Iq16Bit1Ch, ... ) to configure and activate the stream.</td>
</tr>
<tr>
<td>6. Call Producers[“Ch1_to_ODI1”].Activate( 0, Vita49Data, Iq16Bit1Ch, ... ) to configure and activate the stream.</td>
<td>7. Call instrument-specific method to start signal generation or processing from stream, e.g. Driver.Play(). It will stall waiting for data.</td>
</tr>
<tr>
<td>8. Call instrument-specific method to start digitizing or other data-producing activity, e.g. Driver.Arm(). This step starts the flow of data.</td>
<td>9. Use instrument-specific methods to determine when a finite-length stream is complete, or to stop an indefinite-length stream.</td>
</tr>
<tr>
<td>10. Call Producers[“ODI1_to_Ch1”].Deactivate();</td>
<td>11. Call Consumers[“ODI1_to_Ch1”].Deactivate();</td>
</tr>
</tbody>
</table>
4.3.2 Example of aggregating multiple ports

An instrument may allow aggregation of multiple ports to achieve higher data bandwidth. In this case a single stream is associated multiple ports. Below are the steps to configure.

1. Activate the Ports in the Port1-to-Port1 link.
2. Activate the Ports in the Port2-to-Port2 link.
3. On device A, find (or AddStream) the Producer stream. A typical name for this producer stream might be “Ch1_to_ODI1_and_ODI2”. Activate() it.
4. On device B, find (or AddStream) the Consumer stream. A typical name for this consumer stream might be “ODI1_and_ODI2_to_Ch1”. Activate() it.
5. On device B, start Data Consumer 1.

4.4 Producer Stream API Reference

4.4.1 Constant and Enum Definitions

```java
public enum OdiPacketFormat
{
  NoHeader = 1,  // raw binary samples with no header.
```
Vita49Data = 2, // VITA-49 Signal Data packets.
Vita49Extension = 4, // VITA-49 Extension packets.

ProductSpecific = 1000, // Starting enum for other non-standard formats.
Vita49Once = 1001, // Single preamble VITA-49 Context, then NO_HEADER.

} // ClassId enums define common ClassIds, per VITA-49A
// This enum includes common formats for 1 or 2 channels. It is not a complete list of valid classIds.
// To form class ids for higher channel counts, set the least significant byte to channelCount-1.
public enum Vita49ClassId : UInt64
{
    Unknown = 0,
    // ClassIds with org Id 24-5C-CB are defined by ODI.
    // The following processing efficient formats are patterned after VITA-49A
    Re8Bit1Ch = 0x00245CCB00020000, // Real (baseband), 8-bit signed, 1 channel
    Re8Bit2Ch = 0x00245CCB00020001, // Real (baseband), 8-bit signed, 2 channel
    Re16Bit1Ch = 0x00245CCB00030000, // Real (baseband), 16-bit signed, 1 channel
    Re16Bit2Ch = 0x00245CCB00030001, // Real (baseband), 16-bit signed, 2 channel
    Re16Bit4Ch = 0x00245CCB00030003, // Real (baseband), 16-bit signed, 4 channel
    Re32BitFloat1Ch = 0x00245CCB00060000, // Real (baseband), 32-bit float, 1 channel
    Re9BitPacked1Ch = 0x00245CCB00002000, // Real, 9-bit link efficient, 1 ch
    Re10BitPacked1Ch = 0x00245CCB00004000, // Real, 10-bit link efficient, 1 ch
    Re11BitPacked1Ch = 0x00245CCB00006000, // Real, 11-bit link efficient, 1 ch
    Re12BitPacked1Ch = 0x00245CCB00008000, // Real, 12-bit link efficient, 1 ch
    Re13BitPacked1Ch = 0x00245CCB0000A000, // Real, 13-bit link efficient, 1 ch
    Re14BitPacked1Ch = 0x00245CCB0000C000, // Real, 14-bit link efficient, 1 ch
    Re15BitPacked1Ch = 0x00245CCB0000E000, // Real, 15-bit link efficient, 1 ch
    Iq9BitPacked1Ch = 0x00245CCB00102000, // Complex, 9-bit link efficient, 1 ch
    Iq10BitPacked1Ch = 0x00245CCB00104000, // Complex, 10-bit link efficient, 1 ch
    Iq11BitPacked1Ch = 0x00245CCB00106000, // Complex, 11-bit link efficient, 1 ch
    Iq12BitPacked1Ch = 0x00245CCB00108000, // Complex, 12-bit link efficient, 1 ch
    Iq13BitPacked1Ch = 0x00245CCB0010A000, // Complex, 13-bit link efficient, 1 ch
    Iq14BitPacked1Ch = 0x00245CCB0010C000, // Complex, 14-bit link efficient, 1 ch
    Iq15BitPacked1Ch = 0x00245CCB0010E000, // Complex, 15-bit link efficient, 1 ch
    // The following link-efficient ClassIds are not patterned after VITA-49A
    Re12Bit4Event1Ch = 0x00245CCB00C30000, // Real, 12-bit + 4 event bits, 1 ch
    Re14Bit2Event1Ch = 0x00245CCB00830000, // Real, 14-bit + 2 event bits, 1 ch
    Iq14Bit2Event1Ch = 0x00245CCB00930000, // Complex, 14-bit + 2 event bits, 1 ch
    Re15Bit1Event1Ch = 0x00245CCB00430000, // Real, 15-bit + 1 event bit, 1 ch
    Iq15Bit1Event1Ch = 0x00245CCB00530000, // Complex, 15-bit + 1 event bit, 1 ch

} // Context Packet ClassId enums define common ClassIds, per VITA-49A
public enum Vita49ContextClassId : UInt64
{
    None = 0,
    // ClassIds with org Id 24-5C-CB are defined by ODI.
    OdiStandardizedContext = 0x00245CCB20170010, // ODI standard 96 byte context packet

}
public enum OdiTimestampFormat
{
    NoTimestamp = 1,  // Timestamp not used.
    Gps = 2,         // GPS seconds and pico-seconds per VITA-49.
    Relative = 3,    // Seconds and pico-seconds.
    SampleCount = 4, // 64-bit sample count in Fractional Seconds field.
    Utc = 5,         // UTC seconds and ps (not ODI recommended).
}

4.4.2 Odi.Producers.Count

Purpose
Returns the number of Producers currently in the collection.

SCPI Interface
ODI:PRODucer:COUNt? -> <integer>

4.4.3 Odi.Producers.Name(index)

Purpose
Returns name of producer given 0-based index.

SCPI Interface
ODI:PRODucer[N]:NAME? -> "<string>"

4.4.4 Odi.Producers.GetDataSources()

Purpose
Returns a comma separated list of data source names, each valid as the dataSource parameter of AddStream().

.Net Interface
    string GetDataSources();

C Interface
    ViStatus Driver_OdiProducerGetDatasources(ViSession s, int bufferSize, ViString *buffer);

SCPI Interface
ODI:PRODucer:DSOurces? -> <comma separated list>, e.g. "cha1,cha2"

4.4.5 Odi.Producers.AddStream( name, dataSource, destinationPorts)

Purpose
Dynamically add new Producer stream to the collection, connecting a data source to one or more ODI ports.
Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>IN</td>
<td>String</td>
<td>User-supplied handle, becomes key for Item[].</td>
</tr>
<tr>
<td>dataSource</td>
<td>IN</td>
<td>String</td>
<td>Specifies the data source. Source names are instrument specific, and can be queried with GetDataSources(). Although a stream can have only one source name, the name can represent a set of measurement channels.</td>
</tr>
<tr>
<td>destinationPorts</td>
<td>IN</td>
<td>String</td>
<td>DestinationPorts is a comma separated list of Port names.</td>
</tr>
<tr>
<td>options</td>
<td>IN</td>
<td>String</td>
<td>Additional instrument-specific settings</td>
</tr>
</tbody>
</table>

.Net Interface

```csharp
void AddStream(string name, string dataSource, string destinationPorts, string options);
```

C Interface

```c
ViStatus Driver_OdiProducerAddStream(ViSession s, ViString name, ViString dataSource, ViString destinationPorts, ViString options);
```

SCPI Interface

```scpi
ODI:PRODucer:ASTReam <name>, <dataSource>, <destinationPorts>, <options>
```

Remarks

Implementation of this method is optional. Instruments with a limited number of dataSource and destinationPort combinations can instead pre-define each of those Producer streams.

4.4.6 Odi.Producers.RemoveStream( name)

Purpose

Removes a dynamically added Producer stream from the collection.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>IN</td>
<td>String</td>
<td>Name supplied during AddStream.</td>
</tr>
</tbody>
</table>
.Net Interface

```csharp
void RemoveStream(string name);
```

C Interface

```c
ViStatus Driver_OdiProducerRemoveStream(ViSession s, ViString name);
```

SCPI Interface

```
ODI:PRODucer:RSTMReam <name>
```

Remarks

Implementation of this method is optional.

### 4.4.7 Odi.Producers[Name].Name

**Purpose**

Returns name of the producer. Useful if a Port object passed by reference in programming environments that allow this.

**SCPI Interface**

```
ODI:PRODucer[N]:NAME? - > "<string>"
```

### 4.4.8 Odi.Producers[Name].GetCapability()

**Purpose**

Query capability of a producer stream.

**.Net Interface**

```
OdiProducerCapability GetCapability();
public struct OdiProducerCapability {
    public string Name;
    public string Version;  // Version of the standard supported
    public OdiPacketFormat[] PacketFormats; // all packet header formats supported
    public Vita49ClassId[] Vita49ClassIds; // all binary formats supported
    public Vita49ContextClassId[] ContextClassIds; // all context packets supported
    public OdiTimestampFormat[] TimestampFormats; // all timestamp formats supported
}
```

**C Interface**

```
ViStatus Driver_OdiProducerGetCapability(ViSession s, ViString name, 
    ViInt32 structVersion, OdiProducerCapability *capability);
```

#define ODI_PRODUCER_CAPABILITY_VERSION 1
```c
struct OdiProducerCapability {
    ViChar name[64];
    ViChar version[64];
```
SCPI Interface

ODI:PRODucer[N]:CAPability:CLIDs? -> <list of integers values>, e.g. "IQ8BIT1CH", "IQ16BIT1CH"

ODI:PRODucer[N]:CAPability:NAME? -> "<string>"

ODI:PRODucer[N]:CAPability:PFORmats? -> <list of enum values>, e.g. NHEader,IData,CONText,ONCE

ODI:PRODucer[N]:CAPability:VERSion? -> "<string>"

ODI:PRODucer[N]:TIMestamp:TFORmats? -> <list of enum values>, e.g. NOTimestamp,GPS,RELative,SAMPlecount

Remarks

This operation queries a capability of the data producer stream.

The returned lists of OdiPacketFormats and Vita49ClassIds does not imply all combinations are supported. Call IsFormatSupported() to determine support for a combination.

Due to SCPI limitations, 64-bit enum values such as CLIDs are sent as strings.

4.4.9 Odi.Producers[Name].IsFormatSupported( packetFormat, classId)

Purpose

Returns true if the combination of PacketFormat and binary data format (ClassId) is supported.

.Net Interface

    Boolean IsFormatSupported(OdiPacketFormatEnum PacketFormat, Vita49ClassIdEnum Vita49ClassId);

C Interface

    ViStatus Driver_OdiProducerIsFormatSupported(ViSession s, OdiPacketFormatEnum PacketFormat, Vita49ClassIdEnum Vita49ClassId, ViBoolean *supported);
SCPI Interface

odi:producer[n]:ifsupported? <packetformat>, <classid> -> 0|1

4.4.10 Odi.Producers[Name].Activate( linkChannel, packetFormat, classId, contextClassId, streamId, ...)

Purpose

Configure the data producer stream and activate it for use.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linkChannel</td>
<td>IN</td>
<td>Int32</td>
<td>Interlaken channel tag to apply to stream (not the measurement channel).</td>
</tr>
<tr>
<td>packetFormat</td>
<td>IN</td>
<td>OdiPacketFormat enum</td>
<td>Specifies format of any packet header and meta-data. Select No header, VITA-49 header, or other.</td>
</tr>
<tr>
<td>classId</td>
<td>IN</td>
<td>Vita49ClassId enum</td>
<td>Enum (or #defined) value specifying the binary data format of the packet payload, using VITA-49 Class Id as defined in VITA-49.2 and the ODI data format standard.</td>
</tr>
<tr>
<td>contextClassId</td>
<td>IN</td>
<td>Vita49ContextClassId enum</td>
<td>Enum (or #defined) value specifying the format of context packets, or 0 if not using context packets.</td>
</tr>
<tr>
<td>streamId</td>
<td>IN</td>
<td>Int32</td>
<td>Specifies VITA-49 Stream Identifier value to be placed in the IF Data Packet when using VITA-49 packet format. Typically 4096. When aggregating ports, this value is used on the first port, and incremented by 1024 for each additional port.</td>
</tr>
<tr>
<td>timestampFormat</td>
<td>IN</td>
<td>OdiTimestampFormat enum</td>
<td>Specifies format of the timestamp field sent in the VITA-49 prolog.</td>
</tr>
<tr>
<td>packetSizeLimit</td>
<td>IN</td>
<td>Int32</td>
<td>Specifies a maximum size for packets, in bytes. 0 indicates default of 262144.</td>
</tr>
<tr>
<td>...</td>
<td>IN</td>
<td></td>
<td>Additional instrument-specific settings for properties such as packet length.</td>
</tr>
</tbody>
</table>

Exceptions/Errors

“Not Supported” if the stream does not support the specified settings.
“In Use” if the stream or resources are already in use.

.Net Interface

```csharp
void Activate(Int32 linkChannel, OdiPacketFormat packetFormat, Vita49ClassId classId,
              Vita49ContextClassId contextClassId, Int32 streamId,
              OdiTimestampFormat timestampFormat, Int32 packetSizeLimit,...);
```

C Interface

```c
ViStatus Driver_OdiProducerActivate(ViSession s, Int32 linkChannel,
                                    OdiPacketFormat packetFormat, Vita49ClassId classId,
                                    Vita49ContextClassId contextClassId, Int32 streamId,
                                    OdiTimestampFormat timestampFormat, Int32 packetSizeLimit, ...);
```

SCPI Interface

```c
ODI:PRODucer[N]:ACTivate <linkChannel>, <packetFormat>, <classId>,
                 <contextClassId>, <streamId>, <packetSizeLimit>, ...
ODI:PRODucer[N]:ACTivate? -> <linkChannel>, <packetFormat>, <classId>, <contextClassId>,
                           <timestampFormat>, <streamId>, <packetSizeLimit>, ...
```

Remarks

Configures and activates the producer stream. On return the stream is ready to flow data, but in most case another instrument-specific method must be called to start the flow.

Calling this method may result in reconfiguration of an FPGA in the instrument, which may take several seconds.

Due to SCPI limitations, 64-bit enum values such as CLIDs are handled as strings.

4.4.11 Odi.Producers[Name].Deactivate()

Purpose

Deactivate the stream, stop any data flow, and free resources.

.Net Interface

```csharp
void Deactivate();
```

C Interface

```c
ViStatus Driver_OdiProducerDeactivate(ViSession s, ViString name);
```

SCPI Interface

```c
ODI:PRODucer[N]:DEACTivate
```
4.5 Consumer Stream API Reference

4.5.1 Constant and Enum Definitions
Same as defined for producer streams. See 4.4.1 Constant and Enum Definitions

4.5.2 Odi. Consumers.Count

Purpose
Returns the number of Consumers currently in the collection.

SCPI Interface
ODI:CONSumer:COUNt? - > <integer>

4.5.3 Odi. Consumers.Name(index)

Purpose
Returns name of consumer given 0-based index.

SCPI Interface
ODI:CONSumer[N]:NAME? - > "<string>"

4.5.4 Odi.Consumers.GetDataDestinations()

Purpose
Returns a comma separated list of data destination names, each valid as the dataDestination parameter of AddStream().

.Net Interface
string GetDataDestinations();

C Interface
ViStatus Driver_OdiConsumerGetDataDestinations(ViSession s, int bufferSize, ViString *buffer);

SCPI Interface
ODI:CONSumer:DESTination? - > <comma separated list >, e.g. "ch1,ch2"
4.5.5  Odi.Consumers.AddStream( name, dataDestination, sourcePorts)

Purpose

Dynamically add new Consumer stream to the collection, connecting one or more ODI ports to a data destination.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>IN</td>
<td>String</td>
<td>User-supplied handle, becomes key for Item[]</td>
</tr>
<tr>
<td>dataDestination</td>
<td>IN</td>
<td>String</td>
<td>Specifies the data destination. Destination names are instrument specific, and can be queried with GetDataDestinations(). Although a stream can have only one destination name, the name can represent a set of channels.</td>
</tr>
<tr>
<td>sourcePorts</td>
<td>IN</td>
<td>String</td>
<td>SourcePorts is a comma separated list of ODI Port names.</td>
</tr>
<tr>
<td>options</td>
<td>IN</td>
<td>String</td>
<td>Additional instrument-specific settings</td>
</tr>
</tbody>
</table>

.Net Interface

```csharp
void AddStream(string name, string dataDestination, string sourcePorts, string options);
```

C Interface

```c
ViStatus Driver_OdiProducerAddStream(ViSession s, ViString name, 
ViString dataDestination, ViString sourcePorts, ViString options);
```

SCPI Interface

```
ODI:CONSUMER:ASTReam <name>, <dataDestination>, <sourcePorts>, <options>
```

Remarks

Implementation of this method is optional. Instruments with a limited number of dataDestination and destinationPort combinations can instead pre-define each of those Consumer streams.
4.5.6 **Odi.Consumers.RemoveStream( name)**

**Purpose**
Remove a dynamically added Consumer stream from the collection.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>IN</td>
<td>String</td>
<td>User-supplied handle, becomes key for Item[].</td>
</tr>
</tbody>
</table>

**.Net Interface**

```csharp
void RemoveStream(string name);
```

**C Interface**

```c
ViStatus Driver_OdiProducerRemoveStream(ViSession s, ViString name);
```

**SCPI Interface**

```scpi
ODI:CONSumer:RSTReam <name>
```

**Remarks**
Implementation of this method is optional.

4.5.7 **Odi.Consumers[Name].Name**

**Purpose**
Returns name of the consumer. Useful if a Port object passed by reference in programming environments that allow this.

**SCPI Interface**

```scpi
ODI:CONSumer[N]:NAME? -> "<string>"
```

4.5.8 **Odi.Consumers[Name].GetCapability()**

**Purpose**
Query capability of a consumer stream.

**.Net Interface**
ODiConsumerCapability GetCapability();

public struct OdiConsumerCapability {
    public string Name;
    public string Version; // Version of the standard supported
    public OdiPacketFormat[] PacketFormats; // all packet header formats supported
    public Vita49ClassId[] Vita49ClassIds; // all binary formats supported
    public Vita49ContextClassId[] ContextClassIds; // all context packets supported
    public OdiTimestampFormat[] TimestampFormats; // all timestamp formats supported
    public bool HasLinkChannelFilter; // true if can support linkChannelFilter
    public UInt32[] MaxPacketSize; // maximum packet size in bytes
}

C Interface

ViStatus Driver_OdiConsumerGetCapability(ViSession s, ViString name,
                                      ViInt32 structVersion, OdiConsumerCapability *capability);

#define ODI_CONSUMER_CAPABILITY_VERSION 1 
struct OdiConsumerCapability {
    ViChar name[64];
    ViChar version[64];
    OdiPacketFormat packetFormats[64]; // list of data formats supported. 0-terminated
    Vita49ClassId classIds[64]; // list of VITA-49 formats supported. 0-terminated
    Vita49ContextClassId contextClassIds[64]; // list of Context packets supported. 0-terminated
    OdiTimestampFormat timestampFormats[64]; // list of timestamps supported. 0-terminated
    ViBoolean hasLinkChannelFilter; // true if can support linkChannelFilter
    ViUInt32 maxPacketSize; // maximum packet size in bytes
}

SCPI Interface

ODI:CONSumer[N]:CAPlability:CLIDs? -> <list of integers values>, e.g. ”IQ8BIT1CH”, ”IQ16BIT1CH”
ODI:CONSumer[N]:CAPlability:LFILter? -> 0|1
ODI:CONSumer[N]:CAPlability:NAME? -> ”<string>”
ODI:CONSumer[N]:CAPlability:PFORmats? <list of enum values>, e.g. NHEader,IDATa,CONText,ONCE
ODI:CONSumer[N]:CAPlability:VERSion? -> ”<string>”
ODI:CONSumer[N]:CAPlability:MAXPacket?
ODI:PRODucer[N]:CAPlability:TIMestamp:TFORmats? -> <list of enum values>, e.g. NOTimestamp,GPS,RELative,SAMPlecount,UTC

Remarks

This operation queries a capability of the data consumer stream.

The returned lists of PacketFormats and Vita49ClassIds does not imply all combinations are supported. Call IsFormatSupported() to determine support for a combination.
HasLinkChannelFilter indicates the data consumer has the capability to filter received packets based on Interlaken channel number.

4.5.9 Odi.Consumers[Name].IsFormatSupported( packetFormat, classId)

**Purpose**

Returns true if the combination of OdiPacketFormat and binary data format (ClassId) is supported.

**.Net Interface**

```csharp
Boolean IsFormatSupported(OdiPacketFormatEnum PacketFormat, Vita49ClassIdEnum Vita49ClassId);
```

**C Interface**

```c
ViStatus Driver_OdiConsumerIsFormatSupported(ViSession s, OdiPacketFormatEnum PacketFormat, Vita49ClassIdEnum Vita49ClassId, ViBoolean *supported);
```

**SCPI Interface**

```c
ODI:CONSumer[N]:IFSupported? <packetFormat>, <classId> -> 0|1
```

4.5.10 Odi.Consumers[Name].Activate( linkChannel, packetFormat, classId, ...)  

**Purpose**

Configure the data Consumer stream and activate it for use.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linkChannel</td>
<td>IN</td>
<td>Int32</td>
<td>Specifies filtering of packets by Interlaken channel. If set the stream accepts only packets with matching channel number. -1 to disable.</td>
</tr>
<tr>
<td>packetFormat</td>
<td>IN</td>
<td>OdiPacketFormat enum</td>
<td>Specifies expected format of received packets. Select No header, VITA-49 header, or other.</td>
</tr>
<tr>
<td>classId</td>
<td>IN</td>
<td>Vita49ClassId enum</td>
<td>Specifies expected binary data format of received packet payload, using VITA-49 Class Id as defined in VITA-49.2 and the ODI data format standard.</td>
</tr>
<tr>
<td>timestampFormat</td>
<td>IN</td>
<td>OdiTimestamp Format enum</td>
<td>Specifies expected format of the received timestamps in the VITA-49 prolog.</td>
</tr>
</tbody>
</table>
ODI - A Application Programming Interface

... | IN | Additional instrument-specific settings for properties such as packet length.

Exceptions/Errors

"Not Supported" if the stream does not support the specified settings.

"In Use" if the stream or resources are already in use.

.Net Interface

```c
void Activate(Int32 linkChannel, OdiPacketFormat packetFormat, Vita49ClassId classId, OdiTimestampFormat timestampFormat, ...);
```

C Interface

```c
ViStatus Driver_OdiConsumerActivate(ViSession s, Int32 linkChannel, OdiPacketFormat packetFormat, Vita49ClassId classId, OdiTimestampFormat timestampFormat, ...);
```

SCPI Interface

```
ODI:CONSumer[N]:ACTivate <linkChannel>, <packetFormat>, <classId>, ...
ODI:CONSumer[N]:ACTivate? -<linkChannel>, <packetFormat>, <classId>, ...
```

Remarks

Configures and activates the Consumer stream. On return the stream is ready to flow data, but in most case another instrument-specific method must be called to start the data destination so it accepts data.

Calling this method may result in reconfiguration of an FPGA in the instrument, which may take several seconds.

Although the ClassId of the expected binary data is specified as a parameter, ContextClassId is not a parameter because the consumer is expected to accept all of the context packet types it is capable of interpreting.

Due to SCPI limitations, 64-bit enum values such as CLIDs are handled as strings.

4.5.11 Odi.Consumers[Name].Deactivate()

Purpose

Deactivate the stream, stop any data flow, and free resources.

.Net Interface

```c
void Deactivate();
```

C Interface

```c
ViStatus Driver_OdiConsumerDeactivate(ViSession s, ViString name);
```
SCPI Interface

ODI:CONsumer[N]:DEACtivate