How to Use the RSA API in Python

One. Before you do anything else, download and install Anaconda: http://continuum.io/downloads. Anaconda is a 64-bit Python distribution that already has graphing and numerical processing modules installed. Having Anaconda before playing with the RSA_API will mitigate confusion and remove the need to explain the process of installing and managing Python modules. These examples are based on Python 3.6, but Python 2.7 code can also be provided if necessary.

Two. Download and install the most recent version of the RSA_API. As of 5/17 the most recent version is 3.9.0029 and can be downloaded here: http://www.tek.com/model/rsa306-software
Download the documentation for the API. As of 5/17 the most recent version is 077-1031-03 and can be downloaded here: http://www.tek.com/spectrum-analyzer/rsa306-manual-0

Three. This document assumes you have a basic knowledge of Python programming (using variables, calling functions, knowledge of data types, running Python scripts from the command line, etc.)

Four. I have a legend. data types = **bold**, variables = underlined, and functions/methods = *italics()*

Intro

All of the magic in the API happens in RSA_API.dll. A dll is a Dynamic Link Library, which contains precompiled functions that can be accessed by a script or program. This dll was written in and for C, so in order to import the dll and access its functions in Python, we have to use Python's ctypes module, which allows Python to understand and use data types that are native to C. Navigating this translation can be tricky if you have never written code in C before or are unfamiliar with programming in general.

Object-Oriented Programming

Python is an object-oriented language. You can think of *objects* as complex data types that contain functions (or *methods* to be more semantically precise) and attributes that can be used by the rest of the program. The RSA_API.dll provides you with an *object* that has a whole bunch of *methods* that you can use to send commands to and get data from the connected RSA. In all the following examples, my object is called rsa. In order to use any of the *methods* contained in the *object*, use the *object.method()* notation. For example, to use *DEVICE_Disconnect()* on the RSA, write the following:

```
rsa.DEVICE_Disconnect()
```

All the methods listed in the API documentation are accessed this way, which you will see as you read further.

*See end of guide for more information about using a separate Python module to import structs and enums

Working with ctypes variables.

Creating a variable in Python using a C data type is easy after you import the ctypes module. Let's use the refLevel variable as an example. Based on the API documentation, all functions that utilize refLevel expect to see a **double** data type. Ctypes can create ctypes objects with a C data types that have certain attributes we can use. To create this variable as a **double** with a value of 0, type the following:

```
from ctypes import *
refLevel = c_double(0)
```

Creating arrays has one more step between you and a fully usable variable, but it isn't complicated once you understand how it works. Let's create an array that contains a set of 1024 points as **float**s. First, we create a variable that allocates enough space in memory to hold the entire array by multiplying the desired data type (**float** in this case) by the array length. For those familiar with C, this is like using *malloc()*. For example, the following code allocates enough memory for the array and then initializes it.

```
dataArray = c_float*1024
myData = dataArray()
```

Note that ctypes data can't always be used directly by Python. In general, if you want to use the number contained in a variable, you need to use the *.value* attribute. Take the following scenario for example:

```
from ctypes import *

bla = c_double(102)
dingus = bla + 17
print(dingus)
```

This code throws an error because Python can't add a standard Python **int** to a **c_double.**

```
Traceback (most recent call last):
  File "E:\zFAQs\AEU Classes\examples.py", line 4, in <module>
    dingus = bla + 17
TypeError: unsupported operand type(s) for +: 'c_double' and 'int'
[Finished in 0.1s with exit code 1]
```

However, if I change it slightly by using the *.value* attribute (much like passing an argument by value, which we'll talk about later) Python uses the value contained in bla rather than using the whole C-packaged data type. Everything is peachy.

```
from ctypes import *

bla = c_double(102)
dingus = bla.value + 17
print(dingus)
```

```
119.0
[Finished in 0.1s]
```

*See end of guide for more information about using a separate Python module to import structs and enums

Quick overview of pointers.

Pointers are like web page bookmarks or desktop shortcuts, but instead of pointing to a specific website or application, they point directly to a location in computer memory. When you create/initialize a large variable in C (an array used to store IQ data, for example) you can store it in memory and access it using a pointer.

Why does it matter how you access variables, you may ask? Who cares how I get the data as long as I get it, right? The short answer is that using pointers is more time and memory efficient. Imagine if you had a 1M long IQ array. If you wanted to feed that array into a function, you'd have to copy the ENTIRE THING to a local variable and pass it to the function. Now you have a new variable that has been modified, and if you want to replace the original variable with the modified one you need to copy it back to its original memory location. If you pass the variable to a function using a pointer, the program simply goes to the variable's memory location and does calculations on it there without wasting time and memory by copying it back and forth. If you use pointers you can also have multiple functions operating on the same data without having to manage new variables every time you pass the data to a function.

Passing by value vs by reference

C gives you flexibility when sending arguments to functions. You can either pass the VALUE of a variable or you can pass a POINTER to the variable. When passing by value, the function can only use the variable's value, but it can't make any changes to the variable itself since it doesn't have access to its location in memory. When passing by reference, because the function actually has access to the variable's memory location, a function can manipulate the variable itself. For example, if I wanted to set the reference level of the RSA, I could just pass a value because I don't want the dll to do anything to the refLevel variable we created earlier. In another example, if I wanted to get the reference level from the RSA, I would want to pass the variable by reference because I want the dll to change the value of refLevel to reflect the actual reference level of the RSA.

Why am I telling you this? Well, the way Python handles passing by value or by reference is… complicated and beyond the scope of this document. The key is that you can force Python to pass a variable by reference so that a method in the dll can manipulate it. It's fairly simple, just put *byref*() around the variable you're passing to the function and Python will take care of it from there.

```
rsa.CONFIG_GetCenterFreq(byref(cf))
```

How do I know if a variable needs to be passed by value or reference? The API documentation lists certain arguments as pointers. Those are the ones that need to be passed by reference. The others can be passed normally (by value).

*See end of guide for more information about using a separate Python module to import structs and enums

CONFIG_GetCenterFreq	Queries the center frequency.
Declaration:	ReturnStatus CONFIG_GetCenterFreq(double* cf);
Parameters:	
cf:	Pointer to a double. Contains the center frequency when the function completes.
Return Values:	
noError:	The center frequency has been queried.
errorNotConnected:	The device is not connected.
Additional Detail:	The center frequency determines the center location for the spectrum view.

Necessary components of an API script

1. Import the ctypes module into your script. To do this, type the following line:

   ```
   from ctypes import *
   ```

   Although it may seem foreign and scary (what's a module? why can't I just type "import ctypes"?), this statement is simply telling Python to load every method from the ctypes module and allow you to use them in your script.
2. Ensure that C:\Tektronix\RSA_API\lib\x64 is in your PATH environment variable. Don't delete anything from your PATH, just add the API directory to it.
3. Change to the API's directory and load the dll. We change the directory because the dll has other dependencies that reside in the directory in which it was installed. Use the following commands to do this:

   ```
   os.chdir("C:\\Tektronix\\RSA_API\\lib\\x64")
   rsa = cdll.LoadLibrary("RSA_API.dll")
   ```

   This creates an object that lets you access the dll and its methods. This is where the magic happens. Without this statement, everything I've been explaining before is useless information.

*See end of guide for more information about using a separate Python module to import structs and enums

Procedure for connecting to an RSA

Two-step process. Search and connect. There are some variables you need to prepare to do these things. The methods you'll use are *DEVICE_Search*() and *DEVICE_Connect*(). The variables you'll use are numFound, deviceIDs, and deviceSerial. All of this is laid out in the API documentation as follows:

DEVICE_Search	Searches for connectable devices (user buffers)
Declaration:	ReturnStatus DEVICE_Search(int* numDevicesFound, int deviceIDs[], char deviceSerial[][DEVSRCH_SERIAL_MAX_STRLEN], char deviceType[][DEVSRCH_TYPE_MAX_STRLEN]);
Parameters:	
numDevicesFound:	Pointer to an integer variable. Returns the number of devices found by the search call. A returned value of 0 indicates no devices found.
deviceIDs:	Returns an array of device ID numbers, numDevicesFound entries.
deviceSerial:	Returns an array of strings of device serial numbers, numDevicesFound entries. char or wchar_t strings are returned depending on the function used.
deviceType:	Returns an array of strings of device types, numDevicesFound entries. char or wchar_t strings are returned depending on the function used. Valid device type strings are: "RSA306", "RSA306B", "RSA503A", "RSA507A","RSA603A","RSA607A"

Notice that all the variables have their types listed. This is helpful when creating your variables because if you pass a variable with the wrong data type to a method, you'll get an error. Let's initialize these variables.

First we have numDevicesFound. I don't like long variables so I changed it to numFound here. It's an **int**, and by default I want zero devices to be found until I use *DEVICE_Search*() so I initialized numFound with a value of zero. See the first line in the image.

deviceIDs is an array of **ints**. Remember what I said earlier about creating arrays. Two steps: allocate memory and create variable. See the last 2 lines in the image of code below.

```
#search/connect variables
numFound = c_int(0)
intArray = c_int*10
deviceIDs = intArray()
```

deviceSerial and deviceType are both pointers to **c_char_t** arrays. ctypes has a handy way to preallocate string pointers in the *create_string_buffer*() function. Both Tektronix serial numbers and the available device types are no more than 7 characters long, so I preallocate both with 8 bytes of data.

```
deviceSerial = create_string_buffer(8)
deviceType = create_string_buffer(8)
```

Notice also that all the variables being passed to the *DEVICE_Search*() method are specified as pointers (go back and look at the API documentation on the previous page). When using *DEVICE_Search*(), you'll pass all of the variables by reference because the method changes the data contained in the variables

*See end of guide for more information about using a separate Python module to import structs and enums

with the information it gathers. Simply put *byref*() around each variable. After I've finished searching, I want to see how many devices were found. I've written some logic based on numFound, and you can write your own. In my case, I only have a single RSA306B connected, so if I find exactly one, I'll connect to it. Otherwise I'll print an error and exit the program.

```
#search/connect
rsa.DEVICE_Search(byref(numFound), deviceIDs, deviceSerial, deviceType)
if numFound.value < 1:
    print('No instruments found. Exiting script.')
    exit()
elif numFound.value == 1:
    print('One device found.')
    print('Device type: {}'.format(deviceType.value))
    print('Device serial number: {}'.format(deviceSerial.value))
    rsa.DEVICE_Connect(deviceIDs[0])
else:
    print('Unexpected number of devices found, exiting script.')
    exit()
```

Let's talk about *DEVICE_Connect*(). Its only parameter is deviceID, which is an **int**. We initialized deviceIDs earlier, which is an array of **ints**, so in order to get a single device ID rather than the entire array, we just need to send the first index of the array, deviceIDs[0]. Note that deviceID is NOT listed as a pointer in the function description. Because the method isn't changing the value of deviceID, there's no need to pass it by reference.

DEVICE_Connect	Connects to a device specified by the deviceID parameter.
Declaration:	ReturnStatus DEVICE_Connect(int deviceID);
Parameters:	
deviceID:	Device ID found during the Search function call.
Return Values:	
noError:	The device has been connected.
errorTransfer:	The POST status could not be retrieved from the device.
errorIncompatibleFirmware:	The firmware version is incompatible with the API version.
errorNotConnected:	The device is not connected.
Additional Detail:	The deviceID value must be found by the Search function call.

*See end of guide for more information about using a separate Python module to import structs and enums

Custom Data Structures and Enum Types

Certain API methods require the use of custom data structures to configure settings or get information from the RSA. These data structures are described in the API documentation. Data structures, or structs, are simply large custom data types made up of a mix of other fundamental data types. Of course, you could also have structs that are made up of other structs. Structs and enum types are almost always defined in a C header file, and you can either define them directly in your Python script or you can define them in a separate Python module and import that into your script*.

Let's take a look at the **Spectrum_Settings** struct outlined in the API documentation:

Item	Description
double span	Span measured in Hz
double rbw	Resolution bandwidth measured in Hz
bool enableVBW	Enables or disables VBW
double vbw	Video bandwidth measured in Hz
int traceLength	Number of trace points
SpectrumWindows window	Windowing method used for the transform
SpectrumVerticalUnits verticalUnit	Vertical units
double actualStartFreq	Actual start frequency in Hz
double actualStopFreq	Actual stop frequency in Hz
double actualFreqStepSize	Actual frequency step size in Hz
double actualRBW	Actual RBW in Hz
double actualVBW	Not used.
int actualNumIQSamples	Actual number of IQ samples used for transform

The data type is listed first and then the variable name. So the line that says "double span" specifies that the first member in the struct is a **double** called span. The second is a **double** called rbw, and so on and so forth. Things start to get interesting when you find custom data types within structs, which occurs here in the "SpectrumWIndows window" and "SpectrumVerticalUnits verticalUnit" lines. **SpectrumWindows** and **SpectrumVerticalUnits** are enumeration types, which simply means that they are **c_int** types with hard-coded values that refer to constants. It's kind of like an inside joke unless you have the handbook that tells you which constants the different hard-coded numbers refer to. I know that's a lot to swallow at once, but hopefully this example will help make sense of this madness. Following is the description of the **SpectrumVerticalUnits** enumeration type from the API documentation.

*See end of guide for more information about using a separate Python module to import structs and enums

SpectrumVerticalUnits	Value
SpectrumVerticalUnit_dBm	0
SpectrumVerticalUnit_Watt	1
SpectrumVerticalUnit_Volt	2
SpectrumVerticalUnit_Amp	3
SpectrumVerticalUnit_dBmV	4

If a function expecting a SpectrumVerticalUnits enum type receives a value of 0, it means that the vertical unit in the RSA is dBm. If it receives a value of 3, it means that the vertical unit in the RSA is Amps. Enumeration types assign meanings to numbers. Essentially, this means that you don't have to do anything fancy when initializing verticalUnit since it's just a number in the end.

Fortunately, you don't have to do anything really crazy to create a C-like struct or enum type in Python. It's just that in Python structs are categorized as **classes**. Creating a struct is a process of copying the description of the struct from the API documentation into a format that Python understands. Creating enums is very similar, but you don't have to specify types and you can assign values directly. I won't go into classes or inheritance because those are unnecessary when learning to use the API. Compare and contrast the struct/enum description with the code used to create the struct/enum. This is really a copy/paste/reorder project.

*See end of guide for more information about using a separate Python module to import structs and enums

Item	Description
double span	Span measured in Hz
double rbw	Resolution bandwidth measured in Hz
bool enableVBW	Enables or disables VBW
double vbw	Video bandwidth measured in Hz
int traceLength	Number of trace points
SpectrumWindows window	Windowing method used for the transform
SpectrumVerticalUnits verticalUnit	Vertical units
double actualStartFreq	Actual start frequency in Hz
double actualStopFreq	Actual stop frequency in Hz
double actualFreqStepSize	Actual frequency step size in Hz
double actualRBW	Actual RBW in Hz
double actualVBW	Not used.
int actualNumIQSamples	Actual number of IQ samples used for transform

```
"""#################CLASSES AND FUNCTIONS################"""
#create Spectrum_Settings data structure
class Spectrum_Settings(Structure):
    _fields_ = [('span', c_double),
    ('rbw', c_double),
    ('enableVBW', c_bool),
    ('vbw', c_double),
    ('traceLength', c_int),
    ('window', c_int),
    ('verticalUnit', c_int),
    ('actualStartFreq', c_double),
    ('actualStopFreq', c_double),
    ('actualFreqStepSize', c_double),
    ('actualRBW', c_double),
    ('actualVBW', c_double),
    ('actualNumIQSamples', c_double)]
```

*See end of guide for more information about using a separate Python module to import structs and enums

SpectrumVerticalUnits	Value
SpectrumVerticalUnit_dBm	0
SpectrumVerticalUnit_Watt	1
SpectrumVerticalUnit_Volt	2
SpectrumVerticalUnit_Amp	3
SpectrumVerticalUnit_dBmV	4

```
class SpectrumVerticalUnits:
    def __init__(self):
        self.SpectrumVerticalUnit_dBm = c_int(0)
        self.SpectrumVerticalUnit_Watt = c_int(1)
        self.SpectrumVerticalUnit_Volt = c_int(2)
        self.SpectrumVerticalUnit_Amp = c_int(3)
        self.SpectrumVerticalUnit_dBmV = c_int(4)
SpectrumVerticalUnits = SpectrumVerticalUnits()
```

Configuring Settings

Configuring settings on the RSA is fairly simple. Find the appropriate method that does what you want to do, create a variable that contains the value you want to set, and send that variable to the method and watch the API work its magic.

Let's use refLevel as an example. Say my signal is sitting at about -50 dBm and I want change the reference level from the default of 0 dBm to -40 dBm. The first thing to do is find the command that lets me change the reference level. The method is called *CONFIG_SetReferenceLevel*(). Finally, something that makes sense! I create my refLevel variable as a **double** according to the description and assign the value -40 to it. Then just pass refLevel to *CONFIG_SetReferenceLevel*() and you're off to the races. This same procedure can be used for any of the other settings functions. Sometimes these functions will take a struct as the argument rather than a single variable, but we already know how to make structs so that's an obstacle that can be easily overcome.

*See end of guide for more information about using a separate Python module to import structs and enums

CONFIG_SetReferenceLevel	Sets the reference level.
Declaration:	ReturnStatus CONFIG_SetReferenceLevel(double refLevel);
Parameters:	
refLevel:	Reference level measured in dBm. Range: –130 dBm to 30 dBm.
Return Values:	
noError:	The reference level value has been set.
errorNotConnected:	The device is not connected.
Additional Detail:	The reference level setting controls the signal path gain and attenuation settings. The value should be set to the maximum expected signal input power level, in dBm. Setting the value too low may result in over-driving the signal path and ADC, while setting it too high results in excess noise in the signal.

```
refLevel = c_double(-40)
rsa.CONFIG_SetReferenceLevel(refLevel)
```

Getting IQ Data

We're about to hit the ground running here, stay with me and we'll get through this together.

After we've configured the RSA with the appropriate settings (reference level, center frequency, span, RBW, sample rate, record length, etc.), we're ready to get some data. To get data, we have to tell the RSA to start acquiring. Mercifully, the function that does this is *DEVICE_Run*(), and it's literally this simple:

```
#start acquisition
rsa.DEVICE_Run()
```

After the RSA has acquired data, you can get the data from it. The only real parameter that we NEED to get IQ data from the API is record length, which we can set and get using *IQBLK_SetIQRecordLength*() and *IQBLK_GetIQRecordLength*(), respectively. It's helpful to have a variable that contains the actual record length to use in other parts of the program, so I would recommend setting the record length and then querying the value and storing it in a variable. I've done this for my example. Next, let's take a look at the *IQBLK_GetIQDataDeinterleaved*() method.

*See end of guide for more information about using a separate Python module to import structs and enums

IQBLK_GetIQDataDeinterleaved	Retrieves an IQ block data record in separate I and Q array format.
Declaration:	ReturnStatus IQBLK_GetIQDataDeinterleaved(float* iData, float* qData, int* outLength, int reqLength);
Parameters:	
iData:	Pointer to a float. Contains an array of I-data when the function completes.
qData:	Pointer to a float. Contains an array of Q-data when the function completes. The Q-data is not imaginary.
outLength:	Pointer to an integer variable. Returns the actual number of I and Q sample values returned in iData and qData buffers.
reqLength:	Number of IQ samples requested to be returned in iData and qData. The maximum value of reqLength is equal to the recordLength value set in IQBLK_SetIQRecordLength(). Smaller values of reqlLength allow retrieving partial IQ records.

There are three methods that give the user IQ data in different formats (interleaved, deinterleaved, and complex), and I chose deinterleaved because I want to have separate arrays for I and Q rather than putting them in the same array. We allocate memory and initialize the arrays for I and Q and create a variable for outLength, which is the actual number of samples returned by the function. Now we're close, but there are still a couple steps between us and the IQ data. Let's take a look at the *IQBLK_WaitForIQDataReady*() method.

IQBLK_WaitForIQDataReady	Waits for the data to be ready to be queried.
Declaration:	ReturnStatus IQBLK_WaitForIQDataReady(int timeoutMsec, bool* ready);
Parameters:	
timeoutMsec:	Timeout value measured in ms.
ready:	Pointer to a bool. Its value determines the status of the data.
	True indicates the data is ready for acquisition. False indicates the data is not ready and the timeout value is exceeded.
Return Values:	
noError:	The function has executed successfully.

This method tells us when the RSA is ready to give us valid IQ data. If we don't use this before sending the *IQBLK_GetIQDataDeinterleaved*() function, we'll get meaningless data back from the RSA. I initialize the variables needed, timeoutMsec as a **c_int** and ready as a **c_bool**, which can have a value of True or False. Next I use a simple *while*() loop to check the status of the IQ data, but you can check the status however you like. As soon as the value of ready is True, use *IQBLK_GetIQDataDeinterleaved*() and you've got the data. From this point you can perform whatever manual analysis you want on the IQ data.

*See end of guide for more information about using a separate Python module to import structs and enums

```
#data transfer variables
recordLength = c_int(1024)
actLength = c_int(0)
iqArray =  c_float*recordLength.value
iData = iqArray()
qData = iqArray()
timeoutMsec = c_int(1000)
ready = c_bool(False)

rsa.DEVICE_Run()

#check for data ready
while ready.value == False:
    rsa.IQBLK_WaitForIQDataReady(timeoutMsec, byref(ready))

#query I and Q data
rsa.IQBLK_GetIQDataDeinterleaved(byref(iData), byref(qData),
    byref(actLength), recordLength)
print('Got IQ data')
rsa.DEVICE_Stop()
```

Manipulating Transferred Data

This is the dreaded application section of the guide. It assumes you have read the rest of the document and read each example sequentially. Danger, I accept the terms and conditions, warnings/disclaimers, your mileage may vary, enter at your own risk, etc.

Plot IQ vs Time

So we've got the IQ data from the RSA, but if we want to do anything other than congratulate ourselves on having the data, we need to do some data processing. Because this section is so heavily application-dependent, I'll just go through a basic example of plotting I and Q vs time. The first thing we need to do in any post processing instance is convert ctypes arrays into arrays that Python can understand and manipulate. A common and very powerful numerical processing module for Python is called NumPy, and it's the one that I use in all my scripts where that capability is needed. To do plotting in Python, you'll need to import MatPlotLib. As a bonus, both NumPy and MatPlotLib are included in Anaconda, which you downloaded and installed at the very beginning of this document. To use the NumPy and MatPlotLib modules, you'll need to import them much like you imported ctypes at the beginning of your script.

```
from ctypes import *
import numpy as np
import matplotlib.pyplot as plt
```

Notice the import statement for these modules are a little different than that used for ctypes. The import statements are importing these modules and labeling them with shorter names. This means that when you want to use a NumPy method, you'll use the *np.bla*() format to call it. For example, if I want to create an array of numbers ranging from 1 through 5 exclusive using NumPy, it would look like this:

*See end of guide for more information about using a separate Python module to import structs and enums

```
48   bla = np.arange(1,5)
49   print(bla)
```

```
[1 2 3 4]
[Finished in 0.1s]
```

Application time. Convert the ctypes arrays (iData and qData) to NumPy arrays (I and Q) using the following method. Don't worry too much about the theory and details behind this command, just know that it works. Now you have your IQ data in a format that can actually be used.

```
#convert ctypes array to numpy array for ease of use
I = np.ctypeslib.as_array(iData)
Q = np.ctypeslib.as_array(qData)
```

We'll now create a new array that gives us timing information so we can plot I and Q vs Time. We need two pieces of information to create this array: sample rate and record length. Many of you are familiar with the relationship between acquisition time, sample rate, and record length, so I won't go into that here. You can query the sample rate using *IQBLK_GetIQSampleRate*()and the record length using *IQBLK_GetIQRecordLength*(). I suggested that you do this much earlier in the script, so you should have these variables available to you already without having to query their values again. You can use the *np.linspace*() function to create this array. If you're an avid Matlab user, the NumPy version of *linspace*() works the same way: *linspace*(start value, step size, number of points).

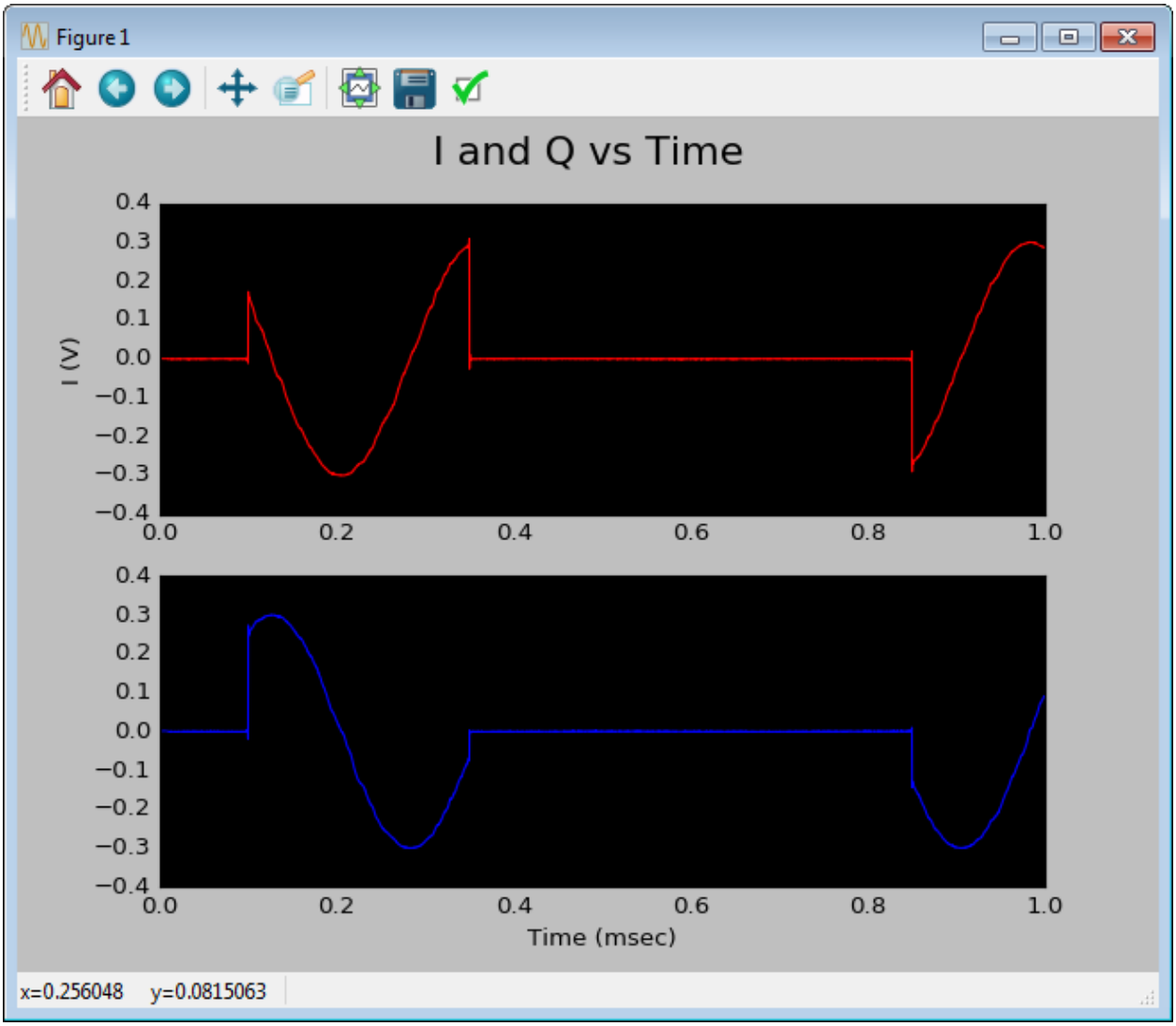
```
recordLength = c_int(0)
iqSampleRate = c_double(0)

rsa.IQBLK_GetIQSampleRate(byref(iqSampleRate))
rsa.IQBLK_GetIQRecordLength(byref(recordLength))
```

```
time = np.linspace(0,recordLength.value/iqSampleRate.value,recordLength.value)
```

Now we've got our I, Q, and time arrays and we can plot them. MatPlotLib works much like the plotting functions in Matlab (thus the name). We use the *subplot*(), *title*(), *plot*(), *xlabel*(), *ylabel*(), and *show*() methods from MatPlotLib. Again, if you're familiar with plotting in Matlab or Mathematica, they work the exact same way.

```
"""#################PLOTS#################"""
plt.suptitle('I and Q vs Time', fontsize='20')
plt.subplot(211, axisbg='k')
plt.plot(time*1e3, I, c='red')
plt.ylabel('I (V)')
plt.subplot(212, axisbg='k')
plt.plot(time*1e3, Q, c='blue')
plt.xlabel('Time (msec)')
plt.show()
```

*See end of guide for more information about using a separate Python module to import structs and enums



After you're all finished, *DEVICE_Disconnect*().

```
print('Disconnecting.')
ret = rsa.DEVICE_Disconnect()
```

Create Spectrum Plot

Although you can now say you've grabbed and plotted IQ data vs Time, that's not very useful. Let's look at another example, this time using spectrum traces rather than the raw IQ from the RSA.

Let's say your signal of interest is a 0 dBm tone at 1 GHz and you want to grab 20 MHz on either side of your signal. There are five settings we need to configure to get a good spectrum: reference level, center frequency, span, resolution bandwidth, and trace length. We know how to set reference level and center frequency since they have their own commands (*CONFIG_SetCenterFreq*() and *CONFIG_SetReferenceLevel*()), but span, RBW, and trace length are set a little differently.

*See end of guide for more information about using a separate Python module to import structs and enums

```
cf = c_double(1e9)              #center freq
refLevel = c_double(0)          #ref level

rsa.CONFIG_SetCenterFreq(cf)
rsa.CONFIG_SetReferenceLevel(refLevel)
```

Remember that time when we talked about the Spectrum_Settings struct? Well now you get to actually do something with it. First define the struct (see waaaaaay above), and create an instance. Before you populate the struct with the settings you want, you need to send *SPECTRUM_SetDefault*() and *SPECTRUM_GetSettings*() commands so that you have a Spectrum_Settings struct that is fully populated with default values. If you don't start with a fully populated struct and only change a couple things, you could be sending unknown numbers to the RSA and there's no telling how the API will handle it. This goes back to the discussion about pointers and memory protection, which is a large topic beyond the scope of this document. Back to business. In this case, we want to set span, RBW, and trace length. The span, rbw, and traceLength members in the struct can be accessed using the dot notation as shown below. You assign values to them based on the data types assigned to them in the struct definition (see discussion on creating structs many pages prior).

```
#create an instance of the Spectrum_Settings struct
specSet = Spectrum_Settings()

"""do other things"""

#configure desired spectrum settings
#some fields are left blank because the default
#values set by SPECTRUM_SetDefault() are acceptable
specSet.span = c_double(40e6)
specSet.rbw = c_double(30e3)
#specSet.enableVBW =
#specSet.vbw =
specSet.traceLength = c_int(801)
#specSet.window =
#specSet.verticalUnit =
#specSet.actualStartFreq =
#specSet.actualFreqStepSize =
#specSet.actualRBW =
#specSet.actualVBW =
#specSet.actualNumIQSamples =
```

Now that we've got that taken care of, we can send the commands that configure the spectrum trace. The methods we'll be using are *SPECTRUM_SetDefault*(), *SPECTRUM_SetEnable*(), *SPECTRUM_SetSettings*(), and *SPECTRUM_GetSettings*(). If you look up those commands in the API documentation, you'll see what variables need to be defined and used with these commands. *SPECTRUM_SetDefault*() doesn't need any arguments, *SPECTRUM_SetEnable*() needs a **bool**, and *SPECTRUM_SetSettings*() and *SPECTRUM_GetSettings*() both need the specSet struct we already created, so all we really need to do now is define enable.

```
enable = c_bool(True)           #easy
```

Then we just send the commands as follows. Remember that since *SPECTRUM_SetSettings*() is reading values from specSet and not changing anything, it doesn't need to be passed by reference.

*See end of guide for more information about using a separate Python module to import structs and enums

SPECTRUM_GetSettings() is actually changing the values contained in specSet to reflect the settings in the RSA, so it needs to be passed by reference. Why do we need to get the settings we just sent? Because we will be getting out more information than we put in. Remember all those lines that were commented out when we defined span, rbw, and traceLength? We'll be getting all those back from *SPECTRUM_GetSettings*(). Some of these parameters are critical for visualizing the spectrum trace, including but not limited to actualStartFreq, actualFreqStepSize, and traceLength. I wish it was simpler but it isn't. SORRY.

```
rsa.SPECTRUM_SetEnable(enable)
rsa.SPECTRUM_SetDefault()
rsa.SPECTRUM_GetSettings(byref(specSet))

#set desired spectrum settings
rsa.SPECTRUM_SetSettings(specSet)
rsa.SPECTRUM_GetSettings(byref(specSet))
```

We also need to prepare for the day when we eventually get the trace data. Let's take a look at the documentation for *SPECTRUM_GetTrace*() to see what we need.

SPECTRUM_GetTrace — This function queries the spectrum trace data.

Declaration: ReturnStatus SPECTRUM_GetTrace(SpectrumTraces trace, int maxTracePoints, float *traceData, int *outTracePoints);

Parameters:

trace: One of the spectrum trace.

SpectrumTraces	Value
SpectrumTrace1	0
SpectrumTrace2	1
SpectrumTrace3	2

maxTracePoints: Maximum number of trace points to be retrieved. The traceData array should be at least this size.

traceData: Return spectrum trace data. The trace data is in the unit of verticalUnit specified in the Spectrum_Settings structure.

outTracePoints: Pointer to int. Returns the actual number of valid trace points in traceData array.

Return Values:

noError: The trace data has been successfully queried.

Alright, it looks like we need to prepare a few variables. Remember the function declaration lists each argument's **type** followed by the variable. Because **SpectrumTraces** trace is an enumeration type, we can either create our own enum type or just read the definition and send the number that corresponds to the active trace. In this case we are using default spectrum settings, which activates Trace 1, so we send a value of zero or use the enum type we created. maxTracePoints is just a number, and we've already gotten this number from *SPECTRUM_GetSettings*() in the form of specSet.traceLength, so we can just send that directly without having to create a new variable. traceData is just an array of **floats.** We've made arrays before to contain I and Q data, so creating another such array should be easy. Finally, outTracePoints is simply an **int** that tells us the number of valid points in traceData. Also easy.

*See end of guide for more information about using a separate Python module to import structs and enums

```
traceArray = c_float * specSet.traceLength
traceData = traceArray()
outTracePoints = c_int()        #boom, done
```

Next we need to create a frequency array to give the spectrum trace points a meaningful horizontal axis. Remember when I mentioned we would be using actualStartFreq, actualFreqStepSize, and traceLength earlier? Their time is now. We'll create a range of frequencies that correspond to the span in our spectrum trace using the NumPy function *arange*(start, end, step). And we'll populate start, end, and step using, you guessed it, combinations of actualStartFreq, actualFreqStepSize, and traceLength. It looks really complicated, but it's just on multiple lines because variable names used by the API are long. SORRY. You can change them if you want when you create the variables and structs.

```
#generate frequency array for plotting the spectrum
freq = np.arange(specSet.actualStartFreq,
    specSet.actualStartFreq + specSet.actualFreqStepSize*specSet.traceLength,
    specSet.actualFreqStepSize)
```

Next we'll follow the same procedure as we did in the previous example to determine when the trace data is ready to be grabbed. While ready is False, send the *SPECTRUM_WaitForDataReady*() command and move on as soon as ready is True. After the data is ready, we need to go get it. We've already prepared everything we need for *SPECTRUM_GetTrace*(), so let's go do it! Send a value of 0 (or SpectrumTraces.SpectrumTrace1) for the trace field, send specSet.traceLength for the maxTracePoints field, etc. etc. etc. You don't need handholding to read through a function declaration any more. Just remember to pass by reference any variables that will be changed by the function.

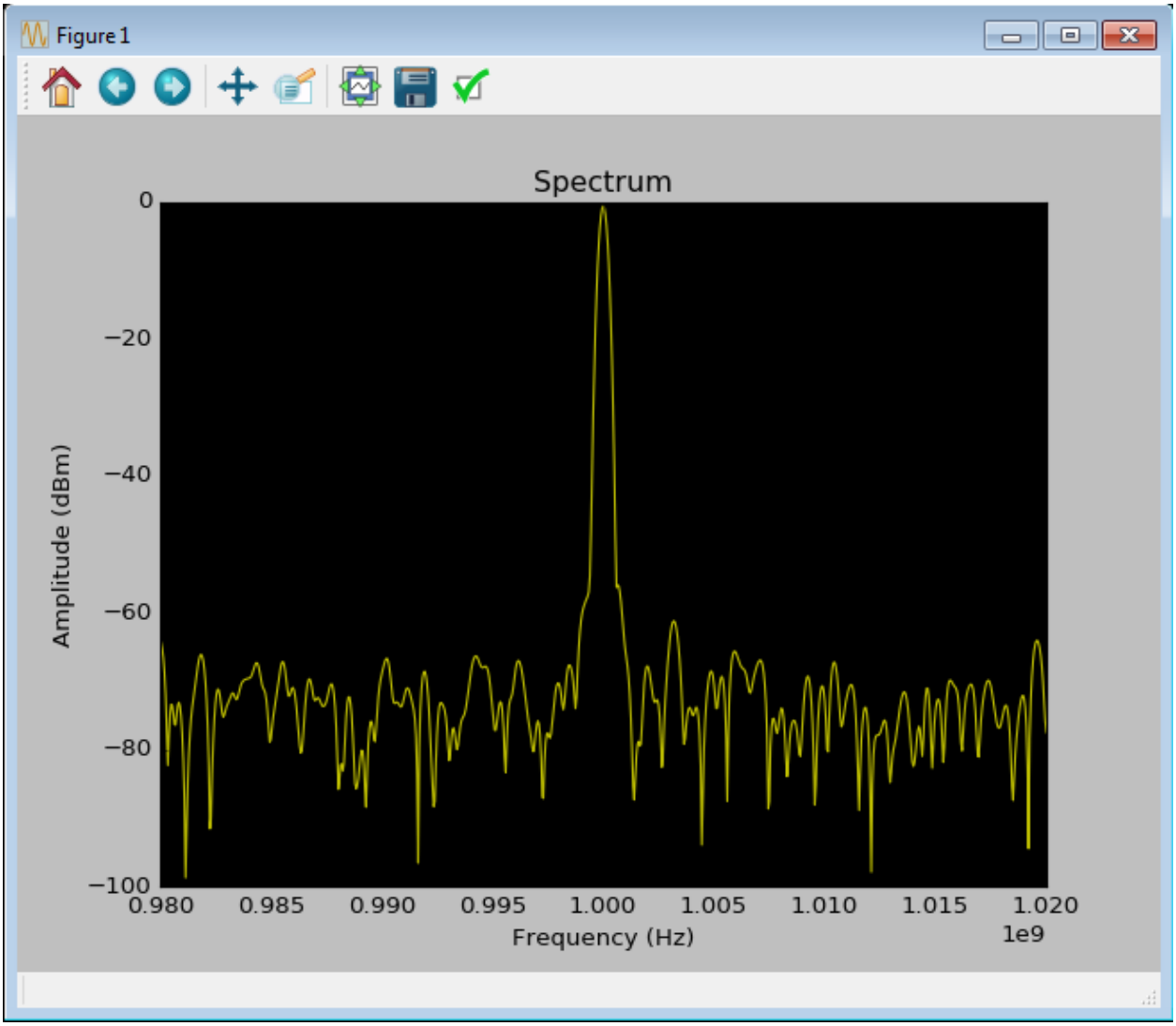
```
"""##################ACQUIRE/PROCESS DATA##################"""
#start acquisition
rsa.DEVICE_Run()
while ready.value == False:
    rsa.SPECTRUM_WaitForDataReady(timeoutMsec, byref(ready))

rsa.SPECTRUM_GetTrace(c_int(0), specSet.traceLength,
    byref(traceData), byref(outTracePoints))
rsa.DEVICE_Stop()
```

Awesome, we have trace data! We don't need the RSA to be running any more so we can send the *DEVICE_Stop*() command. Now to plot traceData. We'll need to convert traceData to a NumPy array just like we did with the I and Q arrays in the last example so we can plot it in Python. All that's left after that is to plot it out just like we did in the last example. Am I repeating myself?

```
"""##################SPECTRUM PLOT##################"""
#plot the spectrum trace (optional)
plt.figure(1)
plt.subplot(111, axisbg='k')
plt.plot(freq, traceData, 'y')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Amplitude (dBm)')
plt.title('Spectrum')
```

*See end of guide for more information about using a separate Python module to import structs and enums

You're done. Here are the fruits of your labor.



Peak Power Detector

This is just a useful little utility that can be easily added to your spectrum trace script. It's just a few lines of code.

```
#Peak power and frequency calculations
peakPower = np.amax(trace)
peakPowerFreq = freq[np.argmax(trace)]
print('Peak power in spectrum: %4.3f dBm @ %d Hz' % (peakPower, peakPowerFreq))
```

np.amax() is a function that finds the largest value in an array and returns it. So we're storing the highest value from trace in peakPower. That gives us the max power, but for this utility to be really useful we want to be able to tell the user the frequency at which the max power occurred. *np.argmax*() tells us the index at which the largest value in an array occurs. We can use that index from trace to find the

*See end of guide for more information about using a separate Python module to import structs and enums

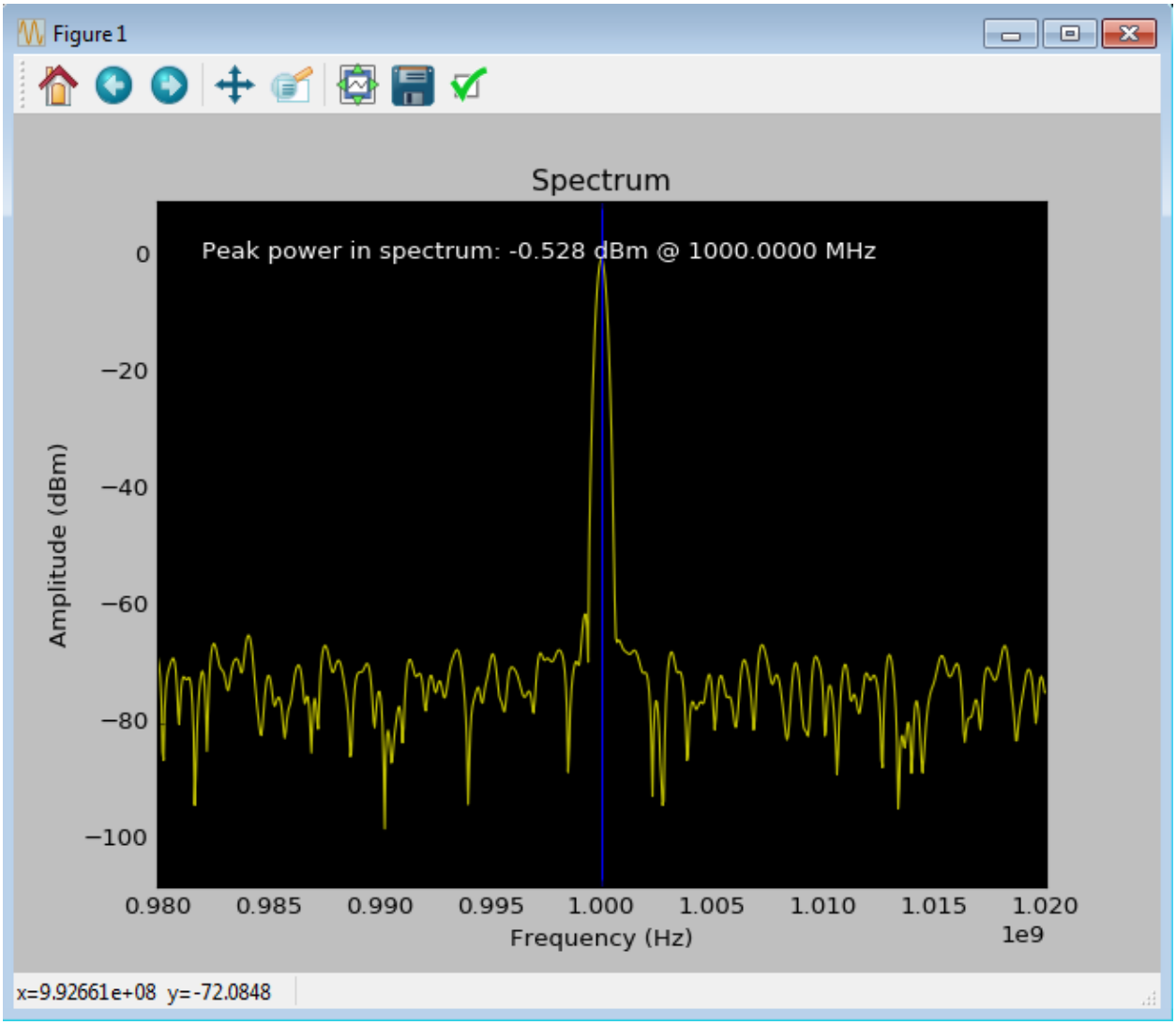
corresponding point in freq, which will give us exactly what we're looking for. To do that, we simply use *np.argmax*(trace) as the index we want to access in freq. Let's try it.

```
c:\Tektronix\RSA306 API\lib\x64>python peakpowerdetector.py
One device found.
Device Serial Number: B010225
Trace Data is Ready
Got trace data.
Peak power in spectrum: -0.835 dBm @ 500000000 Hz
Disconnecting.
```

You can also annotate your spectrum plot to contain a vertical line at the peak power location with text showing the measurement data. I've also cleaned up the plot axes a little bit in the code below.

```
#annotate measurement
plt.axvline(x=peakPowerFreq)
text_x = specSet.actualStartFreq + specSet.span/20
plt.text(text_x, peakPower,
    'Peak power in spectrum: %4.3f dBm @ %5.4f MHz' % (peakPower,
        peakPowerFreq/1e6), color='white')

#BONUS clean up plot axes
xmin = np.amin(freq)
xmax = np.amax(freq)
plt.xlim(xmin,xmax)
ymin = np.amin(trace)-10
ymax = np.amax(trace)+10
plt.ylim(ymin,ymax)

plt.show()
```

*See end of guide for more information about using a separate Python module to import structs and enums



There you have it, an easy peak power detector!

*See end of guide for more information about using a separate Python module to import structs and enums

How to Use the RSA API in Python

Stream IQ Data to a .TIQ File

Some applications require the user to stream IQ data. Fortunately the API lets you do this and this example will show you how.

Three notes before we begin.

1. IQ streaming operates in two modes. The first mode is streaming the data to the client application/script/program. The second mode is streaming the data to a file on the controlling computer's hard drive. The user can select the file format in which the data is saved: .tiq, .siq, or .sidh/siqd. The .tiq option is the native file format for SignalVu-PC and it can be recalled in SignalVu-PC directly. The .siq option has a 1024 byte ASCII header that contains information such as sample rate and file size followed by raw binary data. The .siqh/.siqd option simply splits off the header and data into different files. Regardless of the streaming mode or file format, the user can select the data type of the streamed data: 32-bit single, 32-bit integer, and 16-bit integer. In this example we'll be talking about streaming data to a .tiq file.

2. Any changes to streaming settings (data destination, file type, bandwidth, etc.) should only be done while the instrument is not acquiring data. Any changes made during acquisition will not go into effect until the instrument stops and then starts acquiring again.

3. Order of operations is important. *DEVICE_Run*() absolutely must be sent before *IQSTREAM_Start*(). Since *DEVICE_Run*() is the master control for the RSA acquisition system, it needs to be sent before any command that gathers or accesses acquired data. If you send *IQSTREAM_Start*() before *DEVICE_Run*(), you'll get garbage data or no data at all. Switching the order of these two commands causes and fixes 90% of the problems that have been reported to me regarding IQ streaming.

With that out of the way, let's take a look at the commands we will use from the IQSTREAM family. For streaming setup we will need *IQSTREAM_SetAcqBandwidth*() and *IQSTREAM_GetAcqParameters*(). For file management we will need *IQSTREAM_SetOutputConfiguration*(), *IQSTREAM_SetDiskFilenameBase*(), *IQSTREAM_SetFilenameSuffix*(), and *IQSTREAM_SetDiskFileLength*(). For streaming control we will need *IQSTREAM_Start*(), *IQSTREAM_Stop*(), and *IQSTREAM_GetDiskFileWriteStatus*().

*See end of guide for more information about using a separate Python module to import structs and enums

Let's set up our streaming acquisition bandwidth. The *IQSTREAM_SetAcqBandwidth*() function sends the RSA an acquisition bandwidth request using a **c_double** called bwHz_req and the RSA matches it as closely as it can. There are four acquisition bandwidth "steps" that are outlined in the documentation for *IQSTREAM_GetAcqParameters*(). These bandwidth values are 40 MHz, 20 MHz, 10 MHz, and 5 MHz and they directly affect the effective sample rate and the write speed requirement for saving streamed data to a hard drive (see table below command descriptions). Both the actual bandwidth and sample rate (**c_doubles** called bwHz_act and srSps respectively) are returned by IQ*STREAM_GetAcqParameters*().

IQSTREAM_SetAcqBandwidth	Sets the users request for the acquisition bandwidth of the output IQ data stream samples.
Declaration:	ReturnStatus IQSTREAM_SetAcqBandwidth(double bwHz_req);
Parameters:	
bwHz_req:	Requested acquisition bandwidth of IQ Streaming output data, in Hz.
Return Values:	
noError:	The requested value was accepted
Additional Detail:	No checking of the input value is done by this function. See the table in IQSTREAM_GetAcqParameters() for the mapping of requested bandwidth to actual output bandwidth provided. ***NOTE.*** *The Acq Bandwidth setting should only be changed when the instrument is in the* ***global*** *Stopped state. The new BW setting does not take effect until the global system state is cycled from Stopped to Running.*

IQSTREAM_GetAcqParameters	Reports the processing parameters of IQ output bandwidth and sample rate resulting from the users requested bandwidth.
Declaration:	ReturnStatus IQSTREAM_GetAcqParameters(double* bwHz_act, double* srSps);
Parameters:	
bwHz_act:	Pointer to a double. Returns actual acquisition bandwidth of IQ Streaming output data, in Hz.
srSps:	Pointer to a double. Returns actual sample rate of IQ Streaming output data, in Samples/sec
Return Values:	
noError:	The query was successful.
Additional Detail:	This is the mapping of requested bandwidth to actual output bandwidth and sample rate.

Requested BW	**Output BW**	**Output Sample Rate**
BW ≤ 5 MHz	5 MHz	7.0 Msps
5 MHz < BW ≤ 10 MHz	10 MHz	14.0 Msps
10 MHz < BW ≤ 20 MHz	20 MHz	28.0 Msps
BW > 20 MHz	40 MHz	56.0 Msps

*See end of guide for more information about using a separate Python module to import structs and enums

		IQ Output Data rate	
IQ BW	*IQ Sample Rate*	*32b fixed or float*	*16b fixed*
40 MHz	56 M Sa/sec	448 M B/sec	224 M B/sec
20 MHz	28 M Sa/sec	224 M B/sec	112 M B/sec
10 MHz	14 M Sa/sec	112 M B/sec	56 M B/sec
5 MHz	7 M Sa/sec	56 M B/sec	28 M B/sec

Next is *IQSTREAM_SetOutputConfiguration*(). This function tells the RSA3where to stream the data, either directly to the client to be stored in a variable for real-time processing or to a file on the computer's SSD. The parameters dest and dtype are both enumeration types, which leaves us with a couple options: either send the corresponding **c_int** value or create and use an enum type. The options are pretty clear from the function description and correspond to the file types described at the beginning of this example. The API documentation has a .siq file description in the IQ Streaming section.

IQSTREAM_SetOutputConfiguration — Sets the output data destination and IQ data type.

Declaration:

ReturnStatus IQSTREAM_SetOutputConfiguration(IQSOUTDEST dest, IQSOUTDTYPE dtype);

Parameters:

dest: Destination (sink) for IQ sample output. Valid settings:

dest value	Destination
IQSOD_CLIENT	Client application
IQSOD_FILE_TIQ	TIQ format file (.tiq extension)
IQSOD_FILE_SIQ	SIQ format file with header and data combined in one file (.siq extension)
IQSOD_FILE_SIQ_SPLIT	SIQ format with header and data in separate files (.siqh and .siqd extensions)

dtype: Output IQ data type. Valid settings:

dtype value	Data type
IQSODT_SINGLE	32-bit single precision floating point (not valid with TIQ file destination)
IQSODT_INT32	32-bit integer
IQSODT_INT16	16-bit integer

Return Values:

noError: The requested settings were accepted.

errorIQStreamInvalidFile-DataType: Invalid selection of TIQ file and Single data type together.

Additional Detail:

The destination can be the client application, or files of different formats. The IQ data type can be chosen independently of the file format. IQ data values are stored in interleaved I/Q/I/Q order regardless of the destination or data type.

NOTE. *TIQ format files only allow Int32 or Int16 data types, not Single.*

*See end of guide for more information about using a separate Python module to import structs and enums

These next few commands are configuring the file to which the data is saved. You'll make a **c_char_p** called filenameBase and populate it with the destination of your saved file as shown below.

```
filenameBase = c_char_p('C:\SignalVu-PC Files\sample')
```

Sets the base filename for file output can be accomplished with similar functions. These functions are grouped together.	
IQSTREAM_SetDiskFilenameBase	Sets the base filename for file output (char string)
Declaration:	ReturnStatus IQSTREAM_SetDiskFilenameBase(const char* filenameBase);
Parameters:	
filenameBase:	Base filename for file output. This can include drive/path, as well as the common base filename portion of the file. The filename base should not include a file extension, as the file writing operation will automatically append the appropriate one for the selected file format.
Return Values:	
noError:	The setting was accepted.
Additional Detail:	The complete output filename has the following format: <filenameBase><suffix><.ext> <filenameBase>: as set by this function <suffix>: as set by filename suffix control in IQSTREAM_SetDiskFilename-Suffix() <.ext>: as set by destination control in IQSTREAM_SetOutputConfigura-tion(), [.tiq, .siq, .siqh+.siqd] If separate data and header files are generated, the same path/filename is used for both, with different extensions to indicate the contents.

*See end of guide for more information about using a separate Python module to import structs and enums

The API gives you options for file name suffixes: either nothing, a millisecond-resolution timestamp, or a simple incrementing 5-digit number. Again suffixCtl is an enumeration type so you'll just assign or send a **c_int**.

IQSTREAM_SetDiskFilenameSuffix — Sets the control that determines what, if any, filename suffix is appended to the output base filename.

Declaration: ReturnStatus IQSTREAM_SetDiskFilenameSuffix(int suffixCtl);

Parameters:

suffixCtl: Sets the filename suffix control value.

Return Values:

noError: The setting was accepted.

Additional Detail: See description of IQSTREAM_SetDiskFilename() for the full filename format.

suffixCtl value	Suffix generated
IQSSDFN_SUFFIX_NONE (-2)	None. Base filename is used without suffix. (Note that the output filename will not change automatically from one run to the next, so each output file will overwrite the previous one unless the filename is explicitly changed by calling the Set function again.)
IQSSDFN_SUFFIX_TIMESTAMP (-1)	String formed from file creation time Format: "-YYYY.MM.DD.hh.mm.ss.msec" (Note this time is not directly linked to the data timestamps, so it should not be used as a high-accuracy timestamp of the file data!)
≥ 0	5 digit auto-incrementing index, initial value = suffixCtl. Format: "-nnnnn" (Note index auto-increments by 1 each time IQSTREAM_Start() is invoked with file data destination setting.)

*See end of guide for more information about using a separate Python module to import structs and enums

And finally *IQSTREAM_SetDiskFileLength*() gives you control over how long the file is. Simply create a **c_int** containing the desired file duration in milliseconds and send it as the argument. If the argument is 0, the file will continue increasing in size until *IQSTREAM_Stop*() is called. However, if the time length is reached/exceeded, *IQSTREAM_Stop*() will be called automatically and streaming will cease.

IQSTREAM_SetDiskFileLength	Sets the time length of IQ data written to an output file.
Declaration:	ReturnStatus IQSTREAM_SetDiskFileLength(int msec);
Parameters:	
msec:	Length of time in milliseconds to record IQ samples to file.
Return Values:	
noError:	The setting was accepted.
Additional Detail:	See IQSTREAM_GetDiskFileWriteStatus to find how to monitor file output status to determine when it is active and completed.

msec value	File store behavior
0	No time limit on file output. File storage is terminated when IQSTREAM_Stop() is called.
> 0	File output ends after this number of milliseconds of samples stored. File storage can be terminated early by calling IQSTREAM_Stop().

*See end of guide for more information about using a separate Python module to import structs and enums

Speaking of starting and stopping acquisitions, let's talk about *IQSTREAM_Start*() and its counterpart *IQSTREAM_Stop*(). These are very simple and behave much like *Run*() and *Stop*(), but instead of controlling the entire acquisition system they only control the IQ streaming system.

IQSTREAM_Start	Initializes IQ Stream processing and initiates data output.
Declaration:	ReturnStatus IQSTREAM_Start();
Parameters:	
(none):	
Return Values:	
noError:	The operation was successful
errorBufferAllocFailed:	Internal buffer allocation failed (memory unavailable)
errorIQStreamFileOpen-Failed:	Output file open (create) failed.
Additional Detail:	If the data destination is the client application, data will become available soon after the Start() function is invoked. Even if triggering is enabled, the data will begin flowing to the client without need for a trigger event. The client must begin retrieving data as soon after Start() as possible. If the data destination is file, the output file is created, and if triggering is not enabled, data starts to be written to the file immediately. If triggering is enabled, data will not start to be written to the file until a trigger event is detected. TRIG_ForceTrigger() can be used to generate a trigger event if the specified one does not occur.

IQSTREAM_Stop	This function terminates IQ Stream processing and disables data output.
Declaration:	ReturnStatus IQSTREAM_Stop();
Parameters:	
(none):	
Return Values:	
noError:	The operation was successful.
Additional Detail:	If the data destination is file, file writing is stopped and the output file is closed.

And finally it is always nice to check the status of an operation, and *IQSTREAM_GetDiskFileWriteStatus*() allows us to do just that. This function takes two **c_bool**s, isComplete and isWriting. When this function is called, the API looks at the streaming status and changes the values of isComplete and isWriting to reflect that status, which is why the two arguments need to be sent by reference. isComplete is False while the streaming operation is still in progress and changes to True when the file has been output successfully. In general isWriting is always True after *IQSTREAM_Start*() is called. The exception is when the user has configured a triggered acquisition. If both *Run*() and *IQSTREAM_Start*() have been called but the trigger has not been seen by the RSA, isWriting will be False until it sees a trigger. Again this is only the case when using a triggered acquisition. This functions are useful if you want to poll the streaming status periodically, which is what I have done in this example. Take a look at the documentation for *IQSTREAM_GetFileWriteStatus*() since it has additional information.

*See end of guide for more information about using a separate Python module to import structs and enums

IQSTREAM_GetDiskFileWriteStatus	Allows monitoring the progress of file output.
Declaration:	ReturnStatus IQSTREAM_GetDiskFileWriteStatus(bool* isComplete, bool *isWriting);
Parameters:	
isComplete:	Pointer to a bool. Returns whether the IQ Stream file output writing complete.
isWriting:	Pointer to a bool. Returns whether the IQ Stream processing has started writing to file (useful when triggering is in use). (Input NULL if no return value is desired).
Return Values:	
noError:	The query was successful.

Now on to business. This example assumes you have already found and connected to the RSA. Now all we need to do is set up our acquisition and IQ streaming parameters. Our signal of interest is at 1 GHz and we want to stream 20 MHz of IQ bandwidth for 100 ms. Note that we've also initialized bwHz_act and sRate so we can get information about how the RSA set itself up based on our input.

```
cf = c_double(1e9)
refLevel = c_double(0)
bwHz_req = c_double(5e6)
bwHz_act = c_double(0)
sRate = c_double(0)
durationMsec = c_long(1000)
```

Next we configure our streaming parameters. We want to save an autoincrementing .tiq file with int16 data to C:\SignalVu-PC Files so we create our filename, set dest to 1, dtype to 2, and suffixCtl to 3. We've created complete and writing to update the streaming status as well as some loop control variables that you'll see in use later.

*See end of guide for more information about using a separate Python module to import structs and enums

```
class IQSOUTDEST:
    def __init__(self):
        self.IQSOD_CLIENT = c_int(0)
        self.IQSOD_FILE_TIQ = c_int(1)
        self.IQSOD_FILE_SIQ = c_int(2)
        self.IQSOD_FILE_SIQ_SPLIT = c_int(3)
IQSOUTDEST = IQSOUTDEST()

class IQSOUTDTYPE:
    def __init__(self):
        self.IQSODT_SINGLE = c_int(0)
        self.IQSODT_INT32 = c_int(1)
        self.IQSODT_INT16 = c_int(2)
IQSOUTDTYPE = IQSOUTDTYPE()

IQSSDFN_SUFFIX_INCRINDEX_MIN = c_int(0)
IQSSDFN_SUFFIX_TIMESTAMP = c_int(-1)
IQSSDFN_SUFFIX_NONE = c_int(-2)

filenameBase = 'C:\\SignalVu-PC Files\\sample'
waitTime = 0.1
streaming = True
writing = c_bool(False)
complete = c_bool(False)
loopCount = 0
dest = IQSOUTDEST.IQSOD_FILE_TIQ
dtype = IQSOUTDTYPE.IQSODT_INT16
suffixCtl = IQSSDFN_SUFFIX_NONE
```

After we've connected to the RSA, we configure it and then start streaming.

```
rsa.CONFIG_Preset()
rsa.CONFIG_SetCenterFreq(cf)
rsa.CONFIG_SetReferenceLevel(refLevel)

rsa.IQSTREAM_SetAcqBandwidth(bwHz_req)
rsa.IQSTREAM_GetAcqParameters(byref(bwHz_act), byref(sRate))
rsa.IQSTREAM_SetOutputConfiguration(dest, dtype)
rsa.IQSTREAM_SetDiskFilenameBase(filenameBase)
rsa.IQSTREAM_SetDiskFilenameSuffix(suffixCtl)
rsa.IQSTREAM_SetDiskFileLength(durationMsec)
```

Remember the importance of order of operations with *Run*() and *IQSTREAM_Start*().

```
rsa.DEVICE_Run()
rsa.IQSTREAM_Start()
```

And now the streaming loop. This example simply stops the script for waitTime milliseconds and then uses *IQSTREAM_GetDiskFileWriteStatus*() to give us the streaming status and uses the value from complete to tell us when the file is finished streaming. When it's finished, streaming is set to False and the program exits the loop. If you like, you can have a loop variable that tells you how many times your script checked the write status. I like doing this for a sanity check. Because we all need those from time to time.

*See end of guide for more information about using a separate Python module to import structs and enums

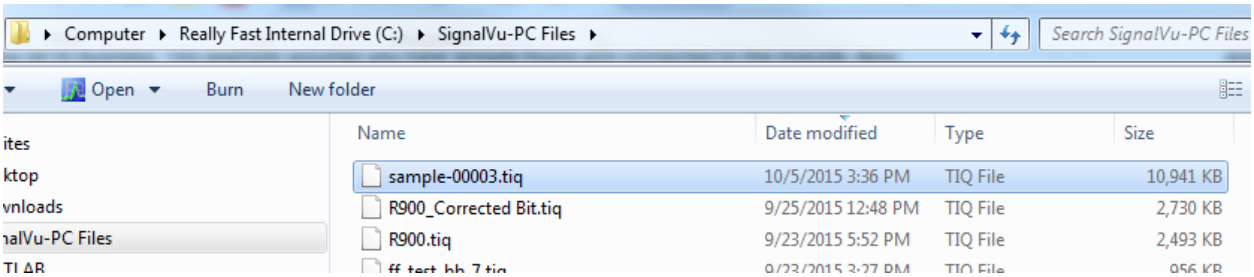
```
while streaming == True:
    time.sleep(waitTime)
    rsa.IQSTREAM_GetDiskFileWriteStatus(byref(complete), byref(writing))
    print(complete)
    if complete.value == True:
        streaming = False
    loopCount += 1
```

Print out your status info and gracefully disconnect.

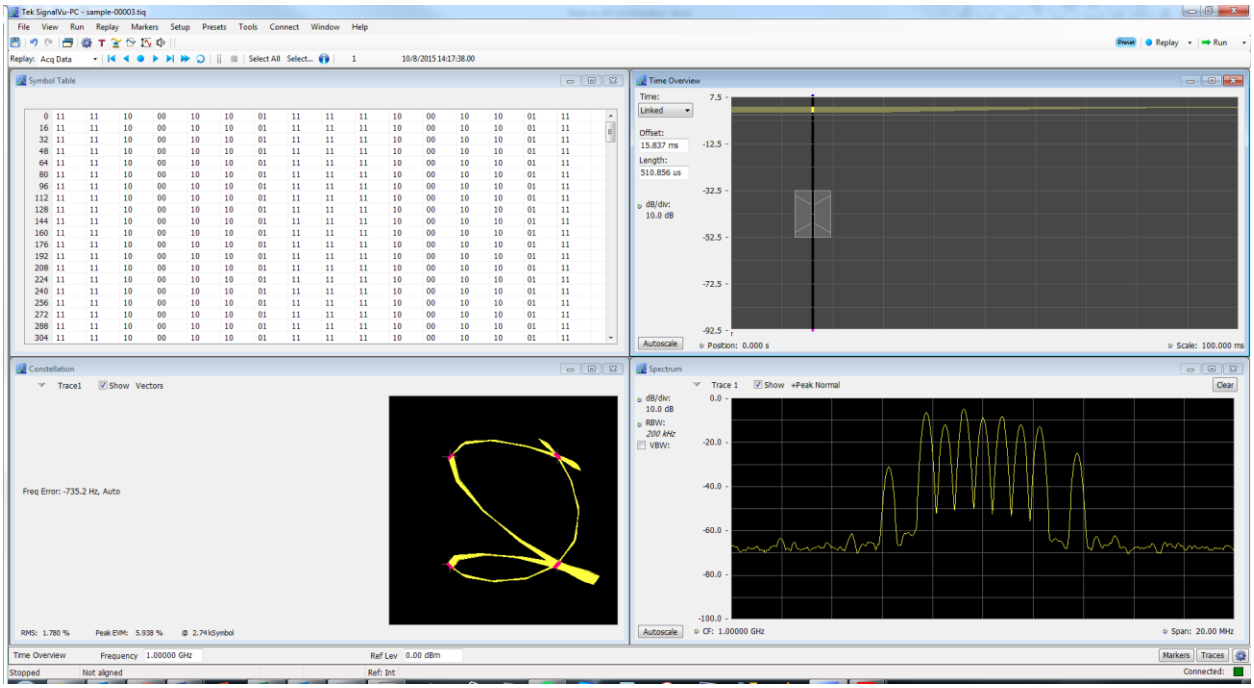
```
print('# of loops: {}'.format(loopCount))
print('FIle saved at {}'.format(filenameBase.value))
print('Disconnecting.')
rsa.DEVICE_Disconnect()
```

*See end of guide for more information about using a separate Python module to import structs and enums

Check your destination folder for success.



Load the .tiq file into SignalVu-PC for data validation.



Congratulations, you've got another skill to put in your bag of programming tricks. You've also successfully used the API and are now an expert.

*See end of guide for more information about using a separate Python module to import structs and enums

Bonus Content

For full example code, visit https://github.com/blistergeist/rsa_api_python_36

The code at this repository contains the content we covered in this document and more. The code is a two-file project that has a script file (rsa_api_full_example.py) that defines functions and runs the examples and a separate Python module (RSA_API.py) that defines all the structs and enum types used by the API.

To run the code, either clone the repository onto your computer and run the script from the cloned directory or download the files to the same directory on your computer, open a command prompt at that directory and run “python rsa_apy_full_example.py” and you’re off to the races.

*See end of guide for more information about using a separate Python module to import structs and enums