VTEXSYSTEM DRIVER

PROGRAMMER'S MANUAL

P/N: 82-0125-000
Released July 30, 2010

VTI Instruments Corp.

2031 Main Street
Irvine, CA 92614-6509
(949) 955-1894

TABLE OF CONTENTS

PROGRAMMING EXAMPLES

CERTIFICATION

VTI Instruments Corp. (VTI) certifies that this product met its published specifications at the time of shipment from the factory.

WARRANTY

The product referred to herein is warranted against defects in material and workmanship for a period of one year from the receipt date of the product at customer's facility. The sole and exclusive remedy for breach of any warranty concerning these goods shall be repair or replacement of defective parts, or a refund of the purchase price, to be determined at the option of VTI.

VTI warrants that its software and firmware designated by VTI for use with a product will execute its programming when properly installed on that product. VTI does not however warrant that the operation of the product, or software, or firmware will be uninterrupted or error free.

LIMITATION OF WARRANTY

The warranty shall not apply to defects resulting from improper or inadequate maintenance by the buyer, buyer-supplied products or interfacing, unauthorized modification or misuse, operation outside the environmental specifications for the product, or improper site preparation or maintenance.

VTI Instruments Corp. shall not be liable for injury to property other than the goods themselves. Other than the limited warranty stated above, VTI Instruments Corp. makes no other warranties, express or implied, with respect to the quality of product beyond the description of the goods on the face of the contract. VTI specifically disclaims the implied warranties of merchantability and fitness for a particular purpose.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software clause in DFARS 252.227-7013.

VTI Instruments Corp.
2031 Main Street
Irvine, CA 92614-6509 U.S.A.

SUPPORT RESOURCES

Support resources for this product are available on the Internet and at VTI Instruments customer support centers.

VTI Instruments Corp.
World Headquarters

VTI Instruments Corp.
2031 Main Street
Irvine, CA 92614-6509

Phone: (949) 955-1894
Fax: (949) 955-3041

VTI Instruments
Cleveland Instrument Division

5425 Warner Road
Suite 13
Valley View, OH 44125

Phone: (216) 447-8950
Fax: (216) 447-8951

VTI Instruments
Lake Stevens Instrument Division

3216 Wetmore Avenue, Suite 1
Everett, WA 98201

Phone: (949) 955-1894
Fax: (949) 955-3041

VTI Instruments, Pvt. Ltd.
Bangalore Instrument Division

642, 80 Feet Road
Koramangala IV Block
Bangalore – 560 034
India

Phone: +91 80 4040 7900
Phone: +91 80 4162 0200
Fax: +91 80 4170 0200

Technical Support

Phone: (949) 955-1894
Fax: (949) 955-3041
E-mail: support@vtiinstruments.com

Visit http://www.vtiinstruments.com for worldwide support sites and service plan information.

SECTION 1

INTRODUCTION

BACKGROUND

The VTEXSystem driver is used to control the IVILxiSync layer of an LXI device as well as some functions belonging to the device controller. The VTEXSystem driver does not take measurements and has no triggerable actions as defined by the LXI standard. As such, many devices that utilize the VTEXSystem driver are classified as “bridge devices”, where routing triggers via the controller board is the major focus.

The intent of this programmer’s manual is to describe the IVI-compliant VTEXSystem driver and to introduce its concepts, structure, and capabilities to software and test application engineers by providing examples of recommended code usage. The reader is expected to be familiar with instrumentation drivers, in particular IVI-COM and IVI-C specifications; COM and C programming terminology; and instrumentation concepts. Understanding IVI driver specifications will significantly help the reader follow the VTEXSystem driver’s design and the code examples provided. For more information regarding the required and optional parts of IVI-compliant switch drivers as well as IVI driver capabilities and operation, please refer to *IVI-4.15: LxiSync Class Specification* and *IVI-3.2, Inherent Capabilities Specification* which are available on the IVI Foundation website.

To further facilitate the use and understanding of the VTEXSystem driver, programming examples are provided when the driver is installed at the following location:

<Hard Drive>\Program Files\IVI Foundation\IVI\Drivers\VTEXSystem\Examples.

To further facilitate the use and understanding of the VTEXSystem driver, programming examples are provided when the driver is installed at the following location:

The driver is laid out in several sections, corresponding to LXI/IVILxiSync functionality.

- **Arm**: This interface allows the user to control the portion of the LXI Trigger model that controls the Arming of devices. Not all LXI devices support this functionality.
- **EventLog**: Functions for interfacing with the LXI Event Log. The LXI Event Log is a useful debugging tool when dealing with the LXI Trigger Model or LAN Events. For more information on the LXI Event Log, please refer to the LXI specification.
- **Events**: The IVILxiSync specification provides for an interface to handle output events from a device or for simple routing of triggers within a device. For maximum cross-platform compatibility, use the Events interface. However, if only VTI devices being used in a system, the Route interface in InstrumentSpecific can be considered as a more powerful alternative to the Event interface.
- **InstrumentSpecific**: Interfaces that are not defined by IVI are placed in this interface. VTI offers several methods and interfaces under this heading. Methods listed under this heading can be found in the InstrumentSpecific section of this manual.
    - **IOPorts**: There are times, especially when debugging, when it is useful to know the state of various hardware lines. The IOPorts interface provides access to the state a user is

trying to drive into the hardware lines (Driven State) as well as the state actually appearing on these lines (Input State). Note, these two values may be different if the user's DriveMode is Off or if the users value is being overridden in WiredOr mode.

- **Route**: A powerful routing interface that allows the user to route hardware lines in a complex manner. The Route interface is discussed more thoroughly in the *Routing* interface section of this manual.

- **Time**: LXI Class A and B instruments offer functions in this interface that allow the user to interact with the Precision Time Protocol (PTP) on the device.
- **Trigger**: This interface allows the user to control the portion of the LXI Trigger model that has to do with the Triggering of devices. Not all LXI devices support this functionality.

TRIGGERS AND ROUTING

The primary function of the VTEXSystem driver is allow users to trigger a device, whether from a software signal, a hardware line state-change, or a LAN event from a device on the network. In the case of the EX7000 series products, the mainframe itself has triggerable actions and components that the VTEXSystem driver can directly affect. This is addressed by the VTEXSystem driver in the Trigger interface.

The EX1200 mainframe, however, does not have this same capability, as the mainframe acts primarily as a bridge between the host PC and the plug-in modules that are installed. This difference in hardware is the primary reason why the EX7000 and EX1200 chassis expose different functionality the VTEXSystem driver. To accommodate trigger signals, the EX1200 contains several backplane lines which can be routed to its various plug-in module. This routing of trigger signals is addressed by the VTEXSystem driver's Route interface.

EX1200 System Driver		
Interface	**Supported Sources/Destinations**	**Supported Alarms**
Arm	None	None
Trigger	None	None
Routing	LXI0-7, DIO0-7, LAN0-7, BPL0-7	ALARM0

EX1200 Scanner Driver		
Interface	**Supported Sources/Destinations**	**Supported Alarms**
Arm	LXI0-7, LAN0-7, DIO0-7, Immediate (SW), Software (SW)	ALARM0
Trigger	LXI0-7, LAN0-7, DIO0-7, Immediate (SW), Software (SW)	ALARM0 (Same as Arm ALARM0)
Routing	None	None
Events	LXI0-7, LAN0-7, DIO0-7, ScanStepComplete (SW), ScanListComplete (SW)	None

EX7000 System Driver		
Interface	**Supported Sources/Destinations**	**Supported Alarms**
Arm	None	None
Trigger	LXI0-7, DIO0-7, LAN0-7	ALARM0
Routing	LXI0-7, DIO0-7, LAN0-7	None
Events	LXI0-7, DIO0-7, LAN0-7, OperationComplete (SW), Settling (SW), Sweeping (SW), WaitingForTrigger (SW), ScanAdvanced (SW)	ALARM0 (Same as Trigger ALARM0)

NOTE Signal with an "SW" designation are software signals and other signals cannot be routed to them. These signals can only be used as inputs (listen-only).

GLOSSARY

Throughout this document, the following terms will be used:

Controller	a single-board computer, hosting the CPU, RAM, Flash, real time software (i.e. firmware) and other devices that enable its operation as an intelligent, LXI platform.
EX-based	VTI Instruments Corp. modular instruments developed for the EX platforms. Synonymously used with “Next Generation System” and “system”.
IVI	acronym for **I**nterchangeable **V**irtual **I**nstruments; a collection of specifications that create a common programming model for several classes of instruments.
Module	any instrument installed in an EX system slot.
Module ID	a string identifying the module; “ex1200-3048”, for example.
Repeated capability	An IVI / COM software construct used to describe a group of similar features supported by an instrument. See *IVI-3.1, Driver Architecture specification* for a complete description of repeated capabilities.
SFP	acronym for **S**oft **F**ront **P**anel; an application running on a host computer (either a Windows or Linux PC) that provides an graphic user interface (GUI) to monitor and control the instrument.

BASIC CONCEPTS

1) The driver complies with the IVILxiSync Class A specification. On Windows platforms, it supports both IVI-COM and IVI-C interfaces. On Linux platforms, it supports a C++ programmatic interface that is nearly identical to the COM interface.
2) The VTEXSystem driver was designed to work equally well with all LXI instruments made or envisioned by VTI.
3) The IVI specification allows for extensions. The VTEXSystem driver includes VTI Instruments Corp. value-added methods and properties in the InstrumentSpecific interface. See the Instrument Specific Interface section for more details.

INITIALIZATION

The resource string of the IVI Initialize method identifies all EX platforms by their IP address or hostname. For an overview of the standard options available to IVI drivers, a description of resource strings and the Initialize method's syntax, please refer to the IVI Foundation's *IVI-3.2, Inherent Capabilities Specification*. The VTEXSystem driver provides additional options, as part of the Option Strings discussed on page 11. For the code examples in this document, the application program (if written in C++/COM) needs to instantiate a copy of the driver, using the syntax listed below.

Programming Examples

IVI-C Initialization

```
//
//    Sample IVI-C source code.
//

#include "stdafx.h"
#include "VTEXSystem.h"

#define ERROR_MESSAGE_SIZE     (500)
static ViSession              iviSession = 0;

void printError (ViStatus status)
{
      ViStatus  localStatus;
      ViChar    errorMessage[ERROR_MESSAGE_SIZE];

      localStatus = VTEXSystem_GetError (iviSession,
                                         &localStatus,
                                         ERROR_MESSAGE_SIZE,
                                         errorMessage);
      printf ("Runtime error: %s \nType any key to continue : ", errorMessage);
      getchar();
      exit (-1);                              // Fix the error before continuing
}


int _tmain(int argc, _TCHAR* argv[])
{
      ViStatus  status = VI_SUCCESS;

      status = VTEXSystem_InitWithOptions ("TCPIP::10.1.16.201::INSTR",     // Resource string
               VI_TRUE,                                                     // Perform an ID Query
               VI_FALSE,                                                    // Do not reset it
               "Simulate=False, QueryInstrStatus=True",                     // Session options
               &iviSession);                                                // Session handle
      if (status < VI_SUCCESS)
      {
        printError (status);
      }
      //
      //       ... use the instrument ...
      //

      return(0);
}
```

IVI-COM Initialization

```
//
//    Sample IVI-COM source code.
//

#include "stdafx.h"
#include <atlstr.h>

int _tmain(int argc, _TCHAR* argv[])
{

      //
      // Start a COM session
      //
      ::CoInitialize(NULL);

      try
      {
          //
          //    Instantiate the driver, obtain a pointer to this object.
          //
           IVTEXSystemPtr driver(__uuidof(VTEXSystem));

          //
          //    Open a session to the platform
          //
          try
          {
            driver->Initialize ("TCPIP::10.1.16.201::INSTR",               // Resource string
                                VARIANT_TRUE,                              // Perform an ID Query
                                VARIANT_FALSE,                             // Do not reset it
                                "Simulate=False, QueryInstrStatus=True");  // Session options
            //
            //  ... use the instrument ...
            //
          }
          catch (_com_error& error1)
          {
            //
            // ... handle the Initialize error ...
            //
          }
      }
      catch (_com_error& error2)
      {
        //
        //      ... handle the driver instantiation error ...
        //
      }

      ::CoUninitialize();
      return (0);
}
```

Linux C++ Initialization

```
//
//    Sample Linux C++ source code.
//

#include "libSystem.h"

int main (int argc, char** argv)
{
      try
      {
          //
          //    Instantiate the driver, obtain a pointer to this object.
          //
           LibSystem* driver = LibSystem::Create();

          //
          //    Open a session to the platform
          //
          try
          {
            driver->Initialize ("TCPIP::10.1.16.201::INSTR",              // Resource string
                                true,                                     // Perform an ID Query
                                false,                                    // Do not reset it
                                "Simulate=False, QueryInstrStatus=True");    // Session options
            //
            //  ... use the instrument ...
            //
          }
          catch (VTEXException& error1)
          {
            //
            // ... handle the Initialize error ...
            //
          }

          delete(driver);
      }
      catch (VTEXException& error2)
      {
        //
        //     ... handle the driver instantiation error ...
        //
      }
      return (0);
}
```

OPTION STRINGS

The VTEX drivers provide option strings that can be used when Initializing an instrument. The option string values exist to change the behavior of the driver. The following options strings are available on VTI IVI drivers:

- **Simulate**: Allows the user to run a program without commanding switch card or instruments. This option is useful as a debugging tool.
- **Cache**: Per the IVI specification, this option "specifies whether or not to cache the value of attributes." Caching allows IVI drivers to maintain certain instrument settings to avoid sending redundant commands. The standard allows for certain values to be cached always or never. In VTI IVI-drivers, all values used are of one of these types. As such, any values entered have no effect.
- **QueryInstrumentStatus**: Queries the instrument for errors after each call is made. As implemented in the VTI IVI drivers, instruments status is always queried regardless of the value of this property.
- **DriverSetup**: Must be last, and contains the following properties:

  - This option is applicable to the VTEXSwitch driver and allows the user to switch individual relays. This is further discussed in the *IndividualRelayMode* discussion below.
  - **Logfile**: Allows the user to specify a file to which the driver can log calls and other data.
  - **Logmode**: Specifies the mode in which the log file is opened. The allowed modes are:
    - **w**: truncate s the file to zero length or creates a text file for writing.
    - **a**: opens the file for adding information to the end of the file. The file is created if it does not exist. The stream is positioned at the end of the file.
  - **LogLevel**: Allows the user to determine the severity of a log message by providing a level-indicator to the log entry.
  - **Slots**: This is the most commonly used option and it allows for a slot number or a slot number and a card model to be specified. This option is further discussed in the *Slots* discussion below..
    "Slots=(2)" - Just slot 2.
    "Slots=(2=EX1200_3048)" - slot and card model
    "Slots=(2,3)" - Multiple slots
- **InterchangeCheck**: Boolean option that enables/disables IVI Interchangeability checking. As implemented in the VTI IVI drivers, values entered for this property have no effect.
- **RangeCheck**: Boolean option that enables or disables driver validation of user-submitted values. As implemented in the VTI IVI drivers, validation of user inputs is always performed at the firmware level regardless of this property's value.
- **RecordCoercions**: Boolean option that enables driver recording of coercions. As implemented in the VTI IVI drivers, coercions are handled in the firmware and cannot be recorded.

SECTION 2

VTEXSYSTEM DRIVER INTERFACES

The LXISync Class A specification targets an individual, intelligent instrument whose built-in firmware fulfills a few roles:

a) Monitors and controls the instrument's primary function e.g. switching, generating waveforms, measuring signals, etc.
b) Implements the arm / trigger state machine described by the LXISync specification.
c) Can log events and provide them to a host upon request.
d) Maintains IEEE 1588-compliant time.
e) Can communicate with other LXI Class A or B compliant instruments using LAN Events or the LXI Trigger Bus.

In contrast, the VTEXSystem driver targets a more modular set of instruments, some of which are LXI Class A instruments and others are LXI Class A multifunction mainframes. As such, some LXISync concepts like the arm / trigger model do not apply in some cases; others such as IEEE 1588 time and LAN Events are always present. In some devices, there is an additional need for platform specific functionality, in particular routing triggers and events between external hosts and the instruments on the platform.

The remainder of *Section 2* describe how the driver implements each of these capabilities.

UNSUPPORTED APIS IN VTEXSYSTEM

VTEXSystem implements an IVILxiSync interface which requires all properties available in the API to be present. It is, however, permissible to return "Not supported" for APIs that the instrument itself does not have available. There are several APIs where this is the case.

Add()

VTI Instruments products do not support dynamically adding repeated capabilities. These are automatically discovered and populated at runtime from the hardware capabilities of the unit.

Remove()

Since the Add API is not supported, it is unnecessary to support the Remove API.

RemoveAllCustom<RepeatedCapabilityCollectionIdentifier>()

This is not supported for the same reason as the Add and Remove APIs.

Arm/Arm Alarms/Arm Sources

All of the properties and methods which are not at the Repeated Capability Collection level under Arm, Arm Sources, or Arm Alarms, will currently return Unimplemented. There are no devices VTEXSystem supports right now that support Arm logic, though this could be changed in a future software update. This does not, however, apply to the Arm properties in VTEXScanner.

Display()

This function, under InstrumentSpecific, is currently reserved for future expansion. In the future, it may allow a user to display data on the front panel of a device.

RetrieveFile()

This function, under InstrumentSpecific, is currently reserved for future expansion. Currently, there are no user-viewable files on plug-in cards that would require this interface.

SystemInventory

This property allows a user who has a VTI multifunction mainframe or bridge device to query the contents of their plug-in slots. This call may return unsupported on some devices which do not have pluggable modules.

INSTRUMENT SPECIFIC INTERFACE

This interface provides IVI-compliant methods and properties for controlling the platform's specific operations. It provides three functionality groups:

1. Utility functions and information.
2. Parallel access to the platform's I/O ports, useful for debugging or reading a system state snapshot.
3. Routing signals and triggers from / to the platform's various sources and destinations. The available sources and destinations are:
   - LXI trigger bus lines
   - Backplane lines
   - DIO lines
   - LAN Events
   - Alarms (Note, Alarms can only be set as sources)

Most of the instrument specific functionality is implemented through the use of repeated capabilities.

NOTE	Not all signals are available on all platforms. To determine what signals are supported by a particular platform, please refer to the *Triggers and Routing* discussion in *Section 1*.

UTILITY FUNCTIONS AND INFORMATION

EventLogOverflowMode

The Event Log Overflow Mode is a property controlling the behavior of the LXI LAN Event Log (see the EventLog section for details). The event log buffer is a fixed size, and may have one of two behaviors when the number of log entries reaches the size of the buffer. In the VTEXSystemEventLogOverflowStop mode, the last entry in the log will have “Records Missing” inserted, and the log will stop accepting data until some event log entries are read. In the VTEXSystemEventLogOverflowOverwrite mode, the log will act as a circular buffer, with a “Records Missing” entry placed just past the latest write, separating the old from the new data.

SerialNumber

This read-only property contains the connected mainframe’s VTI serial number.

SystemInventory

All devices that support the System driver support retrieving a list of the plug-in modules installed in a chassis. If implemented, this call returns data in the form of a comma-separated string. If additional Model number and serial number for that module is available, it will be presented in the form “ModuleType:Model:Serial”. If the slot does not have a plug-in card in it, the slot will be presented as “NOT POPULATED”.

Programming Examples

IVI-C Instrument Specific

```
ViChar serial[500];
VTEXSystem_GetAttributeViString(vi, "", VTEXSYSTEM_ATTR_SERIAL_NUMBER, 500, serial);
//Set to stop filling the log on overflow
status = VTEXSystem_SetAttributeViInt32(vi, "", VTEXSYSTEM_ATTR _EVENT_LOG_OVERFLOW_MODE,
     VTEXSystemEventLogOverflowStop);
```

IVI-COM InstrumentSpecific

```
BSTR serial = driver->InstrumentSpecific->SerialNumber;
//Set to stop filling the log on overflow
driver->InstrumentSpecific->EventLogOverflowMode = VTEXSystemEventLogOverflowStop;
```

Linux C++ InstrumentSpecific

```
std::string serial = driver->InstrumentSpecific->SerialNumber;
//Set to stop filling the log on overflow
driver->InstrumentSpecific->EventLogOverflowMode = VTEXSystemEventLogOverflowStop;
```

PARALLEL ACCESS TO I/O PORTS

IOPorts Repeated Capability Collection

Note that since this is not an IVI defined interface, functions such as Add, Remove, etc. were excluded as they are unimplemented in VTEXSystem.

- ListOfIOPorts
- Count
- Item
- Name

IOPort Repeated Capability

- InputState
- DrivenState

InputState

Data Type	Access	Applies to	Coercion	High Level Functions
ViInt32	RO	N/A	N/A	N/A

COM Property Name

```
InstrumentSpecific.IOPorts.Item().InputState
```

COM Enumeration Name

```
N/A
```

C Constant Name

```
VTEXSYSTEM_ATTR_IOPORT_INPUT_STATE
```

Description

The InputState of an IOPort is a bitmask indicating the state of all of the hardware or LAN lines, as read by the device. This value should correspond to the actual state of the lines.

DrivenState

Data Type	Access	Applies to	Coercion	High Level Functions
ViInt32	RO	N/A	N/A	N/A

COM Property Name

```
InstrumentSpecific.IOPorts.Item().DrivenState
```

COM Enumeration Name

N/A

C Constant Name

VTEXSYSTEM_ATTR_IOPORT_DRIVEN_STATE

Description

The DrivenState of an IOPort is a bitmask indicating the internal state of the device. If the output enables are turned to Driven, this should correspond to the InputState seen above. If it does not, there may be two devices attempting to drive the line at once, or the instrument may be in WiredOr mode. See the LXI Specification for more on WiredOr Mode.

Programming Examples

IVI-C IOPorts

```
//Another way of calculating Enabled
ViInt32 InputState;
ViInt32 DrivenState;
ViInt32 enables;
status = VTEXSystem_GetAttributeViInt32(vi, "LXI", VTEXSYSTEM_ATTR_ IOPORT_INPUT_STATE,
     &InputState);
status = VTEXSystem_GetAttributeViInt32(vi, "LXI", VTEXSYSTEM_ATTR_IOPORT_DRIVEN_STATE,
     &DrivenState);
enables = InputState & DrivenState;

/* The above works best as a verification mechanism to show that the state of the lines matches
     what DriveModes is reporting. However, this will not be accurate in WiredOr mode. */
```

IVI-COM IOPorts

```
//Another way of calculating Enabled
long InputState = driver->InstrumentSpecific->IOPorts->Item[“LXI”]->InputState;
long enabled_sources = InputState & driver->InstrumentSpecific->IOPorts->Item[“LXI”]-
     >DrivenState;

/* The above works best as a verification mechanism to show that the state of the lines matches
     what DriveModes is reporting. However, this will not be accurate in WiredOr mode.*/
```

Linux C++ IOPorts

```
//Another way of calculating Enabled
long InputState = driver->InstrumentSpecific->IOPorts->Item[“LXI”]->InputState;
long enabled_sources = InputState & driver->InstrumentSpecific->IOPorts->Item[“LXI”]-
     >DrivenState;

/* * The above works best as a verification mechanism to show that the state of the lines matches
     what DriveModes is reporting. However, this will not be accurate in WiredOr mode.*/
```

ROUTING

To supplement the IviLxiSync Event class, the VTEXSystem driver provides Route class calls which expand a user's ability to control triggers and hardware lines. Similar to the way that the Arm capabilities function as a multiplexer to logically AND or OR signals together to create a state that causes an Arm event to occur, the Route interface function provides a similar multiplexing ability, but uses the line states themselves. For example, if a `SourcesList` of "LXI2,LAN3" and `InvertedSourcesList` of "DIO2" were to be assigned to the Destination BPL2, and OrEnabled were set to False, the resulting state of BPL2 could be represented as follows: BPL2 = LXI2 & LAN3 & ~DIO2. If `OrEnabled` were then set to True, the representation would change to be BPL2 = LXI2 | LAN3 | ~DIO2. Using this class, the user can utilize software controls and route a multiple sources to a single destination. Note, however, that not all Destinations support multiple sources.

This multiplexing functionality also allows the Route interface to convert trigger types. As was noted in the introduction, devices utilizing the VTEXSystem driver are often LXI bridge devices. If it was required to have a 5 V TTL signal from one device trigger another device that only has an LXI Trigger Bus, the signal could be routed through an EX1200 from the DIO lines to the LXI lines, utilizing the mainframe as a bridge between these two devices. This could not be accomplished using the Event class calls alone.

Note that, although similar to Arm capabilities in format, Route class calls have no bearing on arming the instrument or progressing through the trigger model.

Route Alarms

Route Alarms offer the same functionality as Arm Alarms, and on many platforms are the same physical hardware. No warning will be given by the API before data is overwritten.

The Route Alarms Repeated Capability Collection and Route Alarm Repeated Capabilities offer setup for alarms before routing their signals to their eventual destinations.

RouteAlarms Repeated Capability Collection

- Add()
- DisableAll()
- Remove()
- RemoveAllCustomRouteAlarms()
- ListOfRouteAlarms
- Count
- Item (Not in IVI-C)
- Name

RouteAlarm Repeated Capability

- Configure()
- Enabled
- Period
- RepeatCount
- TimerFraction
- TimeSeconds

Configure

Description

This is a convenience function which allows a user to configure most aspects of a Route Destination. The Software State and OrEnabled properties are not controllable via this interface.

Namespace

VTI.VTEXSystem.Interop

Assembly

VTI.VTEXSystem.Interop (in VTI.VTEXSystem.Interop.dll)

COM Prototype

```
HRESULT Configure(
   BSTR SourcesList,
   BSTR InvertedSourcesList,
   VARIANT_BOOL OrEnabled,
   VTEXSystemEventDriveModeEnum DriveMode,
   BSTR DestinationPath
```

C Prototype

```
VTEXSystem_InstrumentSpecificRouteDestinationConfigure ( ViSession Vi,
ViConstString repCapIdentifier, ViConstString SourcesList, ViConstString
InvertedSourcesList, ViBoolean OrEnabled, ViInt32 DriveMode, ViConstString
DestinationPath );
```

Parameters

Inputs	Description	Data Type
SourcesList	The comma-separated list of route sources for this destination. These source signals will be combined to create the destination signal.	ViConstString
InvertedSourcesList	The comma-separated list of inverted route sources for this destination. These source signals will be complemented and then combined to create the destination signal.	ViConstString
OrEnabled	Specifies how the soruces for this route destination (both normal and inverted) will be combined to yield the resulting signal. If TRUE, the sources will be OR'ed together, else they will be AND'ed together.	ViBoolean
DriveMode	Specifies how this event is transmitted when the route's sources cause such a transmission.	ViInt32
DestinationPath	Comma-separated list of route destinations.	ViConstString

Configure

Data Type	Access	Applies to	Coercion	High Level Functions
BSTR	R/W	N/A	None	None

COM Property Name

`List`

COM Enumeration Name

N/A

C Constant Name

`VTEXSCANNER_ATTR_LIST`

Description

This property has identical functionality to the same property under *ArmAlarms.*

Enabled

Data Type	Access	Applies to	Coercion	High Level Functions
ViBoolean	RW	N/A	N/A	N/A

COM Property Name

`InstrumentSpecific.Route.Alarms.Item().Enabled`

COM Enumeration Name

`N/A`

C Constant Name

`VTEXSYSTEM_ATTR_ROUTE_ALARM_ENABLED`

Description

This property has identical functionality to the same property under *ArmAlarms*.

Period

Data Type	Access	Applies to	Coercion	High Level Functions
ViReal64	RW	N/A	N/A	N/A

COM Property Name

```
InstrumentSpecific.Route.Alarms.Item().Period
```

COM Enumeration Name

```
N/A
```

C Constant Name

```
VTEXSYSTEM_ATTR_ROUTE_ALARM_PERIOD
```

Description

This property has identical functionality to the same property under *ArmAlarms*.

RepeatCount

Data Type	Access	Applies to	Coercion	High Level Functions
ViInt32	RW	N/A	N/A	N/A

COM Property Name

```
InstrumentSpecific.Route.Alarms.Item().RepeatCount
```

COM Enumeration Name

```
N/A
```

C Constant Name

```
VTEXSYSTEM_ATTR_ROUTE_ALARM_REPEAT_COUNT
```

Description

This property has identical functionality to the same property under *ArmAlarms*.

TimeFraction

Data Type	Access	Applies to	Coercion	High Level Functions
ViReal64	RW	N/A	N/A	N/A

COM Property Name

```
InstrumentSpecific.Route.Alarms.Item().TimeFraction
```

COM Enumeration Name

```
N/A
```

C Constant Name

VTEXSYSTEM_ATTR_ROUTE_ALARM_TIME_FRACTION

Description

This property has identical functionality to the same property under *ArmAlarms.*

TimeSeconds

Data Type	Access	Applies to	Coercion	High Level Functions
ViReal64	RW	N/A	N/A	N/A

COM Property Name

InstrumentSpecific.Route.Alarms.Item().TimeSeconds

COM Enumeration Name

N/A

C Constant Name

VTEXSYSTEM_ATTR_ROUTE_ALARM_TIME_SECONDS

Description

This property has identical functionality to the same property under *ArmAlarms.*

Programming Examples

IVI-C IOPorts

```
/* Route all signals from LAN3 to LXI2 as was done in the first Events Interface example above */
driver->InstrumentSpecific->Route->Destinations->Item[“LXI2”]->SourcesList = “LAN3”;
driver->InstrumentSpecific->Route->Destinations->Item[“LXI2”]->DriveMode =
      VTEXSystemEventDriveModeDriven;

/* Route and invert all signals from LAN3 to LXI2 as was done in the second Events Interface
      example. Note that the SourcesList must be cleared because LXI lines do not support
      multiple sources. */
driver->InstrumentSpecific->Route->Destinations->Item[“LXI2”]->SourcesList = “”;
driver->InstrumentSpecific->Route->Destinations->Item[“LXI2”]->InvertedSourcesList = “LAN3”;
driver->InstrumentSpecific->Route->Destinations->Item[“LXI2”]->DriveMode =
      VTEXSystemEventDriveModeDriven

/* Toggle DIO3 for 5 seconds. This cannot be done using the Events interface. Note that the
      SourcesList and InvertedSourcesList must be empty for software control to work. */
//If the line is undriven, changing its state will have no effect.
driver->InstrumentSpecific->Route->Destinations->Item[“DIO3”]->DriveMode =
      VTEXSystemEventDriveModeDriven
driver->InstrumentSpecific->Route->Destinations->Item[“DIO3”]->SoftwareState = 1;
//Use the operating system’s “Sleep” command here
Sleep(5000);
driver->InstrumentSpecific->Route->Destinations->Item[“DIO3”]->SoftwareState = 0;
```

```
/* When multiple sources are assigned to a line, the lines are logical-ANDed together by default,
      just like Arm sources. Note that it is possible to have multiple sources in each
      SourcesList if needed.*/
driver->InstrumentSpecific->Route->Destinations->Item[“BPL2”]->SourcesList = “LXI2, DIO3”;
driver->InstrumentSpecific->Route->Destinations->Item[“BPL2”]->InvertedSourcesList = “LAN3”;
//Change BPL2 to a logical OR.
driver->InstrumentSpecific->Route->Destinations->Item[“BPL2”]->OrEnabled = VARIANT_TRUE;
```

ROUTE DESTINATIONS

Route Destinations are the endpoints of a signal routing. Each routing is constructed by assigning Sources, Alarms, and/or internal software events to the SourcesList and/or InvertedSourcesList. These events are documented in the *Triggers and Routing* discussion in *Section 1*.

Some devices support allowing multiple sources to be combined for a single destination, in which case SourcesList and InvertedSourcesList could both be populated, and either list could also contain multiple event sources. Refer to the *Triggers and Routing* discussion in *Section 1* to determine if the device has any Route Destinations which support multiple sources.

NOTES

1) Since the Routing interface is an extension of the Event interface, all Enum values in the Route Destination Repeated Capability Collection and Route Destination Repeated Capabilities use their Event counterparts.

2) Since the Routing interface is an extension of the Event interface, it is meant to be used instead of that interface. Interleaving use of the two interfaces is not recommended, and can cause unexpected behavior.

RouteDestinations Repeated Capability Collection

See *Repeated Capabilities* section in the *IVI-3.1, Driver Architecture specification* for more details.

- Add()
- DisableAll()
- Remove()
- RemoveAllCustomRouteDestinations()
- ListOfRouteDestinations
- Count
- Item (Not in IVI-C)
- Name

RouteDestination Repeated Capability

- Configure()
- Pulse()
- DestinationPath
- DriveMode
- InvertedSourcesList
- OrEnabled
- SoftwareState
- SourcesList

Configure

Description

This is a convenience function which allows a user to configure most aspects of a Route Destination. The Software State and OrEnabled properties are not controllable via this interface.

COM Prototype

```
HRESULT Configure(
   BSTR SourcesList,
   BSTR InvertedSourcesList,
   VARIANT_BOOL OrEnabled,
   VTEXSystemEventDriveModeEnum DriveMode,
   BSTR DestinationPath
```

C Prototype

```
VTEXSystem_InstrumentSpecificRouteDestinationConfigure ( ViSession Vi,
ViConstString repCapIdentifier, ViConstString SourcesList, ViConstString
InvertedSourcesList, ViBoolean OrEnabled, ViInt32 DriveMode, ViConstString
DestinationPath );
```

Parameters

Inputs	Description	Data Type
Vi	Instrument handle	ViSession

Return Values

The *IVI-3.2: Inherent Capabilities Specification* defines general status codes that this function can return.

Pulse

Description

Creates a short duration pulse on the designated route destination. Regardless of the current state of the Route Destination, this function toggles it twice

COM Prototype

```
HRESULT Pulse(
);
```

C Prototype

```
VTEXSystem_InstrumentSpecificRouteDestinationPulse ( ViSession Vi,
ViConstString repCapIdentifier );
```

Parameters

Inputs	Description	Data Type
Vi	Instrument handle	ViSession

Return Values

The *IVI-3.2: Inherent Capabilities Specification* defines general status codes that this function can return.

DestinationPath

Data Type	Access	Applies to	Coercion	High Level Functions
ViString	RW	N/A	N/A	N/A

COM Property Name

```
InstrumentSpecific.Route.Destinations.Item().DestinationPath
```

COM Enumeration Name

```
N/A
```

C Constant Name

```
VTEXSYSTEM_ATTR_ROUTE_DESTINATION_DESTINATION_PATH
```

Description

This property operates identically to the DestinationPath property in the Event interface.

DriveMode

Data Type	Access	Applies to	Coercion	High Level Functions
VTEXSystemEventDriveModeEnum	RW	N/A	N/A	N/A

COM Property Name

```
InstrumentSpecific.Route.Destinations.Item().DriveMode
```

COM Enumeration Name

```
N/A
```

C Constant Name

```
VTEXSYSTEM_ATTR_ROUTE_DESTINATION_DRIVE_MODE
```

Description

This property operates identically to the DriveMode property in the Event interface.

InvertedSourcesList

Data Type	Access	Applies to	Coercion	High Level Functions
ViString	RW	N/A	N/A	N/A

COM Property Name

```
InstrumentSpecific.Route.Destinations.Item().InvertedSourcesList
```

COM Enumeration Name

```
N/A
```

C Constant Name

```
VTEXSYSTEM_ATTR_ROUTE_DESTINATION_INVERTED_SOURCES_LIST
```

Description

If a source is placed in this list the Route Destination will be the logical invert of the source placed in the list. If there are multiple sources, they will either be logically AND'ed or OR'ed depending on the state of the OrEnabled property.

OrEnabled

Data Type	Access	Applies to	Coercion	High Level Functions
ViBoolean	RW	N/A	N/A	N/A

COM Property Name

```
Instrumentspecific.Route.Destinations.Item().OrEnabled
```

COM Enumeration Name

```
N/A
```

C Constant Name

```
VTEXSYSTEM_ATTR_ROUTE_DESTINATION_OR_ENABLED
```

Description

This property is only accessible on Route Destinations that support multiple sources. If set to true, the output of the destination will be the logical OR of the state of all the members of the SourcesList and InvertedSourcesList. If set to false, the output will be the logical AND of the same.

SoftwareState

Data Type	Access	Applies to	Coercion	High Level Functions
ViInt32	RW	N/A	N/A	N/A

COM Property Name

```
InstrumentSpecific.Route.Destinations.Item().SoftwareState
```

COM Enumeration Name

```
N/A
```

C Constant Name

```
VTEXSYSTEM_ATTR_ROUTE_DESTINATION_SOFTWARE_STATE
```

Description

The software state property is only alterable when there are no sources in either the SourcesList or InvertedSourcesList. This is a debugging aid which allows a user to manually toggle the state of the line to either “1” or “0” to cause his test system to operate.

SourcesList

Data Type	Access	Applies to	Coercion	High Level Functions
ViString	RW	N/A	N/A	N/A

COM Property Name

```
InstrumentSpecific.Route.Destinations.Item().SourcesList
```

COM Enumeration Name

```
N/A
```

C Constant Name

```
VTEXSYSTEM_ATTR_ROUTE_DESTINATION_SOURCES_LIST
```

Description

If a source is placed in this list the Route Destination will be linked to the source placed in the list, rising when it rises and falling when it falls. If there are multiple sources, they will either be logically AND’ed or OR’ed depending on the state of the OrEnabled property.

Programming Examples

IVI-C Route Destinations

```
//Setting up a single source non-inverted
status = VTEXSystem_SetAttributeViString(vi, "LAN1",
     VTEXSYSTEM_ATTR_ROUTE_DESTINATION_INVERTED_SOURCES_LIST, "");
status = VTEXSystem_SetAttributeViString(vi, "LAN1",
     VTEXSYSTEM_ATTR_ROUTE_DESTINATION_SOURCES_LIST, "DIO3");
status= VTEXSystem_SetAttributeViInt32(vi, "LAN1", VTEXSYSTEM_ATTR_ROUTE_DESTINATION_DRIVE_MODE,
     VTEXSystemEventDriveModeDriven);

/* Note that in the above example, InvertedSourcesList is cleared before SourcesList is set. This
     is because LAN sources only support single-source routing - if there had been a source in
     the InvertedSourcesList and it had not been cleared, this would have resulted in an error.
     Also note that DriveMode is enabled last, which prevents the line from being enabled with
     an unknown configuration. */

//Setting up a single source, inverted
status = VTEXSystem_SetAttributeViString(vi, "LAN1",
     VTEXSYSTEM_ATTR_ROUTE_DESTINATION_SOURCES_LIST, "");
status = VTEXSystem_SetAttributeViString(vi, "LAN1",
     VTEXSYSTEM_ATTR_ROUTE_DESTINATION_INVERTED_SOURCES_LIST, "DIO3");
status = VTEXSystem_SetAttributeViInt32(vi, "LAN1", VTEXSYSTEM_ATTR_ROUTE_DESTINATION_DRIVE_MODE,
     VTEXSystemEventDriveModeDriven);

/* The above setup indicates that LAN1 will be transmitting the inversion of DIO3's state */

//Setting up multiple sources, ANDed
status = VTEXSystem_SetAttributeViString(vi, "BPL3",
     VTEXSYSTEM_ATTR_ROUTE_DESTINATION_INVERTED_SOURCES_LIST, "DIO1,DIO2");
status = VTEXSystem_SetAttributeViString(vi, "BPL3",
     VTEXSYSTEM_ATTR_ROUTE_DESTINATION_SOURCES_LIST, "DIO3");
status = VTEXSystem_SetAttributeViBoolean(vi, "BPL3",
     VTEXSYSTEM_ATTR_ROUTE_DESTINATION_OR_ENABLED, VI_FALSE);
status = VTEXSystem_SetAttributeViInt32(vi, "BPL3", VTEXSYSTEM_ATTR_ROUTE_DESTINATION_DRIVE_MODE,
     VTEXSystemEventDriveModeDriven);

/* The above is logically equivalent to 'BPL3 = !DIO1 & !DIO2 & DIO3'. Note that BPL lines allow
     the configuration of multiple sources, if present in the device. See the Triggers and
     Routing discussion in Section 1 for more information. The configuration example above might
     be useful if DIO1 and DIO2 were 0-true logic events indicating device preparedness, and
     DIO3 were a trigger line indicating measurement start. In this case a signal would be
     transmitted on BPL3 only when all devices were ready and a start signal was received. */

//Setting up multiple sources, ORed
status = VTEXSystem_SetAttributeViString(vi, "BPL5",
     VTEXSYSTEM_ATTR_ROUTE_DESTINATION_INVERTED_SOURCES_LIST, "");
status = VTEXSystem_SetAttributeViString(vi, "BPL5",
     VTEXSYSTEM_ATTR_ROUTE_DESTINATION_SOURCES_LIST, "DIO3, DIO4, DIO5");
status = VTEXSystem_SetAttributeViBoolean(vi, "BPL5",
     VTEXSYSTEM_ATTR_ROUTE_DESTINATION_OR_ENABLED, VI_TRUE);
status = VTEXSystem_SetAttributeViInt32(vi, "BPL5", VTEXSYSTEM_ATTR_ROUTE_DESTINATION_DRIVE_MODE,
     VTEXSystemEventDriveModeDriven);

/* Note that it is not necessary to use both InvertedSourcesList and SourcesList when configuring
     multiple sources. The statements above are logically equivalent to 'BPL5 = DIO3 | DIO4 |
     DIO5'. This kind of configuration could be useful if each of DIO3, DIO4, and DIO5 were
     triggers coming from another device that needed to be listened to. */

//Manually configuring a source for debugging
status = VTEXSystem_SetAttributeViString(vi, "LAN1",
     VTEXSYSTEM_ATTR_ROUTE_DESTINATION_INVERTED_SOURCES_LIST, "");
status = VTEXSystem_SetAttributeViString(vi, "LAN1",
     VTEXSYSTEM_ATTR_ROUTE_DESTINATION_SOURCES_LIST, "");
status = VTEXSystem_SetAttributeViString(vi, "LAN1",
     VTEXSYSTEM_ATTR_ROUTE_DESTINATION_DESTINATION_PATH, "10.20.5.3");
status = VTEXSystem_SetAttributeViInt32(vi, "LAN1",
     VTEXSYSTEM_ATTR_ROUTE_DESTINATION_SOFTWARE_STATE, 0);
```

```
status = VTEXSystem_SetAttributeViInt32(vi, "LAN1", VTEXSYSTEM_ATTR_ROUTE_DESTINATION_DRIVE_MODE,
      VTEXSystemEventDriveModeDriven);
status = VTEXSystem_SetAttributeViInt32(vi, "LAN1",
      VTEXSYSTEM_ATTR_ROUTE_DESTINATION_SOFTWARE_STATE, 1);
status = VTEXSystem_InstrumentSpecificRouteDestinationPulse(vi, "LAN1");

/* In the above example, the Sources and InvertedSources lists are emptied first. Then the
      software state is set to 0 before enabling the line, in order to have a known state.
      Setting the state to 1 after changing the DriveMode will send out a RISE event on LAN1, and
      sending a Pulse message while in WiredOr mode will also ONLY send out a RISE event. The
      total output from this command set will be two LAN1 RISE messages directed to IP address
      10.20.5.3. (See LXI Specification for more on the intricacies of WiredOr mode) */

/* Note that these examples did not cover the use of Configure Route Destination, as it is fairly
      trivial. Also, be sure to check the “status” value between function calls. This can be
      automated. */
```

IVI-COM Route Destinations

```
//Setting up a single source non-inverted
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->InvertedSourcesList = “”;
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->SourcesList = “DIO3”;
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->DriveMode =
      VTEXSystemEventDriveModeDriven;

/* Note that in the above example, the InvertedSourcesList is cleared before the SourcesList is
      set. This is because LAN sources only support single-source routing - if there had been a
      source in the InvertedSourcesList and it had not been cleared, this would have resulted in
      an error. Also note that DriveMode is enabled last, which prevents the line from being
      enabled with an unknown configuration. */

//Setting up a single source, inverted
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->SourcesList = “”;
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->InvertedSourcesList = “DIO3”;
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->DriveMode =
      VTEXSystemEventDriveModeDriven;

/* The above setup indicates that LAN1 will be transmitting the inversion of DIO3’s state */

//Setting up multiple sources, ANDed

driver->InstrumentSpecific->Route->Destinations->Item[“BPL3”]->InvertedSourcesList = “DIO1,
      DIO2”;
driver->InstrumentSpecific->Route->Destinations->Item[“BPL3”]->SourcesList = “DIO3”;
driver->InstrumentSpecific->Route->Destinations->Item[“BPL3”]->OrEnabled = VARIANT_FALSE;
driver->InstrumentSpecific->Route->Destinations->Item[“BPL3”]->DriveMode =
      VTEXSystemEventDriveModeDriven;

/* The above is logically equivalent to ‘BPL3 = !DIO1 & !DIO2 & DIO3’. Note that BPL lines allow
      the configuration of multiple sources, if present in the device. See the Triggers and
      Routing discussion in Section 1 for more information. The configuration example above might
      be useful if DIO1 and DIO2 were 0-true logic events indicating device preparedness, and
      DIO3 were a trigger line indicating measurement start. In this case a signal would be
      transmitted on BPL3 only when all devices were ready and a start signal was received. */

//Setting up multiple sources, ORed
driver->InstrumentSpecific->Route->Destinations->Item[“BPL5”]->InvertedSourcesList = “”;
driver->InstrumentSpecific->Route->Destinations->Item[“BPL5”]->SourcesList = “DIO3, DIO4, DIO5”;
driver->InstrumentSpecific->Route->Destinations->Item[“BPL5”]->OrEnabled = VARIANT_TRUE;
driver->InstrumentSpecific->Route->Destinations->Item[“BPL5”]->DriveMode =
      VTEXSystemEventDriveModeDriven;

/* Note that it is not necessary to use both InvertedSourcesList and SourcesList when configuring
      multiple sources. The statements above are logically equivalent to ‘BPL5 = DIO3 | DIO4 |
      DIO5’. This kind of configuration could be useful if each of DIO3, DIO4, and DIO5 were
      triggers coming from another device that needed to be listened to. */

//Manually configuring a source for debugging
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->SourcesList = “”;
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->InvertedSourcesList = “”;
```

```
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->DestinationPath = “10.20.5.3”
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->SoftwareState = 0;
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->DriveMode =
     VTEXSystemEventDriveModeWiredOr;
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->SoftwareState = 1;
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->Pulse();

/* In the above example, the Sources and InvertedSources lists are emptied first. Then the
     software state is set to 0 before enabling the line, in order to have a known state.
     Setting the state to 1 after changing the DriveMode will send out a RISE event on LAN1, and
     sending a Pulse message while in WiredOr mode will also ONLY send out a RISE event. The
     total output from this command set will be two LAN1 RISE messages directed to IP address
     10.20.5.3. (See LXI Specification for more on the intricacies of WiredOr mode) */
```

Linux C++ Route Destinations

```
//Setting up a single source non-inverted
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->InvertedSourcesList = “”;
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->SourcesList = “DIO3”;
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->DriveMode =
     VTEXSystemEventDriveModeDriven;

/* Note that in the above example, the InvertedSourcesList is cleared before the SourcesList is
     set. This is because LAN sources only support single-source routing - if there had been a
     source in the InvertedSourcesList and had it not been cleared, this would have resulted in
     an error. Also note that DriveMode is enabled last, which prevents the line from being
     enabled with an unknown configuration. */

//Setting up a single source, inverted
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->SourcesList = “”;
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->InvertedSourcesList = “DIO3”;
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->DriveMode =
     VTEXSystemEventDriveModeDriven;

/* The above setup indicates that LAN1 will be transmitting the inversion of DIO3’s state */

//Setting up multiple sources, ANDed

driver->InstrumentSpecific->Route->Destinations->Item[“BPL3”]->InvertedSourcesList = “DIO1,
     DIO2”;
driver->InstrumentSpecific->Route->Destinations->Item[“BPL3”]->SourcesList = “DIO3”;
driver->InstrumentSpecific->Route->Destinations->Item[“BPL3”]->OrEnabled = false;
driver->InstrumentSpecific->Route->Destinations->Item[“BPL3”]->DriveMode =
     VTEXSystemEventDriveModeDriven;

/* The above is logically equivalent to ‘BPL3 = !DIO1 & !DIO2 & DIO3’. Note that BPL lines allow
     the configuration of multiple sources, if present in the device. See the Triggers and
     Routing discussion in Section 1 for more information. The configuration example above might
     be useful if DIO1 and DIO2 were 0-true logic events indicating device preparedness, and
     DIO3 were a trigger line indicating measurement start. In this case a signal would be
     transmitted on BPL3 only when all devices were ready and a start signal was received. */

//Setting up multiple sources, ORed
driver->InstrumentSpecific->Route->Destinations->Item[“BPL5”]->InvertedSourcesList = “”;
driver->InstrumentSpecific->Route->Destinations->Item[“BPL5”]->SourcesList = “DIO3, DIO4, DIO5”;
driver->InstrumentSpecific->Route->Destinations->Item[“BPL5”]->OrEnabled = true;
driver->InstrumentSpecific->Route->Destinations->Item[“BPL5”]->DriveMode =
     VTEXSystemEventDriveModeDriven;

/* Note that it is not necessary to use both InvertedSourcesList and SourcesList when configuring
     multiple sources. The statements above are logically equivalent to ‘BPL5 = DIO3 | DIO4 |
     DIO5’. This kind of configuration could be useful if each of DIO3, DIO4, and DIO5 were
     triggers coming from another device that needed to be listened to. */

//Manually configuring a source for debugging
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->SourcesList = “”;
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->InvertedSourcesList = “”;
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->DestinationPath = “10.20.5.3”
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->SoftwareState = 0;
```

```
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->DriveMode =
      VTEXSystemEventDriveModeWiredOr;
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->SoftwareState = 1;
driver->InstrumentSpecific->Route->Destinations->Item[“LAN1”]->Pulse();

/* In the above example, the Sources and InvertedSources lists are emptied first. Then the
      software state is set to 0 before enabling the line, in order to have a known state.
      Setting the state to 1 after changing the DriveMode will send out a RISE event on LAN1, and
      sending a Pulse message while in WiredOr mode will also ONLY send out a RISE event. The
      total output from this command set will be two LAN1 RISE messages directed to IP address
      10.20.5.3. (See LXI Specification for more on the intricacies of WiredOr mode) */
```

RouteSources Repeated Capability Collection

See *Repeated Capabilities* section in the *IVI-3.1, Driver Architecture specification* for more details.. Also note that since Route Sources have no Enabled or DriveMode properties, that there is no DisableAll function.

- Add()
- Remove()
- RemoveAllCustomRouteSources()
- ListOfRouteSources
- Count
- Item **(Not in IVI-C)**
- Name

RouteSource Repeated Capability

- EventID
- Filter

EventID

Data Type	Access	Applies to	Coercion	High Level Functions
ViString	RW	N/A	N/A	N/A

COM Property Name

```
InstrumentSpecific.Route.Sources.Item().EventId
```

COM Enumeration Name

```
N/A
```

C Constant Name

```
VTEXSYSTEM_ATTR_ROUTE_SOURCE_EVENT_ID
```

Description

This is identical to the Filter property under TriggerSources.

Filter

Data Type	Access	Applies to	Coercion	High Level Functions
ViString	RW	N/A	N/A	N/A

COM Property Name

```
InstrumentSpecific.Route.Sources.Item().Filter
```

COM Enumeration Name

```
N/A
```

C Constant Name

```
VTEXSYSTEM_ATTR_ROUTE_SOURCE_FILTER
```

Description

This is identical to the Filter property under TriggerSources.

Programming Examples

See examples in the Trigger Interface section.

SECTION 3

PROGRAMMING EXAMPLES

INTRODUCTION

The VTEXSystem driver uses IVI-compliant APIs to control its operation. The help file for the driver, installed with the driver and available on the product's *Distribution CD* as a standalone .chm file, provides all the needed property and method descriptions as well as the enum values. To supplement the help file and to provide a better understanding of how the APIs work together, programming examples are provided in this section. All examples provided were written in IVI-COM. For more information on standard IVI function calls, please refer to the IVI Foundation website for complete documentation. Additional programming examples are also included with the driver distribution which can be used and modified if desired.

NOTE	For Linux users, .chm viewer are available. These viewers vary from one distribution to another.

TRIGGER INTERFACE

In the trigger subsystem, there is a single property, **TriggerSource**, which can be set to any of the available trigger sources. There is also a TriggerSources area where the trigger sources are configured. Most devices do not support the Trigger subsystem.

```
/* Set an LXI line as the trigger source */
driver->Trigger->TriggerSource = "LXI1";

/* Set an alarm as the trigger source. For more information on configuring alarms, see the "Time
     & Alarms" section */
driver->Trigger->TriggerSource = "ALARM0"

/* Configure a LAN trigger source. See the LXI Specification for more information on setting LAN
     Event Filters */
//Incoming events required to have eventId "Test" to be accepted
driver->Trigger->Sources->Item["LAN0"]->EventId = "Test";
//Incoming events must come from IP 1.2.3.4 on port 5678 to be accepted
driver->Trigger->Sources->Item["LAN0"]->Filter = "1.2.3.4:5678"
/* Triggering now occurs when a falling-edge event is received rather than a rising-edge event */
driver->Trigger->Sources->Item["LAN0"]->Detection = VTEXSystemSourceSlopeFall;
```

ARM INTERFACE

The Arm subsystem is more complex than the trigger subsystem, as it allows for multiple Arm Sources to be enabled simultaneously and adds two more values to the Detection property (High and Low). Since synchronizing multiple hardware edges is very difficult, it is possible to change from logical-AND to logical-OR of these sources.

```
/* Set DIO3 and an arm source */
driver->Arm->Sources->Item["DIO3"]->Enabled = VARIANT_TRUE;

/* Allow either DIO3 or LXI2 to be used as an arm source. */
driver->Arm->Sources->Item["DIO3"]->Enabled = VARIANT_TRUE;
```

```
driver->Arm->Sources->Item[“LXI2”]->Enabled = VARIANT_TRUE;
//Logical-OR the two sources so that an event at either source will cause the device to arm.
driver->Arm->Sources->OrEnabled = VARIANT_TRUE;

/* Set an ARM event so that all DIO lines must go high and a rising edge must occur on LAN4 */
driver->Arm->Sources->Item[“DIO0”]->Enabled = VARIANT_TRUE;
driver->Arm->Sources->Item[“DIO1”]->Enabled = VARIANT_TRUE;
driver->Arm->Sources->Item[“DIO2”]->Enabled = VARIANT_TRUE;
driver->Arm->Sources->Item[“DIO3”]->Enabled = VARIANT_TRUE;
driver->Arm->Sources->Item[“DIO4”]->Enabled = VARIANT_TRUE;
driver->Arm->Sources->Item[“DIO5”]->Enabled = VARIANT_TRUE;
driver->Arm->Sources->Item[“DIO6”]->Enabled = VARIANT_TRUE;
driver->Arm->Sources->Item[“DIO7”]->Enabled = VARIANT_TRUE;
driver->Arm->Sources->Item[“DIO0”]->Detection = VTEXSystemSourceSlopeHigh;
driver->Arm->Sources->Item[“DIO1”]->Detection = VTEXSystemSourceSlopeHigh;
driver->Arm->Sources->Item[“DIO2”]->Detection = VTEXSystemSourceSlopeHigh;
driver->Arm->Sources->Item[“DIO3”]->Detection = VTEXSystemSourceSlopeHigh;
driver->Arm->Sources->Item[“DIO4”]->Detection = VTEXSystemSourceSlopeHigh;
driver->Arm->Sources->Item[“DIO5”]->Detection = VTEXSystemSourceSlopeHigh;
driver->Arm->Sources->Item[“DIO6”]->Detection = VTEXSystemSourceSlopeHigh;
driver->Arm->Sources->Item[“DIO7”]->Detection = VTEXSystemSourceSlopeHigh;
driver->Arm->Sources->Item[“LAN4”]->Enabled = VARIANT_TRUE;
```

TIME & ALARM INTERFACES

Alarms are arm events that are based on IEEE-1588 time and allow users to create arm events at a specified time. This can be used to set a arm event for a date and time in the future or may be used to specify a time interval.

NOTE Arm alarms should not be used unless the device is synchronized to an IEEE-1588 time source.

```
/* Set an Arm alarm to go off at Fri Feb 13 2009 18:31:30.0503 EST and fire exactly once. Note
      that Arm Alarms have an Enabled property in their Configure call which also adds the Alarm
      to the enabled Arm sources. */
if(driver->Time->IsSynchronized)
{
      //Note that 0 is a special case for Period and RepeatCount
driver->Arm->Alarms->Item[“ALARM0”]->Configure(VARIANT_TRUE, 1234567890.0, 0.0503, 0, 0);
}

/* Set a Route alarm to go off 10 seconds from the current time and fire 10 times with a period
      of one second. Note that the start time won’t be exactly 10 seconds from now due to network
      delays. Also note that the Route Alarms have an Enabled property which only enables the
      alarm, but does nothing else. */

if(driver->Time->IsSynchronized)
{
      double seconds;
      double fraction;
      driver->Time->GetSystemTime(&seconds, &fraction); //Get the current time
      driver->InstrumentSpecific->Route->Alarms->Item[“ALARM0”]->Configure(VARIANT_TRUE,
      seconds+10.0, fraction, 1.0, 10);
      driver->InstrumentSpecific->Route->Alarms->Item[“ALARM0”]->Enabled = VARIANT_TRUE;
}
```

EVENTS INTERFACE

The VTEXSystem drivers utilizes the standard IVI interface for routing events from source to another. It also expands upon the interface with the Routing interface, but the Events interface still exists and is l.

```
/* Route all signals from LAN3 to LXI2 */
driver->Events->Item[“LXI2”]->Source = “LAN3”;
driver->Events->Item[“LXI2”]->DriveMode = VTEXSystemEventDriveModeDriven;
```

```
/* Route and invert all signals from LAN3 to LXI2 */
driver->Events->Item[“LXI2”]->Source = “LAN3”;
driver->Events->Item[“LXI2”]->Slope = VTEXSystemSourceSlopeFall;
driver->Events->Item[“LXI2”]->DriveMode = VTEXSystemEventDriveModeDriven;
```

ROUTE INTERFACE

The route interface provides more flexible than the Events interface. If a device supports a backplane, any of the backplane destinations can have multiple sources. The Route interface also allow for software to control a line and for a Pulse command, which sends a short pulse on the defined line.

```
/* Route all signals from LAN3 to LXI2 as was done in the first Events Interface example above */
driver->InstrumentSpecific->Route->Destinations->Item[“LXI2”]->SourcesList = “LAN3”;
driver->InstrumentSpecific->Route->Destinations->Item[“LXI2”]->DriveMode =
      VTEXSystemEventDriveModeDriven;

/* Route and invert all signals from LAN3 to LXI2 as was done in the second Events Interface
      example. Note that the SourcesList must be cleared because LXI lines do not support
      multiple sources. */
driver->InstrumentSpecific->Route->Destinations->Item[“LXI2”]->SourcesList = “”;
driver->InstrumentSpecific->Route->Destinations->Item[“LXI2”]->InvertedSourcesList = “LAN3”;
driver->InstrumentSpecific->Route->Destinations->Item[“LXI2”]->DriveMode =
      VTEXSystemEventDriveModeDriven

/* Toggle DIO3 for 5 seconds. This cannot be done using the Events interface. Note that the
      SourcesList and InvertedSourcesList must be empty for software control to work. */
//If the line is undriven, changing its state will have no effect.
driver->InstrumentSpecific->Route->Destinations->Item[“DIO3”]->DriveMode =
      VTEXSystemEventDriveModeDriven
driver->InstrumentSpecific->Route->Destinations->Item[“DIO3”]->SoftwareState = 1;
//Use the operating system’s “Sleep” command here
Sleep(5000);
driver->InstrumentSpecific->Route->Destinations->Item[“DIO3”]->SoftwareState = 0;

/* What happens if a line is assigned assign multiple sources? By default they’re logical AND’ed
      together, just like Arm sources. Note that multiple sources can be defined in each
      SourcesList if desired.*/
driver->InstrumentSpecific->Route->Destinations->Item[“BPL2”]->SourcesList = “LXI2, DIO3”;
driver->InstrumentSpecific->Route->Destinations->Item[“BPL2”]->InvertedSourcesList = “LAN3”;
//It can be changed to Logical-OR as well
driver->InstrumentSpecific->Route->Destinations->Item[“BPL2”]->OrEnabled = VARIANT_TRUE;
```

IOPORTS INTERFACE

The IOPorts interface allows for the current state of the device’s hardware and LAN lines to be checked. These states are presented as 8-bit bitmasks that indicate the states of lines 0 through 7. Using this interface, either the DrivenState or the InputState can be checked. The DrivenState is the state that is driven into the line if DriveMode is enabled, while the InputState is the state that the hardware reads, regardless of is being driven. The InputState and DrivenState can be different if the line is not being driven, if the line is in WiredOr mode and a different device is pulling it high, or if there is an electrical fault on the line where one device is attempting to drive the line high and another is attempting to drive the line low.

```
#include "stdafx.h"
#using <mscorlib.dll>
#import "IviDriverTypeLib.dll" no_namespace
#import "VTEXSystem.dll" no_namespace

using namespace System;

int _tmain()
```

```
{
    ::CoInitialize(NULL); //Start the COM layer
     /*We want to instantiate a pointer to the driver in a try/catch block so that we fail
     properly if the driver is not found in the COM registry*/

     try
     {
       IVTEXSystemPtr system(__uuidof(VTEXSystem));

       /*We want to do the Initialization a try/catch block so that our test code
       doesn't run if we fail to initialize.*/
       try
       {
              /*We chose to give the driver an empty options string. For more information on
              options, refer to the Option Strings discussion in Section 1. Note that we also
              set the Reset bit so that we get a clean start to work from.*/
              system->Initialize("TCPIP::10.1.4.55::INSTR",VARIANT_TRUE,VARIANT_TRUE, "");

              /*The IOPorts interface allows us to check the current state of the device's
              hardware and LAN lines. These states are presented as 8-bit bitmasks, indicating
              the states of lines 0-7. Using this interface we can either check the DrivenState,
              meaning the state that we attempt to drive onto the line if the DriveMode is
              enabled, or the InputState, meaning the state that the hardware itself reads,
              regardless of what we are attempting to drive. The InputState and DrivenState can
              be different if the line is not being driven, if the line is in WiredOr mode and a
              different device is pulling it high, or if there is an electrical fault on the
              line where one device is attempting to drive the line high, and another is
              attempting to drive the line low. */

              //This is the current state of the 8 Backplane lines.
              int state = system->InstrumentSpecific->IOPorts->Item["BPL"]->InputState;
              int bpl0state = state & 0x1; //Select the first bit of the bitmask to check the
              state of BPL0.

              //Change the DrivenState for BPL0 by altering the software state
              //Note that this has no impact on the state of the line since the DriveMode should
              still be Off.
              system->InstrumentSpecific->Route->Destinations->Item["BPL0"]->SoftwareState=1;

              //Retrieve the driven state
              state = system->InstrumentSpecific->IOPorts->Item["BPL"]->DrivenState;

              //We would expect the two states to not match, due to the change in SoftwareState
              state = bpl0state != (state & 0x1); //should be true

              //Now we can enable the DriveMode and change the InputState as well, since we will
              now be driving the line
              system->InstrumentSpecific->Route->Destinations->Item["BPL0"]->DriveMode =
     VTEXSystemEventDriveModeDriven;

              //Retrieve the InputState again
              state = system->InstrumentSpecific->IOPorts->Item["BPL"]->InputState;

              //And now we can confirm that the input state no longer matches the original input
                     state, since we are driving the line
              state = bpl0state != (state & 0x1); //should be true

              //One note about this functionality - since the BPL_INSFAIL line is not exactly a
              backplane line, it has its own entry in the IOPorts capability.
              //It is also only a single bit wide.
              state = system->InstrumentSpecific->IOPorts->Item["BPL_INSFAIL"]->InputState;

              //Close the initialized session
              system->Close();
       }
       catch(_com_error &e)
       {
              ::MessageBox(NULL, e.Description(), e.ErrorMessage(), MB_ICONERROR);
       }

     }
```

```
      catch(...)
      {
        /*We put this here to catch any error the program generates.*/
        //Do something to intelligently deal with errors
      }
}
```

EVENTLOG INTERFACE

The Event Log keeps a record of LXI LAN Event Packets that are received. It is enabled in circular-buffer mode by default, but also has a FIFO-mode which stops once the buffer is full.

```
#include "stdafx.h"
#using <mscorlib.dll>
#import "IviDriverTypeLib.dll" no_namespace
#import "VTEXSystem.dll" no_namespace

using namespace System;

int _tmain()
{
    ::CoInitialize(NULL); //Start the COM layer
     /*We want to instantiate a pointer to the driver in a try/catch block so that we fail
     properly if the driver is not found in the COM registry*/

     try
     {
       IVTEXSystemPtr system(__uuidof(VTEXSystem));

       /*We want to do the Initialization a try/catch block so that our test code
       doesn't run if we fail to initialize.*/
       try
       {
              /*We chose to give the driver an empty options string. For more information on
              options, refer to the Option Strings discussion in Section 1 Note that we also set
              the Reset bit so that we get a clean start to work from.*/
              system->Initialize("TCPIP::10.1.4.55::INSTR",VARIANT_TRUE,VARIANT_TRUE, "");

              /* The Event Log keeps track of received LXI LAN Event Packets for us. It is
     enabled in circular-buffer mode by default,
              but also has a fifo-mode which stops when it is full. */

              //The event log can be disabled if the user doesn't want to spend system resources
     logging packets.
              system->EventLog->Enabled = VARIANT_FALSE;

              //However, leaving the log enabled can be helpful if using LAN triggers, to help
     debug issues with a test system
              system->EventLog->Enabled = VARIANT_TRUE;

              //We can also change the system so that it stops recording events when the log is
     full.
              system->InstrumentSpecific->EventLogOverflowMode = VTEXSystemEventLogOverflowStop;

              //We can also retrieve current entries in the event log, if there are any
              if(system->EventLog->EntryCount > 0)
              {
                     _bstr_t entry = system->EventLog->GetNextEntry();
              }

              //Or, we can clear the entire log
              system->EventLog->Clear();

              //Close the initialized session
              system->Close();
       }
       catch(_com_error &e)
       {
              ::MessageBox(NULL, e.Description(), e.ErrorMessage(), MB_ICONERROR);
```

```
        }

      }
      catch(...)
      {
        /*We put this here to catch any error the program generates.*/
        //Do something to intelligently deal with errors
      }
}
```