Getting started with IVI and Python

Your Guide to Using IVI-Drivers with Python

Version 1.0

Contents

Chapter 1 Introduction

Purpose

Welcome to ***Getting Started with IVI and Python***. This guide introduces key concepts about IVI drivers and shows you how to create a short program to perform a measurement. The guide is part of the IVI Foundation's series of guides, ***IVI Getting Started Guides***.

IVI Getting Started Guides are intended for individuals who write and run programs to control test-and-measurement instruments. Each guide focuses on a different programming environment. As you develop test programs, you face decisions about how you communicate with the instruments. Some of your choices include Direct I/O, VXI*plug&play* drivers, or IVI drivers. If you are new to using IVI drivers or just want a quick refresher on the basics, ***IVI Getting Started Guides*** can help.

IVI Getting Started Guides shows that IVI drivers can be straightforward, easy-to-use tools. IVI drivers provide a number of advantages that can save time and money during development, while improving performance as well. Whether you are starting a new program or making improvements to an existing one, you should consider the use of IVI drivers to develop your test programs.

So, consider this the "hello, instrument" guide for IVI drivers. If you recall, the "hello world" program, which originally appeared in Programming in C: A Tutorial, simply prints out "hello, world." The "hello, instrument" program performs a simple measurement on a simulated instrument and returns the result. We think you'll find that far more useful.

Why Use Python with an IVI Driver?

Python is a rising star in the programming world. Major T&M companies have added Python support everywhere. Python is an important tool that customers want to use. There is strong third-party support: Google and Microsoft, for example. And support from the Python community is outstanding.

Python is a multi-paradigm programming language. Object-oriented programming and structured programming are fully supported, and many of its features support functional programming and aspect-oriented programming.

And now, with this getting started guide, you will be able to add the benefits of IVI drivers to your Python ATE applications.

Why Use an Instrument Driver?

To understand the benefits of IVI drivers, we need to start by defining instrument drivers in general and describing why they are useful. An instrument driver is a set of software routines that controls a programmable instrument. Each routine corresponds to a programmatic operation, such as configuring, writing to, reading from, and triggering the instrument. Instrument drivers simplify instrument control and reduce test program

development time by eliminating the need to learn the programming protocol for each instrument.

Starting in the 1970s, programmers used device-dependent commands for computer control of instruments. But lack of standardization meant even two digital multimeters from the same manufacturer might not use the same commands. In the early 1990s a group of instrument manufacturers developed Standard Commands for Programmable Instrumentation (SCPI). This defined set of commands for controlling instruments uses ASCII characters, providing some basic standardization and consistency to the commands used to control instruments. For example, when you want to measure a DC voltage, the standard SCPI command is "MEASURE:VOLTAGE:DC?".

In 1993, the VXI*plug&play* Systems Alliance created specifications for instrument drivers called VXI*plug&play* drivers. Unlike SCPI, VXI*plug&play* drivers do not specify how to control specific instruments; instead, they specify some common aspects of an instrument driver. By using a driver, you can access the instrument by calling a subroutine in your programming language instead of having to format and send an ASCII string as you do with SCPI. With ASCII, you have to create and send the instrument the syntax "MEASURE:VOLTAGE:DC?", then read back a string, and build it into a variable. With a driver you can merely call a function called MeasureDCVoltage( ) and pass it a variable to return the measured voltage.

Although you still need to be syntactically correct in your calls to the instrument driver, making calls to a subroutine in your programming language is less error prone. If you have been programming to instruments without a driver, then you are probably all too familiar with hunting around the programming guide to find the right SCPI command and exact syntax. You also have to deal with an I/O library to format and send the strings, and then build the response string into a variable.

Why IVI?

The VXI*plug&play* drivers do not provide a common programming interface. That means programming a Keithley DMM using VXI*plug&play* still differs from programming a Keysight DMM. For example, the instrument driver interface for one may be ke2000_read while another may be ag34401_get or something even farther afield. Without consistency across instruments manufactured by different vendors, many programmers still spent a lot of time learning each individual driver.

To carry VXI*plug&play* drivers a step (or two) further, in 1998 a group of end users, instrument vendors, software vendors, system suppliers, and system integrators joined together to form a consortium called the Interchangeable Virtual Instruments (IVI) Foundation. If you look at the membership, it's clear that many of the foundation members are competitors. But all agreed on the need to promote specifications for programming test instruments that provide better performance, reduce the cost of program development and maintenance, and simplify interchangeability.

For example, for any IVI driver developed for a DMM, the measurement command is IviDmmMeasurement.Read, regardless of the vendor. Once you learn how to program the commands specified by IVI for the instrument class, you can use any vendor's instrument and not need to relearn the commands.

Also commands that are common to all drivers, such as Initialize and Close, are identical regardless of the type of instrument. This commonality lets you spend less time browsing through the help files in order to program an instrument, leaving more time to get your job done.

That was the motivation behind the development of IVI drivers. The IVI specifications enable drivers with a consistent and high standard of quality, usability, and completeness. The specifications define an open driver architecture, a set of instrument classes, and shared software components. Together these provide consistency and ease of use, as well as the crucial elements needed for the advanced features IVI drivers support: instrument simulation, automatic range checking, state caching, and interchangeability.

The IVI Foundation has created IVI class specifications that define the capabilities for drivers for the following thirteen instrument classes:

Class	IVI Driver
Digital multimeter (DMM)	IviDmm
Oscilloscope	IviScope
Arbitrary waveform/function generator	IviFgen
DC power supply	IviDCPwr
AC power supply	IviACPwr
Switch	IviSwtch
Power meter	IviPwrMeter
Spectrum analyzer	IviSpecAn
RF signal generator	IviRFSigGen
Upconverter	IviUpconverter
Downconverter	IviDownconverter
Digitizer	IviDigitizer
Counter/timer	IviCounter

IVI Class Compliant drivers usually also include numerous functions that are beyond the scope of the class definition. This may be because the capability is not common to all instruments of the class or because the instrument offers some control that is more refined than what the class defines.

IVI also defines custom drivers. Custom drivers are used for instruments that are not members of a class. For example, there is not a class definition for network analyzers, so a network analyzer driver must be a custom driver. Custom drivers provide the same consistency and benefits described below for an IVI driver, except interchangeability.

IVI drivers that conform to the IVI specifications are permitted to display the IVI- Conformant logo.

Why Use an IVI Driver?

Why choose IVI drivers over other possibilities? Because IVI drivers can increase performance and flexibility for more intricate test applications. Here are a few of the benefits:

Consistency – IVI drivers all follow a common model of how to control the instrument. That saves you time when you need to use a new instrument.

Ease of use – IVI drivers feature enhanced ease of use in popular Application Development Environments (ADEs). The APIs provide fast, intuitive access to functions. IVI drivers use technology that naturally integrates in many different software environments.

Quality – IVI drivers focus on common commands, desirable options, and rigorous testing to ensure driver quality.

Simulation – IVI drivers allow code development and testing even when an instrument is unavailable. That reduces the need for scarce hardware resources and simplifies test of measurement applications. The example programs in this document use this feature.

Range checking – IVI drivers ensure the parameters you use are within appropriate ranges for an instrument.

State caching – IVI drivers keep track of an instrument's status so that I/O is only performed when necessary, preventing redundant configuration commands from being sent. This can significantly improve test system performance.

Interchangeability – IVI class compliant drivers also enable exchange of instruments with minimal code changes, reducing the time and effort needed to integrate measurement devices into new or existing systems. The IVI class specifications provide syntactic interchangeability but may not provide behavioral interchangeability. In other words, the program may run on two different instruments, but the results may not be the same due to differences in the way the instrument itself functions.

Flavors of IVI Drivers

To support all popular programming languages and development environments, IVI drivers provide either an IVI-C or IVI-COM (Component Object Model) API. Driver developers may provide either or both interfaces, as well as wrapper interfaces optimized for specific development environments.

Although the functionality is the same, IVI-C drivers are optimized for use in ANSI C development environments; IVI-COM drivers are optimized for environments that support the Component Object Model (COM) such as the .NET programming environment. IVI-C drivers extend the VXI*plug&play* driver specification and their usage is similar. IVI-COM drivers provide easy access to instrument functionality through methods and properties.

The getting started examples communicate with the instruments using the Virtual Instrument Software Architecture (VISA) I/O library, a widely used standard library for communicating with instruments from a personal

computer. The VISA standard is also provided by the IVI Foundation.

Shared Components

To make it easier to combine drivers and other software from various vendors, the IVI Foundation members have cooperated to provide common software components, called IVI Shared Components. These components provide services to drivers and driver clients that need to be common to all drivers. For instance, the IVI Configuration Server enables administration of system-wide configuration.

Important! You must install the IVI Shared Components before an IVI driver can be installed.

The IVI Shared Components can be downloaded from vendors' web sites as well as from the IVI Foundation Web site.

To download and install shared components from the IVI Foundation Web site:

1. Go to the IVI Foundation Web site at http://www.ivifoundation.org.
2. Locate the Shared Components page.
3. Choose the IVI Shared Components msi file for the Microsoft Windows Installer package or the IVI Shared Components exe for the executable installer.

Download and Install IVI Drivers

After you've installed Shared Components, you're ready to download and install an IVI driver. For most ADEs, the steps to download and install an IVI driver are identical. For the few that require a different process, the relevant ***IVI Getting Started Guides*** guide provides the information you need. IVI Drivers are available from the hardware or software vendors' web site or by linking to them from the IVI Foundation web site.

The IVI Foundation requires that compliant drivers be registered before the display the IVI conformant logo. To see the list of drivers registered with the IVI Foundation, go to the registration section of the IVI web site at http://www.ivifoundation.org.

Familiarizing Yourself with the Driver

Although the examples in IVI Getting Started Guides use a DMM driver, you will likely employ a variety of IVI drivers to develop test programs. To jumpstart that task, you'll want to familiarize yourself quickly with drivers you haven't used before. Most ADEs provide a way to explore IVI drivers to learn their functionality. In each IVI guide, where applicable, we add a note explaining how to view the available functions. In addition, browsing an IVI driver's help file often proves an excellent way to learn its functionality.

Examples

As we noted above, each guide in the ***IVI Getting Started Guides*** series shows you how to use an IVI driver to write and run a program that performs a simple measurement on a simulated instrument and returns the result. The examples demonstrate common steps using IVI drivers. Where practical, every example includes the steps listed below:

- Download and Install the IVI driver– covered in the Download and Install IVI Drivers section above.
- Determine the VISA address string – Examples in ***IVI Getting Started Guides*** use the simulate mode, so we chose the address string **GPIB0::23::INSTR,** often shown as GPIB::23. If you need to determine the VISA address string for your instrument and the ADE does not provide it automatically, use an IO application, such as National Instruments Measurement and Automation Explorer (MAX) or Keysight Connection Expert.
- Reference the driver or load driver files – For the examples in this guide, the driver is the IVI-COM/IVI-C Version 1.3.0.0 for 34401A, March 2015 (from Keysight Technologies) … or the Keysight 34401A IVI-C driver, Version 4.5, January 2015 (from National Instruments).
- Create an instance of the driver in ADEs that use COM – For the examples in the IVI guides, the driver is the Agilent 34401A (IVI-COM) or HP 34401 (IVI-C).
- Write the program. The programs in this series all perform the following steps:
    - Initialize the instrument – Initialize is required when using any IVI driver. Initialize establishes a communication link with the instrument and must be called before the program can do anything with the instrument. The examples set reset to **true**, ID query to **false**, and simulate to **true**.

      Setting reset to true tells the driver to initially reset the instrument. Setting the ID query to false prevents the driver from verifying that the connected instrument is the one the driver was written for. Finally, setting simulate to true tells the driver that it should not attempt to connect to a physical instrument, but use a simulation of the instrument.
    - Configure the instrument – The examples set a range of **1.5 volts** and a resolution of 0.001 volts (1 millivolt).
    - Access an instrument property – The examples set the trigger delay to **0.01 seconds**.
    - Set the reading timeout – The examples set the reading timeout to **1000 milliseconds (1 second).**
    - Take a reading
    - Close the instrument – This step is required when using any IVI driver, unless the ADE explicitly does not require it. We close the session to free resources.

      Important! Close may be the most commonly missed step when using an IVI driver. Failing to do this could mean that system resources are not freed up and your program may behave unexpectedly on subsequent executions.

- Check the driver for any errors.
- Display the reading.

Note: *Examples that use a console application do not show the display.*

Now that you understand the logic behind IVI drivers, let's see how to get started.

Chapter 2
Using IVI-C with Python

The Environment

Python is an open source programming language developed by the Python Software Foundation. It is interpreted and fully object-oriented with a focus on readability and efficient development. Python is used across a wide variety of applications and features a robust set of third-party modules. This chapter provides detailed instructions on how to call an IVI-C specific driver using Python.

Example Requirements

- Python 2.7 or Python 3.6+
- Matplotlib 2.2.3 for Python 2.7 or 3.0.1 for Python 3.6+
- Tektronix tktds1k2k IVI-C driver, Version 3.7, October 2015 (from National Instruments)

Download and Install the Driver (prerequisites)

If you have not already installed the driver, go to the NI Instrument Driver website and follow the instructions to download and install it. You can also refer to Chapter 1, Download and Install IVI Drivers, for instructions.

This example uses an IVI-C driver. Base Python installs include *ctypes*, a function library for interfacing with C DLLs.

Defining VISA Types as ctypes

To help with translating function calls from the tktds1k2k driver, you can first create *ctype* aliases for commonly used VISA types.

1. Create a new Python source file named `visatype.py`.
2. Import the *`ctypes`* library.
3. Declare type aliases for primitive types defined in `visatype.h` in equivalent `ctypes`.

```
# -*- coding: utf-8 -*-
import ctypes

ViChar = ctypes.c_char
ViInt8 = ctypes.c_int8
ViInt16 = ctypes.c_int16
ViUInt16 = ctypes.c_uint16
ViInt32 = ctypes.c_int32
ViUInt32 = ctypes.c_uint32
ViInt64 = ctypes.c_int64
ViString = ctypes.c_char_p
ViReal32 = ctypes.c_float
ViReal64 = ctypes.c_double

# Types that are based on other visatypes
ViBoolean = ViUInt16
VI_TRUE = ViBoolean(True)
VI_FALSE = ViBoolean(False)
ViStatus = ViInt32
```

```
ViSession = ViUInt32
ViAttr = ViUInt32
ViConstString = ViString
ViRsrc = ViString
```

Load the tktds1k2k DLL into Memory

To load the tktds1k2k DLL into memory for use in Python, first import *ctypes* and the *visatype* library that you recently created.

```
import ctypes
from visatype import *
```

Now you can call the *ctypes.cdll.LoadLibrary* method with a path to the location on disk where the IVI driver C DLL is installed. This example is designed for use with 64-bit Python so it uses the path to the 64-bit driver C DLL.

```
#load tktds1k2k DLL into memory
tkDLL = ctypes.cdll.LoadLibrary(r'C:\Program Files\IVI
Foundation\IVI\Bin\tktds1k2k_64.dll')
```

Python stores a reference to the C DLL in the specified variable, `tkDLL`. The path string literal is prefixed with 'r' to indicate this is a raw string, treating the backslashes as literal characters. You are now able to call functions from the tktds1k2k driver as methods on the reference to the driver DLL.

Initialize the Instrument

Before calling the initialization and configuration methods of the driver, begin by declaring the variables you will need to pass into the function calls such as the session handle, the resource name, the option string, and any other pieces of data relevant to your application.

1. Create a session variable and set it to an instance of a `ViSession` object.
   ```
   session = ViSession()
   ```

2. Define the resource name, option string, and channel string. To allocate string variables for passing into functions, call *ctypes* the method `create_string_buffer()`. If using string literals, you need to set the OS encoding as well.

   ```
   resourceName = ctypes.create_string_buffer('1001C'.encode('wi
   ndows-1251'))
   optionString = ctypes.create_string_buffer('Simulate=1,Range
   Check=1,QueryInstrStatus=0,Cache=1'.encode('windows-1251'))
   channel = ctypes.create_string_buffer('CH1'.encode('windows-
   1251'))
   ```

3. Make a call to the initialization function in the `tktds1k2k DLL`. Because the session is returned as an output parameter from `tktds1k2k_InitWithOptions`, you need to pass a pointer to the session variable into the method. *ctypes* provides the method `pointer()` to pass variables to C DLLs as pointers.

   ```
   tkDLL.tktds1k2k_InitWithOptions(resourceName, VI_TRUE,
   VI_TRUE, optionString, ctypes.pointer(session))
   ```

Configure the Instrument

Now that the instrument is initialized, the configuration methods can be called utilizing the session variable as a reference to the current instrument session.

1. For ease of use and readability, define the constants needed for the configuration functions being used. Set the constants to their matching values in the `tktds1k2k` header file, wrapped in constructors of the type aliases defined in `visatype.py`.

```
# constants to be used by tktds1k2k driver
TKTDS1K2K_VAL_NORMAL = ViInt32(0)
TKTDS1K2K_VAL_DC = ViInt32(1)
TKTDS1K2K_VAL_EDGE_TRIGGER = ViInt32(1)
TKTDS1K2K_VAL_EDGE_TRIGGER = ViInt32(1)
TKTDS1K2K_VAL_MATH_FFT_CH1 = ViInt32(6)
TKTDS1K2K_VAL_POSITIVE = ViInt32(1)
```

2. Make calls to the necessary configuration functions. The first parameter is always the session handle variable. Any additional static data should be wrapped in constructors for the matching types in `visatype.py` to ensure data is being properly passed to the functions.

```
tkDLL.tktds1k2k_ConfigureAcquisitionType(session,
TKTDS1K2K_VAL_NORMAL)
tkDLL.tktds1k2k_ConfigureChannel(session,
channel, ViReal64(1.0), ViReal64(0),
TKTDS1K2K_VAL_DC, ViReal64(1.0), VI_TRUE)
tkDLL.tktds1k2k_ConfigureChanCharacteristics(session,
channel, ViReal64(1.0e6), ViReal64(20.0e6))
tkDLL.tktds1k2k_ConfigureAcquisitionRecord(session, ViReal64(
0.01), ViInt32(2500), ViReal64(-0.005))
tkDLL.tktds1k2k_ConfigureTrigger(session,
TKTDS1K2K_VAL_EDGE_TRIGGER, ViReal64(500e-9))
tkDLL.tktds1k2k_ConfigureMathChannel(session,
TKTDS1K2K_VAL_MATH_FFT_CH1)
tkDLL.tktds1k2k_ConfigureMathFFT(session, ViInt32(50), ViInt3
2(1), ViInt32(0), ViReal64(1))
tkDLL.tktds1k2k_ConfigureEdgeTriggerSource(session,
channel, ViReal64(0.4), TKTDS1K2K_VAL_POSITIVE)
tkDLL.tktds1k2k_ConfigureTriggerCoupling(session,
TKTDS1K2K_VAL_DC)
```

Acquire the Measurement Waveform

To handle arrays of data using *ctypes*, you must first know the size of the array you want to allocate.

1. Create variables to store the outputs of the waveform read function.

```
actualRecordLength = ViInt32()
actualPts = ViInt32()
initialX = ViReal64()
incrementX = ViReal64()
```

2. Acquire the actual number of data points from the read function.

```
tkDLL.tktds1k2k_ActualRecordLength(session, ctypes.pointer(a
ctualRecordLength))
```

3. To create an array for use with *ctypes*, you must first create a new type on the fly by using the base type, the multiplication operator, and the size of the array. The constructor can then be called to allocate a *ctypes* array variable in Python.

```
waveform = (ViReal64 * actualRecordLength.value)()
```

4. Finally, call the waveform read function, passing in the output variables as pointers.

```
tkDLL.tktds1k2k_ReadWaveform(session, channel,
actualRecordLength, ViReal64(10000.0), ctypes.pointer(wavefo
rm), ctypes.pointer(actualPts), ctypes.pointer(initialX), ct
ypes.pointer(incrementX))
```

Display the Acquired Waveform

This example uses Matplotlib, a popular third-party scientific graphing solution, to display the acquired waveform.

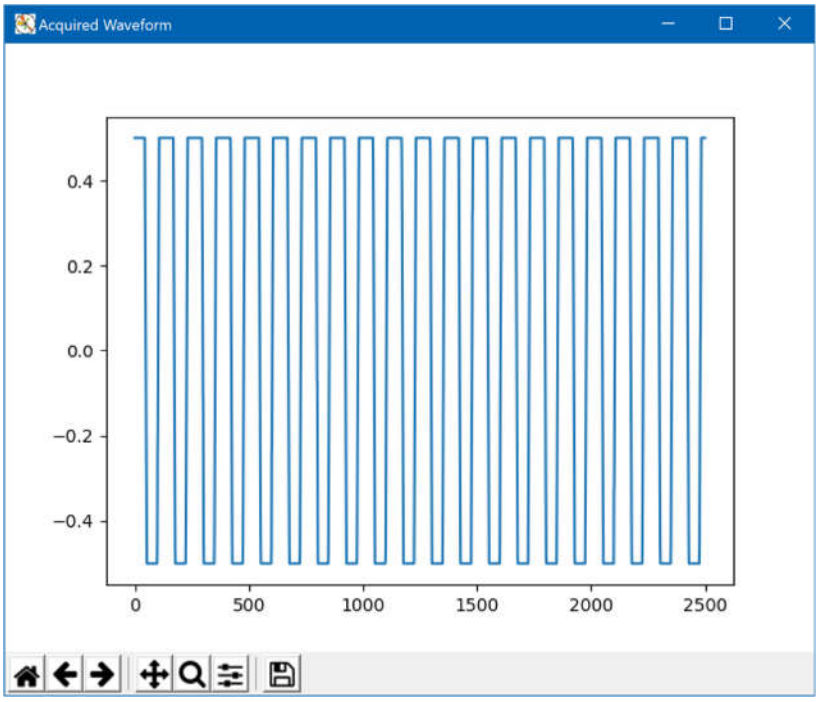
1. Import the `pyplot` class as an object.

```
import matplotlib.pyplot as plt
```

2. Configure the `plot, pass` in the acquired waveform, and display the `plot`.

```
if actualRecordLength.value !=0:
       plt.figure(0).canvas.set_window_title('Acquired
Waveform')
       plt.plot(list(waveform))
       plt.show()
```

You should see the acquired waveform in a pop-up window:



Add Error Handling

To add Pythonic error handling, you need to create a function that takes the return value of the `tktds1k2k DLL` functions and throws an exception in the case of an error. This allows you to use a try/except block to handle errors.

1. Define a new exception class that extends the Python Exception base class and minimally stores an error code.

```
class ErrorCodeException(Exception):
	def __init__(self, error_code):
		self.code = error_code
```

2. Now that you have an exception defined, build an error check function that checks the return value of the DLL calls and throws an instance of your exception class when an error occurs. Any non-zero code indicates an error occurred.

```
def checkErr(error_code):
	if error_code != 0:
		raise ErrorCodeException(error_code)
```

3. Wrap all calls to the `tktds1k2k DLL` inside of the `errCheck` function:

```
checkErr(tkDLL.tktds1k2k_InitWithOptions(resourceName,
VI_TRUE, VI_TRUE, optionString, ctypes.pointer(session)))
checkErr(tkDLL.tktds1k2k_ConfigureAcquisitionType(session,
TKTDS1K2K_VAL_NORMAL))
...
```

4. Add the main body of your application, from instrument initialization to the acquisition and displaying of data, to a `try/except` block.

```
try:
checkErr(tkDLL.tktds1k2k_InitWithOptions(resourceName,
VI_TRUE, VI_TRUE, optionString, ctypes.pointer(session)))
checkErr(tkDLL.tktds1k2k_ConfigureAcquisitionType(session,
TKTDS1K2K_VAL_NORMAL))
...
	plt.plot(list(waveform))
	plt.show()
	except ErrorCodeException as err:
```

5. In the `except` portion, call the function to get an error message from the error code stored in the exception and display the exception to the user.

```
except ErrorCodeException as err:
error_message = ctypes.create_string_buffer('\000'.encode('w
indows-1251'), 256)
tkDLL.tktds1k2k_error_message(session, ViStatus(err.code), e
rror_message )
print('Error ' + str(err.code) + ':
' + error_message.value.decode('windows-1251'))
```

Close the Instrument Session

Now that you've finished with the main body of the application, call the close method on the session variable, checking first that the session is not null.

```
if session.value != 0:
        tkDLL.tktds1k2k_close(session)
```

Further Information

- Learn more about Python at https://www.python.org/about/
- Learn more about *ctypes* at https://docs.python.org/3/library/ctypes.html

Chapter 3
Using IVI-COM with Python

The Environment

Python is an open source programming language developed by the Python Software Foundation. It is interpreted and fully object-oriented with a focus on readability and efficient development. Python is used across a wide variety of applications and features a robust set of third-party modules. This chapter provides detailed instructions on how to call an IVI-COM specific driver using Python.

Example Requirements

- Python 2.7 or Python 3.6+
- Pure Python COM package (https://pypi.python.org/pypi/comtypes)
- Matplotlib 2.2.3 for Python 2.7 or 3.0.1 for Python 3.6+
- TekSeriesScope IVI-COM Driver for 4, 5 and 6 Series Mixed Signal Oscilloscopes. V1.6.0, developed by Tektronix.

Download and Install the Driver (prerequisites)

If you have not already installed the driver, go to the IVI foundation Driver Registry website and find the above driver to download and install it. You can also refer to the driver readme file for instructions to Install the IVI COM Driver.

This example uses an IVI-COM driver. By default, the base Python does not install *comtypes* Python package. Please install the above package which is required for COM driver APIs to be called from Python.

This guide also provide example to fetch waveform data from the Oscilloscope and display it using a plot. Install the Matplotlib Python package to plot the waveform data.

comtypes, Packages and Functions

The *comtypes* package, is a pure Python COM package. The *comptype* package makes it easy to access and implement both custom and dispatch-based COM interfaces.

The *comtypes.client* package implements the high-level *comtype* functionality. We will be using *CreateObject* function for creating an object of the IVI driver.

CreateObject(progid, clsctx=None, machine=None, interface=None, dynamic=False, pServerInfo=None)

Create a COM object and return an interface pointer to it.

Load the IviSope interface from the TekSeriesScope IVI-COM Driver

```
GetModule('IviScopeTypeLib.dll')
from comtypes.gen import IviScopeLib
```

Initialize the Instrument driver

Before calling the initialization and configuration methods of the driver, lets create a driver object first.

1. Create a driver object.
   ```
   ivi_scope = CreateObject('TekSeriesScope.TekSeriesScope',in
   terface=IviScopeLib.IIviScope)
   ```
2. Make a call to the initialization function now. Pass the VISA resource string and other required parameters.
   ```
   ivi_scope.Initialize('USB::0x0699::0x0522::C011595::INSTR',
   False, False, '')
   ```

Configure the Instrument

Now that the instrument is initialized, the configuration methods can be called using the ivi_scope driver object.

1. Access the driver's Identity interface and get the instrument model. This shows how to access any driver property.
   ```
   scope_model = ivi_scope.Identity.InstrumentModel
   ```
2. Access the driver's Acquisition interface and get the current record length.
   ```
   rec_len = ivi_scope.Acquisition.RecordLength
   ```
3. Configure the Horizontal parameters, if required.
   ```
   ivi_scope.Acquisition.ConfigureRecord(TimePerRecord=.1, Min
   NumPts=10000, AcquisitionStartTime=0)
   ```
4. Access to the Channel interface and configure its parameters.
   ```
   ch1 = ivi_scope.Channels.Item('Channel1')
   ch1.Configure(Range=.1, Offset=0.1,
   Coupling=IviScopeLib.IviScopeVerticalCouplingDC, ProbeAtten
   uation=1, Enabled=True)
   ```

Acquire the Measurement Waveform

To get the waveform data from any channel of the Oscilloscope, we have to call some of the IVI Scope class defined functions like ReadWaveform or FetchWaveform. These functions are available in IviMeasurement interface of Measurements repeated capabilities group.

1. Get the access to the respective Measurement interface for the Channel.
   ```
   meas1 = ivi_scope.Measurements.Item('Channel1')
   ```
2. Call the FetchWaveform function on the above Measurement interface. FetchWaveform returns a tuple that contains WaveformArray as an array of double, InitialX and XIncrement.
   ```
   waveform = meas1.FetchWaveform()
   #print ("Waveform Array:", waveform[0])
   # waveform[0] contains the Waveform data, which we will
   plot in the next section
   # waveform[1] and waveform[2] contains
   the InitialX and XIncrement respectively
   print ("InitialX :", waveform[1])
   print ("XIncrement :", waveform[2])
   ```

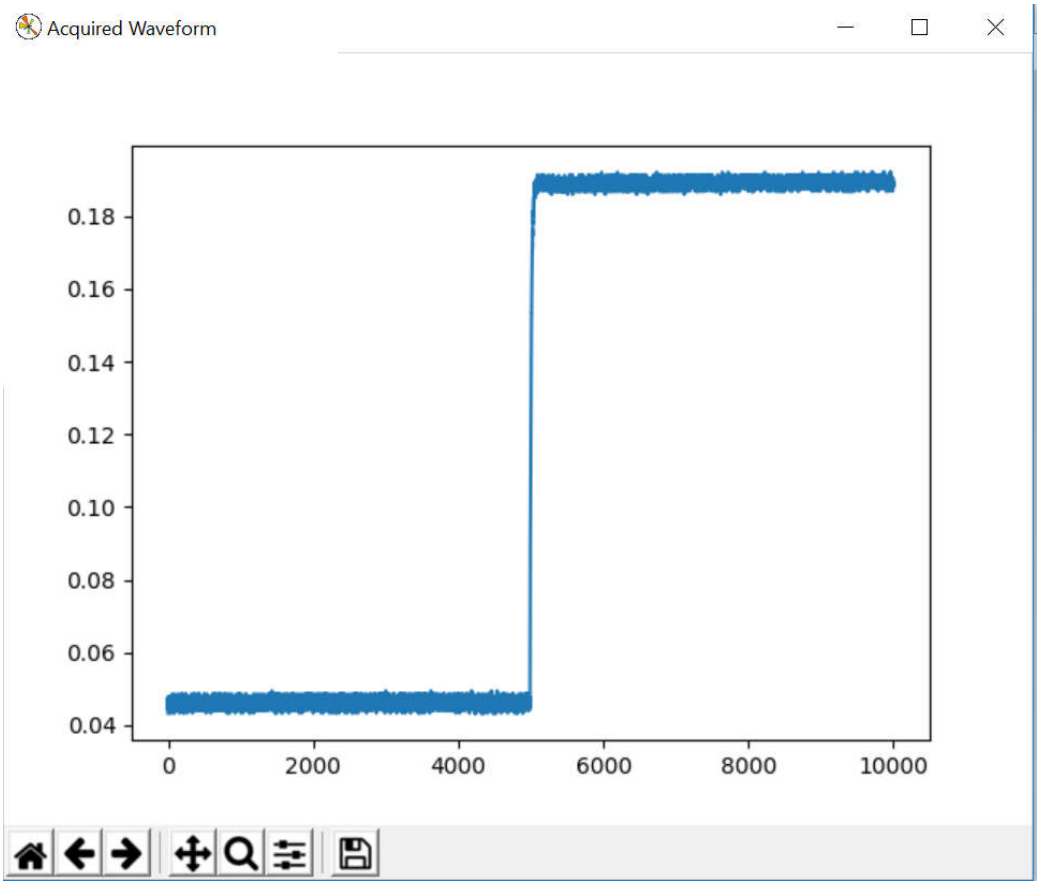
Display the Acquired Waveform

This example uses Matplotlib, a popular third-party scientific graphing solution, to display the acquired waveform.

1. Import the pyplot class as an object.
```
import matplotlib.pyplot as plt
```

2. Configure the plot, pass in the acquired waveform, and display the plot.
```
plt.figure(0).canvas.set_window_title('Acquired Waveform')
plt.plot(waveform[0])
plt.show()
```

You should see the acquired waveform in a pop-up window:



Add Error Handling

For the comtypes, the error handling is very much simplified using the “try/except” statements.
If the COM method call fails, a COMError exception is raised, containing the HRESULT value and the error message details also.
The following example shows, how to handle errors in your Python code for the IVI COM driver calls.

```
from comtypes.client import CreateObject, GetModule
import comtypes

GetModule('IviScopeTypeLib.dll')
from comtypes.gen import IviScopeLib
ivi_scope = CreateObject('TekSeriesScope.TekSeriesScope',
interface=IviScopeLib.IIviScope)
try:
    ivi_scope.Initialize('USB::0x0699::0x0522::C011595::INSTR',
False, False, '')
except comtypes.COMError as ce:
    #Get the Error details as a tuples
    com_error = ce.args
    #Get the HRESULT value
    hresult = com_error[0]
    #Get the Error messages details
    error_messages = com_error[2]
```

Close the Instrument driver connection

Now that you’ve finished with the main body of the application, call the Close method on the driver object.

```
ivi_scope.Close()
```

Further Information

- Learn more about Python at https://www.python.org/about/
- Learn more about comtypes at https://pythonhosted.org/comtypes/
- IVI Foundation: http://ivifoundation.org/

Note:

The example code provided in this document was tested with

- Python 3.7 (32-bit versions)
- TekSeriesScope IVI-COM Driver ver 1.6.0
- TekVISA version 4.2.0.16