

# Stop.

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 API will mitigate confusion and remove the need to explain the process of installing Python modules.

This document assumes you have a basic knowledge of programming (using variables, calling functions, knowledge of data types, etc.). If this is not the case, I would recommend going through Learn Python the Hard Way: <http://learnpythonthehardway.org/book/>

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

## Intro

All of the magic in the API happens in RSA300API.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/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. After finding and connecting to your RSA306, the API provides you with an RSA306 *object* that has a whole bunch of functions (or *methods* to be semantically precise) that you can use to send commands to and get data from the connected RSA306. In all these examples, my object is called `rsa300`. In order to access/use any of the *methods* contained in the *object*, you need to use the *object.method()* notation. For example, if I wanted to use *Disconnect()* on my RSA306, I would write the following:

```
rsa300.Disconnect()
```

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

## Working with ctypes variables.

Creating a variable in Python using a C data type is fairly easy after you've imported ctypes. Let's use the refLevel variable as an example. Based on the API documentation, all functions that utilize refLevel expect to see a **double**. To create this variable as a **double** with a value of 0, we would just type the following:

```
refLevel = c_double(0)
```

## How to Use the RSA306 API in Python

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

```
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 the variable, you need to use the *.value* method. Take the following scenario for example:

```
from ctypes import *
bla = c_double(102)
dingus = bla + 17
print(dingus)
```

If I run this code, it gives me an error because Python refuses to add a standard Python **int** data type with a **c\_double** data type.

```
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* method (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]
```

## 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 computer memory location. 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.

## How to Use the RSA306 API in Python

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 costly time and memory by copying. 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, it can manipulate the variable itself. For example, if I wanted to set the reference level of the RSA306, I could just pass a value because I don't want the dll to do anything to the refLevel we created earlier. In another example, if I wanted to get the reference level from the RSA306, I would want to pass the 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.

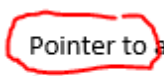
```
rsa300.GetCenterFreq(byref(cf))
```

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).

#### GetCenterFreq

ReturnStatus GetCenterFreq(double\* cf);

##### Parameters:

*cf*:  Pointer to a double. It will contain the center frequency when the function completes. Its value will range from 0Hz to 6.2GHz.

### Necessary components of an API script

1. Save your script in the same directory as RSA300API.dll. For this example, it's installed in C:\Program Files\Tektronix\RSA306\RSA306 API or C:\Tektronix\RSA306 API\lib\x64, so you'll need to save your script in one of those locations.
2. 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"? what is this madness?), this statement is simply telling Python to load every method from the ctypes module and allow you to use them in your script.

3. Load the dll. To do this type the following:

```
rsa300 = WinDLL("RSA306API.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.

### 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 *Search()* and *Connect()*. The variables you'll use are deviceIDs, deviceSerial, and numFound. All of this is laid out in the API documentation as follows:

#### Search

ReturnStatus Search(long deviceIDs[], wchar\_t\* deviceSerial[], int\* numDevicesFound);

##### Parameters:

<i>deviceIDs:</i>	Pointer to a long. It will contain an array of device ID numbers.
<i>deviceSerial:</i>	Pointer to a wchar_t array. It will contain an array of device serial numbers.
<i>numDevicesFound:</i>	Pointer to an integer. It will contain the number of devices found by the search call.

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

deviceSerial is a pointer to a **wchar\_t** array. Variables in the **char** data type family are a bit odd, and ctypes treats **c\_wchar\_p** and **c\_char\_p** data types as preallocated arrays of **chars**, so there's no need to preallocate the memory yourself. I know it's weird, but just go with it for now.

## How to Use the RSA306 API in Python

Because I haven't found any devices yet, I want the serial number to be blank so I initialized it with an empty string. See the third line in the image below.

And finally 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 `Search()` so I initialized `numFound` with a value of zero. See the fourth line in the image.

```
longArray = c_long*10
deviceIDs = longArray()
deviceSerial = c_wchar_p('')
numFound = c_int(0)
```

Notice also that all the variables being passed to the `Search()` method are specified as pointers (go back and look at the API documentation on the previous page). When using `Search()`, you'll pass all of the variables by reference because the method changes the data contained in the variables 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. In my case, I only have one RSA306 connected, so if I find exactly one, I'll connect to it. Otherwise I'll print an error and exit the program.

```
rsa300.Search(byref(deviceIDs), byref(deviceSerial), byref(numFound))
if numFound.value == 1:
    rsa300.Connect(deviceIDs[0])
else:
    print('Unexpected number of instruments found.')
    exit()
```

Let's talk about `Connect()`. Its only parameter is `deviceId`, which is a `long`. We initialized `deviceIDs` earlier, which is an array of `longs`, 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.

### Connect

ReturnStatus Connect(long deviceId);

#### Parameters:

*deviceId*: Device ID found during the Search function call.

#### Return Values:

*noError*: The device has been connected.

## Custom Data Structures

Certain API methods require the use of custom data structures to configure settings or get information from the RSA306. These data structures are described in the API documentation, and if the current

## How to Use the RSA306 API in Python

customer-facing version doesn't contain this information, it should be added immediately. 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.

Anyway, let's take a look at the **Spectrum\_Settings** struct outlined in the API documentation:

<b>Spectrum_Settings</b>
double span
double rbw
bool enableVBW
double vbw
int traceLength
SpectrumWindows window
SpectrumVerticalUnits verticalUnit
double actualStartFreq
double actualFreqStepSize
double actualRBW
double actualVBW
double actualNumIQSamples

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 values refer to. I know that's a lot to swallow at once, but hopefully this example will help make sense of this madness. Below is the description of the **SpectrumVerticalUnits** enumeration type from the API documentation.

<b>SpectrumVerticalUnits</b>	<b>Value</b>
SpectrumVerticalUnit_dBm	0
SpectrumVerticalUnit_Watt	1
SpectrumVerticalUnit_Volt	2
SpectrumVerticalUnit_Amp	3
SpectrumVerticalUnit_dBmV	4

If a variable of this type has a value of 0, it means that the vertical unit in the RSA is dBm. If the variable has a value of 3, it means that the vertical unit in the RSA306 is Amps. Enumeration types assign meanings to numbers. Since Python doesn't have an elegant way of handling enumeration types, you don't have to do anything fancy when initializing verticalUnit since it's just going to be a number in the end.

## How to Use the RSA306 API in Python

Fortunately, you don't have to do anything really crazy to create a C-like struct 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. I won't go into classes or inheritance because those are unnecessary when learning to use the API. Compare and contrast the struct description with the code used to create the struct. This is a copy/paste/reorder project.

<b>Spectrum_Settings</b>
double span
double rbw
bool enableVBW
double vbw
int traceLength
SpectrumWindows window
SpectrumVerticalUnits verticalUnit
double actualStartFreq
double actualFreqStepSize
double actualRBW
double actualVBW
double actualNumIQSamples

```
#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),
                ('actualFreqStepSize', c_double), ('actualRBW', c_double),
                ('actualVBW', c_double), ('actualNumIQSamples', c_double)]
```

## Configuring Settings

Configuring settings on the RSA306 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 shove that variable into the method and watching 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 *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 *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.

## How to Use the RSA306 API in Python

### SetReferenceLevel

ReturnStatus SetReferenceLevel(double refLevel);

#### Parameters:

*refLevel:* Reference level measured in dB. Its value can range from -130dB to 30dB.

```
refLevel = c_double(-40)
rsa300.SetReferenceLevel(refLevel)
```

## Getting IQ Data

We're about to hit the ground running here. I've thrown a lot at you so far, but 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 RSA306 to start acquiring. Mercifully, the function that does this is *Run()*, and it's literally this simple:

```
rsa300.Run()
```

After the RSA306 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 *SetIQRecordLength()* and *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 *GetIQDataDeinterleaved()* method.

### GetIQDataDeinterleaved

ReturnStatus GetIQDataDeinterleaved(float\* iData, float\* qData, int startIndex, int length);

#### Parameters:

*iData:* Pointer to a float. It will contain an array of I-data when the function completes.

*qData:* Pointer to a float. It will contain an array of Q-data when the function completes. The Q-data is not imaginary.

*startIndex:* Starting index of the IQ record . The sum of the startIndex and length must be less than the IQ record length.

*length:* Amount of samples of IQ data to acquire. The sum of the startIndex and length must be less than the IQ record length.



## How to Use the RSA306 API in Python

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 initialize startIndex to 0 because we want to start getting data from the beginning of the record. Now we're close, but there are still a couple steps between us and the IQ data. Let's take a look at the *WaitForIQDataReady()* method.

### WaitForIQDataReady

ReturnStatus 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.

#### Return Values:

*noError:* The function has executed successfully.

#### Description:

This function waits for the data to be ready to be queried. If the data is not ready and the timeout value is exceeded, the *ready* parameter will be false. Otherwise, the data is ready for acquisition and the *ready* parameter will be true.

This method tells us when the RSA is ready to give us valid IQ data. If we don't use this before sending the *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 *GetIQDataDeinterleaved()* and you've got the data. From this point you can perform whatever manual analysis you want on the IQ data.

## How to Use the RSA306 API in Python

```
recordLength = c_int(0)
rsa300.GetIQRecordLength(byref(recordLength))

iqArray = c_float*recordLength.value
iData = iqArray()
qData = iqArray()
startIndex = c_int(0)
timeoutMsec = c_int(100)
ready = c_bool(False)

rsa300.Run()
while ready.value == False:
    rsa300.WaitForIQDataReady(timeoutMsec, byref(ready))
    print('IQ Data is Ready')

rsa300.GetIQDataDeinterleaved(byref(iData), byref(qData), startIndex, recordLength)
print('Got IQ data')
```

## Manipulating Transferred Data

This is the dreaded application section of the guide. Danger, I accept the terms and conditions, warnings/disclaimers, actual results may vary, enter at your own risk, etc.

So we've got the IQ data from the RSA306, 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:

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

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

## How to Use the RSA306 API in Python

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 *GetIQSampleRate()* and the record length using *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).

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

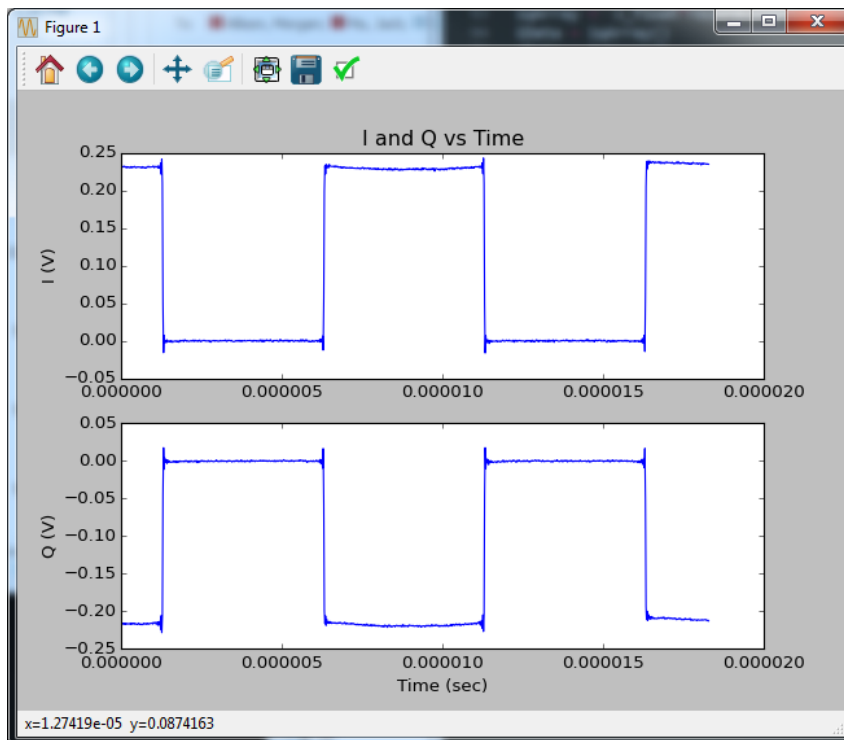
rsa300.GetIQRecordLength(byref(recordLength))
rsa300.GetIQSampleRate(byref(iqSampleRate))
```

```
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 or Mathematica (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.

```
plt.subplot(2,1,1)
plt.title('I and Q vs Time')
plt.plot(time,I)
plt.ylabel('I (V)')
plt.subplot(2,1,2)
plt.plot(time,Q)
plt.ylabel('Q (V)')
plt.xlabel('Time (sec)')
plt.show()
```

## How to Use the RSA306 API in Python



After you're all finished, `Disconnect()`.

```
print('Disconnecting.')  
ret = rsa300.Disconnect()
```

## How to Use the RSA306 API in Python

### 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 RSA306.

Let's say your signal of interest is a 0 dBm tone at 500 MHz and you want to grab 10 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 (*SetCenterFreq()* and *SetReferenceLevel()*), but span, RBW, and trace length are set a little differently.

```
cf = c_double(500e6)      #set center freq
refLevel = c_double(0)    #set ref level

rsa300.SetCenterFreq(cf)
rsa300.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 RSA306 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 =
```

## How to Use the RSA306 API in Python

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. `SPECTRUM_GetSettings()` is actually changing the values contained in `specSet` to reflect the settings in the RSA306, 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.

```
rsa300.SPECTRUM_SetEnable(enable)
rsa300.SPECTRUM_SetDefault()
rsa300.SPECTRUM_GetSettings(byref(specSet))
#change individual values
rsa300.SPECTRUM_SetSettings(specSet)
rsa300.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.

<b>SPECTRUM_GetTrace</b>	This function queries the spectrum trace data.								
<b>Declaration:</b>	ReturnStatus SPECTRUM_GetTrace(SpectrumTraces trace, int maxTracePoints, float *traceData, int *outTracePoints);								
<b>Parameters:</b>									
<i>trace:</i>	One of the spectrum trace.								
	<table><tr><th>SpectrumTraces</th><th>Value</th></tr><tr><td>SpectrumTrace1</td><td>0</td></tr><tr><td>SpectrumTrace2</td><td>1</td></tr><tr><td>SpectrumTrace3</td><td>2</td></tr></table>	SpectrumTraces	Value	SpectrumTrace1	0	SpectrumTrace2	1	SpectrumTrace3	2
SpectrumTraces	Value								
SpectrumTrace1	0								
SpectrumTrace2	1								
SpectrumTrace3	2								
<i>maxTracePoints:</i>	Maximum number of trace points to be retrieved. The traceData array should be at least this size.								
<i>traceData:</i>	Return spectrum trace data.								
<i>outTracePoints:</i>	Pointer to int. Returns the actual number of valid trace points in traceData array.								
<b>Return Values:</b>									
<i>noError:</i>	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 just read the definition and send the number that corresponds to the active trace. In this case we

## How to Use the RSA306 API in Python

are using default spectrum settings, which activates Trace 1, so we send a value of zero. 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 **ints**. 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.

```
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. 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 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.

```
rsa300.Run()
while ready.value == False:
    rsa300.SPECTRUM_WaitForDataReady(timeoutMsec, byref(ready))
print('Trace Data is Ready')

rsa300.SPECTRUM_GetTrace(c_int(0), specSet.traceLength,
                        byref(traceData), byref(outTracePoints))
print('Got trace data.')
```

Awesome, we have trace data! We don't need the RSA to be running any more so we can send the *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?

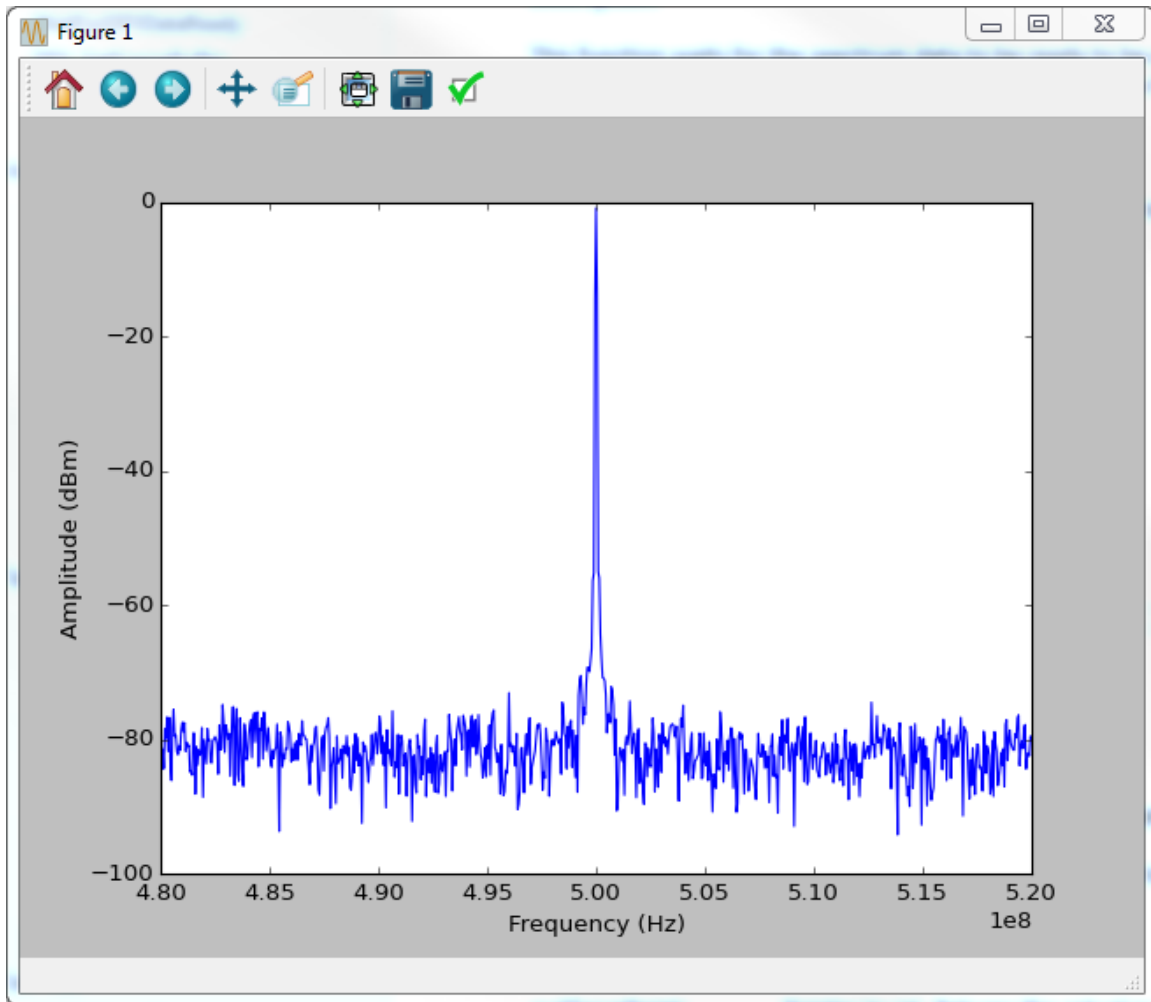
## How to Use the RSA306 API in Python

```
#stop acquisition
rsa300.Stop()

#convert trace data from a ctypes array to a numpy array
trace = np.ctypeslib.as_array(traceData)

#plot the spectrum trace (optional)
plt.plot(freq, traceData)
plt.xlabel('Frequency (Hz)')
plt.ylabel('Amplitude (dBm)')
plt.show()
```

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.



## How to Use the RSA306 API in Python

```
#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 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.
```

There you have it, an easy peak power detector!

Congratulations! You've successfully used the API and are now an expert.

## How to Use the RSA306 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 and .siqd. The .tiq option is the native file format for SignalVu-PC, and iff the file contains less than one second of saved data 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. *Run()* absolutely must be sent before *IQSTREAM\_Start()*. Since *Run()* is the master control for the RSA306 acquisition system, it needs to be sent before any command that gathers or accesses acquired data. If you send *IQSTREAM\_Start()* before *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()*, *IQSTREAMSetDiskFilenameBase()*, *IQSTREAM\_SetFilenameSuffix()*, and *IQSTREAM\_SetDiskFileLength()*. For streaming control we will need *IQSTREAM\_Start()*, *IQSTREAM\_Stop()*, and *IQSTREAM\_GetDiskFileWriteStatus()*.

## How to Use the RSA306 API in Python

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 `QSTREAM_GetAcqParameters()`.

Table 10: IQ Streaming functions

<b>IQSTREAM_SetAcqBandwidth</b>	<p>This function sets the users request for the acquisition bandwidth of the output IQ data stream samples. No checking of the input value is done by this function. See the table in <code>IQSTREAM_GetAcqParameters()</code> for the mapping of requested bandwidth to actual output bandwidth provided.</p> <p><i>NOTE: The Acq Bandwidth setting should only be changed when the instrument is in the <b>global</b> Stopped state. The new BW setting does not take effect until the global system state is cycled from Stopped to Running.</i></p> <p><b>Declaration:</b>  <b>Parameters:</b>  <i>bwHz_req:</i>  <b>Return Values:</b>  <i>noError:</i></p>	<p>ReturnStatus <code>IQSTREAM_SetAcqBandwidth(double bwHz_req);</code></p> <p>Requested acquisition bandwidth of IQ Streaming output data, in Hz.</p> <p>The requested value was accepted</p>															
<b>IQSTREAM_GetAcqParameters</b>	<p>This function reports the processing parameters of IQ output bandwidth and sample rate resulting from the users requested bandwidth. The table below gives the mapping of requested bandwidth to actual output bandwidth and sample rate.</p> <table border="1"> <thead> <tr> <th>Requested BW</th><th>Output BW</th><th>Output Sample Rate</th></tr> </thead> <tbody> <tr> <td><math>BW \leq 5 \text{ MHz}</math></td><td>5 MHz</td><td>7.0 Msps</td></tr> <tr> <td><math>5 \text{ MHz} &lt; BW \leq 10 \text{ MHz}</math></td><td>10 MHz</td><td>14.0 Msps</td></tr> <tr> <td><math>10 \text{ MHz} &lt; BW \leq 20 \text{ MHz}</math></td><td>20 MHz</td><td>28.0 Msps</td></tr> <tr> <td><math>BW &gt; 20 \text{ MHz}</math></td><td>40 MHz</td><td>56.0 Msps</td></tr> </tbody> </table> <p><b>Declaration:</b>  <b>Parameters:</b>  <i>bwHz_act:</i>  <i>srSps:</i>  <b>Return Values:</b>  <i>noError:</i></p>	Requested BW	Output BW	Output Sample Rate	$BW \leq 5 \text{ MHz}$	5 MHz	7.0 Msps	$5 \text{ MHz} < BW \leq 10 \text{ MHz}$	10 MHz	14.0 Msps	$10 \text{ MHz} < BW \leq 20 \text{ MHz}$	20 MHz	28.0 Msps	$BW > 20 \text{ MHz}$	40 MHz	56.0 Msps	<p>ReturnStatus <code>IQSTREAM_GetAcqParameters(double* bwHz_act, double* srSps);</code></p> <p>Ptr-to-double. Returns actual acquisition bandwidth of IQ Streaming output data, in Hz.</p> <p>Ptr-to-double. Returns actual sample rate of IQ Streaming output data, in Samples/sec</p> <p>The query was successful</p>
Requested BW	Output BW	Output Sample Rate															
$BW \leq 5 \text{ MHz}$	5 MHz	7.0 Msps															
$5 \text{ MHz} < BW \leq 10 \text{ MHz}$	10 MHz	14.0 Msps															
$10 \text{ MHz} < BW \leq 20 \text{ MHz}$	20 MHz	28.0 Msps															
$BW > 20 \text{ MHz}$	40 MHz	56.0 Msps															

		<i><b>IQ Output Data rate</b></i>	
<i><b>IQ BW</b></i>	<i><b>IQ Sample Rate</b></i>	<i><b>32b fixed or float</b></i>	<i><b>16b fixed</b></i>
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

## How to Use the RSA306 API in Python

Next is `IQSTREAM_SetOutputConfiguration()`. This function tells the RSA306 where 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 that Python doesn't handle very well, so you just need to define them as `c_int` numbers or simply send a `c_int` instead of a variable like we have done before. 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**

This function sets the output data destination and IQ data type. 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*.

#### **Declaration:**

`ReturnStatus IQSTREAM_SetOutputConfiguration(IQSOUTDEST dest, IQSOUTDTYPE dtype);`

#### **Parameters:**

*dest:*

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

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

*dtype:*

Output IQ data type. Valid settings:

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

#### **Return Values:**

*noError:*

The requested settings were accepted.

*error/QStreamInvalidFile-Data Type:*

Invalid selection of TIQ file and Single data type together.

---

## How to Use the RSA306 API in Python

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')
```

Table 10: IQ Streaming functions (cont.)

<b>IQSTREAM_SetDiskFilenameBase</b>	<p>This function sets the base filename for file output.</p> <p>The complete output filename has the following format:</p> <p>&lt;filenameBase&gt;&lt;suffix&gt;&lt;.ext&gt;</p> <p>&lt;filenameBase&gt;: as set by this function</p> <p>&lt;suffix&gt;: as set by filename suffix control in IQSTREAM_SetDiskFilename-Suffix()</p> <p>&lt;.ext&gt;: as set by destination control in IQSTREAM_SetOutputConfigura-tion(), [ .tiq, .siq, .siqh+.siqd]</p> <p>If separate data and header files are generated, the same path/filename is used for both, with different extensions to indicate the contents.</p> <p>ReturnStatus = IQSTREAM_SetDiskFilenameBaseW(const wchar_t* filenameBaseW)</p>
<b>Declaration:</b>	ReturnStatus IQSTREAM_SetDiskFilenameBase(const char* filenameBase);
<b>Parameters:</b>	
<i>filenameBase:</i>	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. Wide (2 byte) or standard (byte) char string function versions are available. Other than accepting different type strings, they do the same operation.
<b>Return Values:</b>	
<i>noError:</i>	The setting was accepted.

## How to Use the RSA306 API in Python

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`.

Table 10: IQ Streaming functions (cont.)

### IQSTREAM\_SetDiskFilenameSuffix

This function sets the control that determines what, if any, filename suffix is appended to the output base filename. See description of `IQSTREAM_SetDiskFilename()` for the full filename format.

<i>suffixCtl</i> value	Suffix generated
<code>IQSSDFN_SUFFIX_NONE</code> (-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 <i>Set</i> function again.)
<code>IQSSDFN_SUFFIX_TIMESTAMP</code> (-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!)
$\geq 0$	5 digit auto-incrementing index, initial value = <i>suffixCtl</i> . Format: "-nnnnn" (Note index auto-increments by 1 each time <code>IQSTREAM_Start()</code> is invoked with file data destination setting.)

Below are examples of output filenames generated with different `suffixCtl` settings. Multiple filenames show suffix auto-generation behavior with each `IQSTREAM_Start()`. The most recent `suffixCtl` setting remain in effect until changed by another function call.

(Assume <filenameBase> is "myfile" and TIQ file format is selected.)

<i>suffixCtl</i> value	Full Filename (and behavior with multiple runs)
<code>IQSSDFN_SUFFIX_NONE</code>	"myfile.tiq" "myfile.tiq" "myfile.tiq" ...
<code>IQSSDFN_SUFFIX_TIMESTAMP</code>	"myfile-2015.04.15.09.33.12.522.tiq" "myfile-2015.04.15.09.33.14.697.tiq" "myfile-2015.04.15.09.33.17.301.tiq" ...
10	"myfile-00010.tiq" "myfile-00011.tiq" "myfile-00012.tiq" ...
4	"myfile-00004.tiq" "myfile-00005.tiq" ...

Declaration:

`ReturnStatus IQSTREAM_SetDiskFilenameSuffix(int suffixCtl);`

Parameters:

*suffixCtl*:

Sets the filename suffix control value.

Return Values:

*noError*:

The setting was accepted.

## How to Use the RSA306 API in Python

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.

Table 10: IQ Streaming functions (cont.)

<b>IQSTREAM_SetDiskFileLength</b>	<p>This function sets the time length of IQ data written to an output file. See <i>IQSTREAM_GetDiskFileWriteStatus()</i> to find how to monitor file output status to determine when it is active and completed.</p> <table border="1"> <thead> <tr> <th>msec value</th><th>File store behavior</th></tr> </thead> <tbody> <tr> <td>0</td><td>No time limit on file output. File storage is terminated when <i>IQSTREAM_Stop()</i> is called.</td></tr> <tr> <td>&gt; 0</td><td>File output ends after this number of milliseconds of samples stored. File storage can be terminated early by calling <i>IQSTREAM_Stop()</i>.</td></tr> </tbody> </table>	msec value	File store behavior	0	No time limit on file output. File storage is terminated when <i>IQSTREAM_Stop()</i> is called.	> 0	File output ends after this number of milliseconds of samples stored. File storage can be terminated early by calling <i>IQSTREAM_Stop()</i> .
msec value	File store behavior						
0	No time limit on file output. File storage is terminated when <i>IQSTREAM_Stop()</i> is called.						
> 0	File output ends after this number of milliseconds of samples stored. File storage can be terminated early by calling <i>IQSTREAM_Stop()</i> .						
<b>Declaration:</b>	ReturnStatus IQSTREAM_SetDiskFileLength(int msec);						
<b>Parameters:</b>							
msec:	Length of time in milliseconds to record IQ samples to file.						
<b>Return Values:</b>							
noError:	The setting was accepted.						

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.

<b>IQSTREAM_Start()</b>	<p>This function initializes IQ Stream processing and initiates data output. 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. ForceTrigger() can be used to generate a trigger event if the specified one does not occur.</p>
<b>Declaration:</b>	ReturnStatus IQSTREAM_Start();
<b>Parameters:</b>	
(none):	
<b>Return Values:</b>	
noError:	The operation was successful
errorBufferAllocFailed:	Internal buffer allocation failed (memory unavailable)
errorIQStreamFileOpenFailed:	Output file open (create) failed.
<b>IQSTREAM_Stop()</b>	<p>This function terminates IQ Stream processing and disables data output. If the data destination is file, file writing is stopped and the output file is closed.</p>
<b>Declaration:</b>	ReturnStatus IQSTREAM_Stop();
<b>Parameters:</b>	
(none):	
<b>Return Values:</b>	
noError:	The operation was successful.



## How to Use the RSA306 API in Python

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\_bools**, *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 RSA306, *isWriting* will be False until it sees a trigger. Again this is only the case when using a triggered acquisition. These functions are useful if you want to poll the streaming status periodically, which is what I have done in this example.

---

<b>IQSTREAM_GetDiskFileWriteStatus()</b>	<p>This function allows monitoring the progress of file output. The returned values indicate when the file output has started and completed. These become valid after <i>IQSTREAM_Start()</i> is invoked, with any file output destination selected.</p> <p><i>isComplete</i>:</p> <p>    false: indicates that file output is not complete.     true: indicates file output is complete.</p> <p><i>isWriting</i>:</p> <p>    false: indicates file writing is not in progress.     true: indicates file writing is in progress. When untriggered, this value is true immediately after <i>Start()</i> is invoked.</p> <p>For untriggered configuration, <i>isComplete</i> is all that needs to be monitored. When it switches from false-&gt;true, file output has completed. Note that if "infinite" file length is selected (file length parameter msec=0), <i>isComplete</i> only changes to true when the run is stopped with <i>IQSTREAM_Stop()</i>.</p> <p>If triggering is used, <i>isWriting</i> can be used to determine when a trigger has been received. The client application can monitor <i>isWriting</i>, and if a maximum wait period has elapsed while it is still false, the output operation can be aborted. <i>isWriting</i> behaves the same for both finite and infinite file length settings.</p> <p>The indicators sequence is as follows (assumes a finite file length setting):</p> <p>Untriggered operation:</p> <p>    <i>IQSTREAM_Start()</i></p> <p>        =&gt; File output in progress: [<i>isComplete</i> =false, <i>isWriting</i> =true]         =&gt; File output complete: [<i>isComplete</i> =true, <i>isWriting</i> =true]</p> <p>Triggered operation:</p> <p>    <i>IQSTREAM_Start()</i></p> <p>        =&gt; Waiting for trigger, File writing not started: [<i>isComplete</i>=false, <i>isWriting</i> =false]         =&gt; Trigger event detected, File writing in progress: [<i>isComplete</i>=false, <i>isWriting</i> =true]         =&gt; File output complete: [<i>isComplete</i>=true, <i>isWriting</i> =true]</p>
<b>Declaration:</b>	ReturnStatus <i>IQSTREAM_GetDiskFileWriteStatus</i> (bool* <i>isComplete</i> , bool* <i>isWriting</i> );
<b>Parameters:</b>	
<i>isComplete</i> :	Ptr-to-bool. Returns whether the IQ Stream file output writing complete.
<i>isWriting</i> :	Ptr-to-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).
<b>Return Values:</b>	
<i>noError</i> :	The query was successful.

---



## How to Use the RSA306 API in Python

Now on to business. This example assumes you have already found and connected to the RSA306. 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 RSA306 set itself up based on our input.

```
"""Main SA parameters"""
cf = c_double(1e9)
refLevel = c_double(0)
bwHz_req = c_double(20e6)
bwHz_act = c_double(0)
sRate = c_double(0)
durationMsec = c_int(100)
```

Next we configure our streaming parameters. We want to save an autoincrementing .tiq file with int32 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.

```
"""Stream Control Variables"""
filenameBase = c_char_p('C:\SignalVu-PC Files\sample')
#dest: 0 = client, 1 = .tiq, 2 = .siq, 3 = .siqd/.siqh
dest = c_int(1)
#dtype: 0 = single, 1 = int32, 2 = int16
dtype = c_int(2)
#SuffixCtl: 0 = none, 1 = YYYY.MM.DD.hh.mm.ss.msec, 3 = -xxxxx autoincrement
suffixCtl = c_int(3)
#streaming status boolean variable
complete = c_bool(False)
#write status boolean variable (always true if non-triggered acquisition)
writing = c_bool(False)
#time to wait between streaming loopCounts
waitTime = durationMsec.value/2/1e3
#bool used for streaming loopCount control
streaming = True
loopCount = 0
```

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

```
"""Configure Settings"""
rsa300.Preset()
rsa300.SetCenterFreq(cf)
rsa300.SetReferenceLevel(refLevel)
rsa300.IQSTREAM_SetAcqBandwidth(bwHz_req)
rsa300.IQSTREAM_GetAcqParameters(byref(bwHz_act), byref(sRate))
rsa300.IQSTREAM_SetOutputConfiguration(dest, dtype)
rsa300.IQSTREAM_SetDiskFilenameBase(filenameBase)
rsa300.IQSTREAM_SetDiskFilenameSuffix(suffixCtl)
rsa300.IQSTREAM_SetDiskFileLength(durationMsec)
```

Remember the importance of order of operations with *Run()* and *IQSTREAM\_Start()*.

```
rsa300.Run()
rsa300.IQSTREAM_Start()
```

## How to Use the RSA306 API in Python

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.

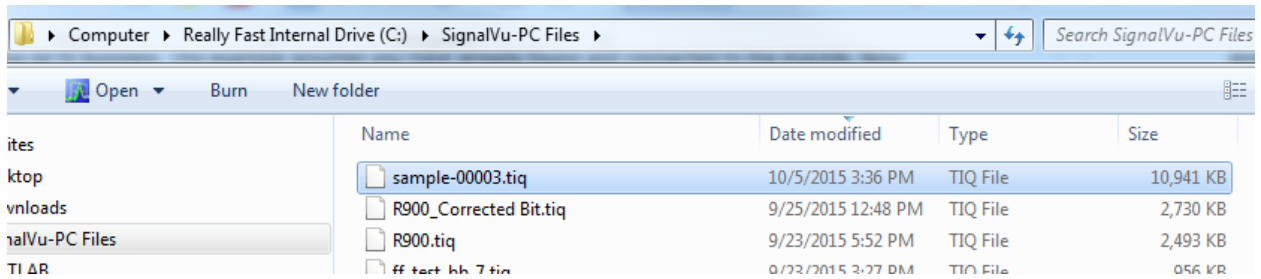
```
#Streaming control loop example, feel free to make your own
while streaming == True:
    time.sleep(waitTime)
    rsa300.IQSTREAM_GetDiskFileWriteStatus(byref(complete), byref(writing))
    #print('Complete? {}'.format(complete.value))
    #print('Writing? {}'.format(writing.value))
    if complete.value == True:
        streaming = False
    loopCount += 1
```

Print out your status info and gracefully disconnect.

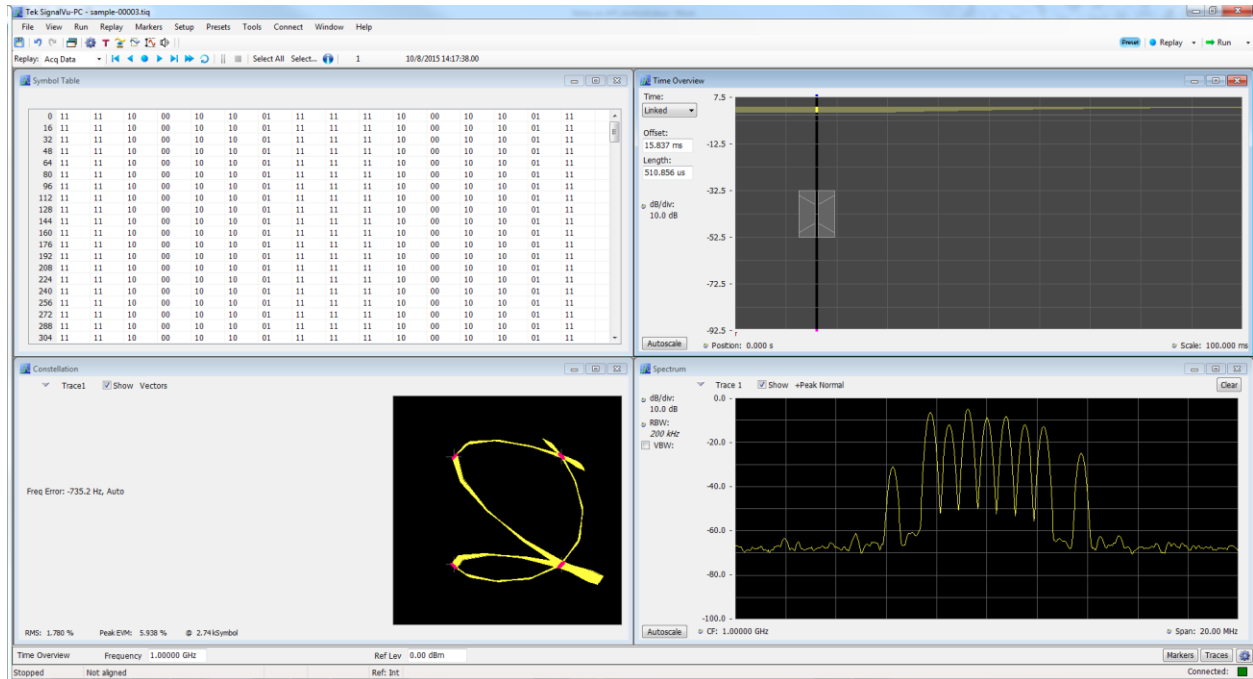
```
print('# of loops: {}'.format(loopCount))
print('File saved at {}'.format(filenameBase.value))
print('Disconnecting.')
rsa300.Disconnect()
```

## How to Use the RSA306 API in Python

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.