**Introduction**

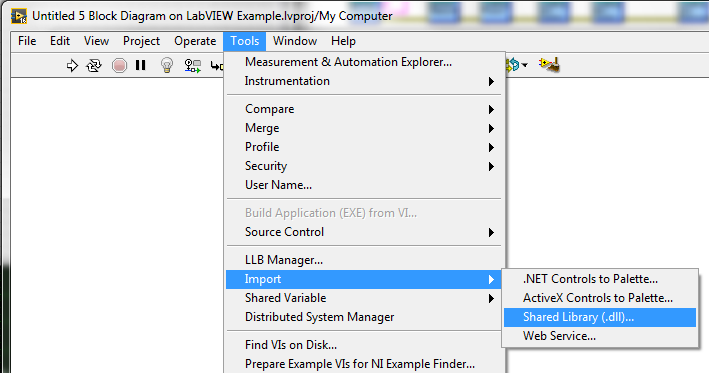
This document assumes you have a basic knowledge of programming (using variables, calling functions, knowledge of data types, how to wire functions, controls, and indicators together in LabVIEW, etc.) and have installed the RSA API: [RSA API for 64-Bit Windows - V3.13.0087](RSA%20API%20for%2064-Bit%20Windows%20-%20V3.13.0087). Don’t worry, the download includes both 32- and 64-bit versions.

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

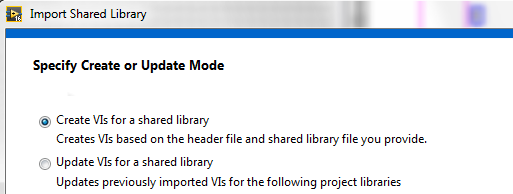
All of the magic in the API happens in RSA\_API.dll. A dll is a Dynamic Link Library, which contains precompiled functions that can be accessed by a script or program. This dll was written in/for C, so in order to import the dll and access its functions in LabVIEW, we have to use LabVIEW’s Import Shared Library function, which creates a separate VI for each function in the dll. LabVIEW is fairly intelligent when wrapping C functions from a dll, but there *are* some changes that need to be made to certain function parameters. Don’t worry, I’ve done all the frustrating research and trial & error so you don’t have to (you’re welcome)!

**Importing RSA\_API.dll**

Let’s get started. Go to Tools>Import>Shared Library (.dll)…, which will open an exciting dialog box.



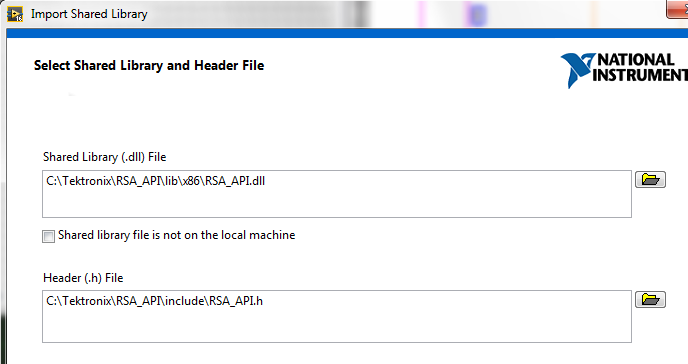
Select “Create Vis for a shared library” and hit Next.



Browse to or enter the following paths to the .dll and .h files and click Next. Note that these are x86, not x64. LabVIEW is a 32 bit program and it would not be happy with a 64 bit dll. Here are the paths if you’d like to copy and paste them.

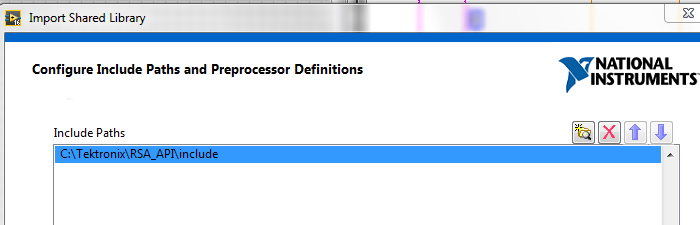
C:\Tektronix\RSA\_API\lib\x86\RSA\_API.dll

C:\Tektronix\RSA\_API\include\RSA\_API.h

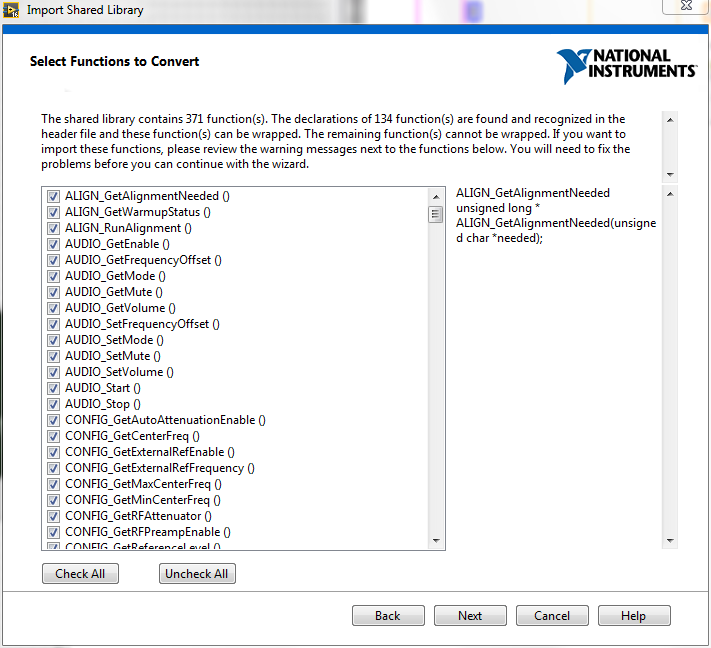


Similar to the last step, enter the include path below.

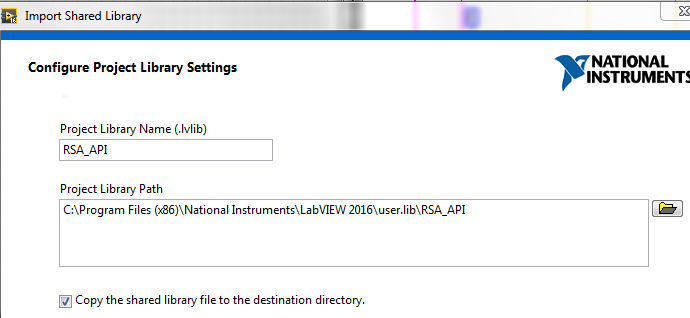
C:\Tektronix\RSA\_API\include



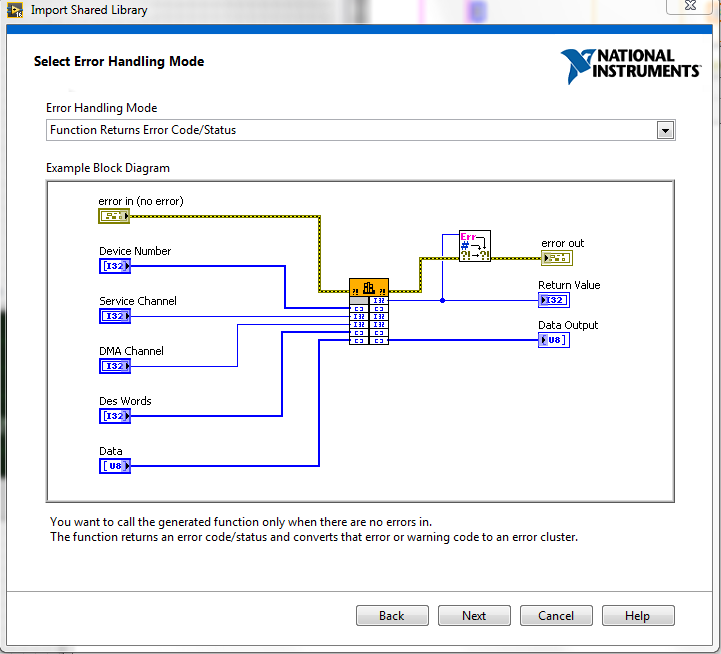
With any luck, you should see exactly the window shown below. Don’t worry about the unwrappable functions, they are from an older deprecated version of the API, so you only need the 134 functions that are wrappable by LabVIEW. Click Next.



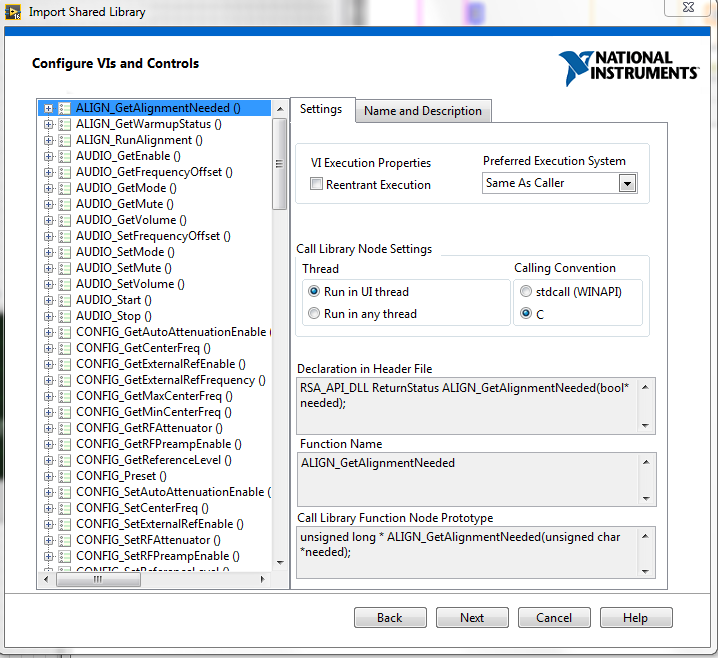
You can then create your own project library name and pick the save location. I just left the relatively intelligent name that LabVIEW automatically picks. You can also place a copy of the shared library file to the project directory. I’ll leave this up to you. I left it checked.



You can pick error handling mode. I used Function Returns Error Code/Status so I can at least tell if there’s been an execution failure between my API functions. The API functions have their own error message return values that can be used for debugging as well.



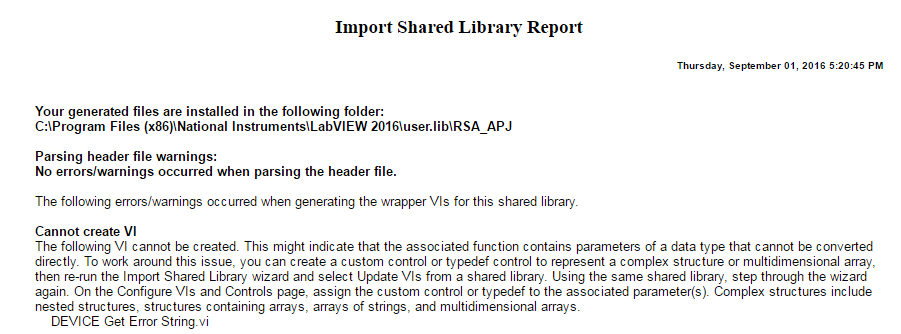
The next step in the wizard is to configure VIs and controls. Leave it alone and hit Next.



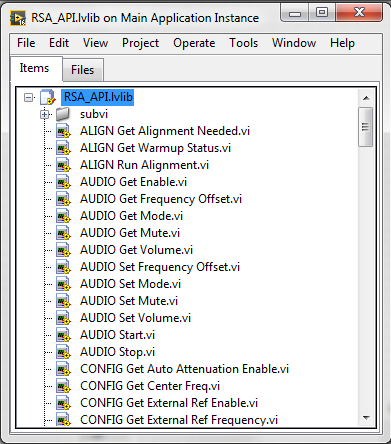
Next is a summary and the generation process. LabVIEW will go through and wrap every single function in the API into its own VI.



It’s very likely that you’ll see an error report next. There’s one function that LabVIEW doesn’t like because of its return data type: *DEVICE GetErrorString.* This is a task for another day.



After reading the error message, you’ll see this window pop up on your screen. It’s got a list of the VIs it created for the functions in RSA\_API.dll. You’ll be using this a lot, so don’t close it if you can help yourself.



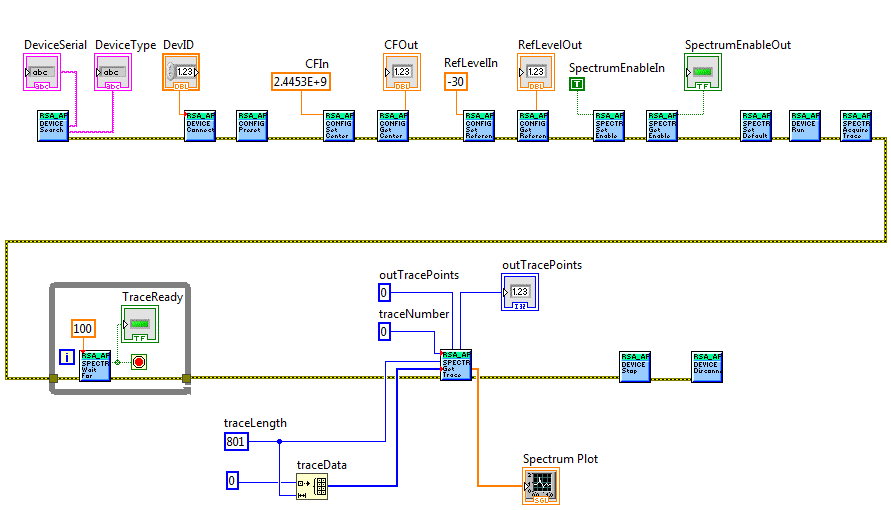
You’ll use the VIs LabVIEW just created to build your own VI. It will be a VIception.



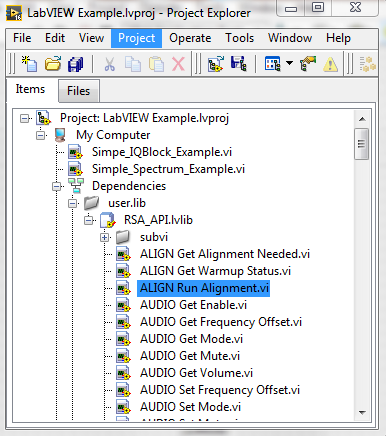
It will be very useful to have the official Tektronix documentation for the RSA API handy (<http://www.tek.com/spectrum-analyzer/rsa306-manual-6?aliId=260455870>). The VIs created by LabVIEW contain the function prototypes and have built-in variable inputs and outputs so it’s fairly clear what each function/VI requires to work correctly.

**The Program**

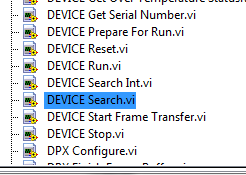
The example VI I’ve written connects to a USB RSA, sets the center frequency and reference level, enables the Spectrum consumer, acquires a spectrum trace, and plots it on a waveform graph. Here’s the block diagram for that VI.



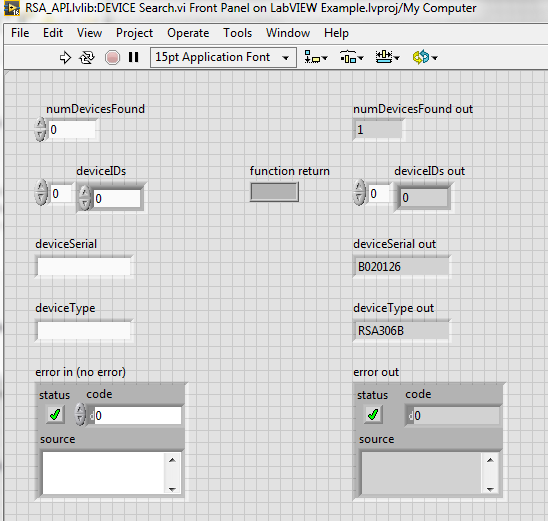
To access the API function VIs needed to build this VI, you can either use the window that popped up after LabVIEW imported all the functions or you can use the Project Explorer window.



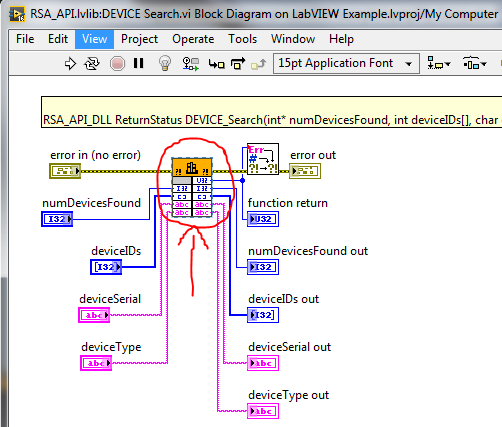
The first thing we need to do is search for any connected devices. We will use *DEVICE Search* to do this, and we will also use *DEVICE Search* as an example for general VI behavior. Click and drag it from the Project Explorer to your Block Diagram.

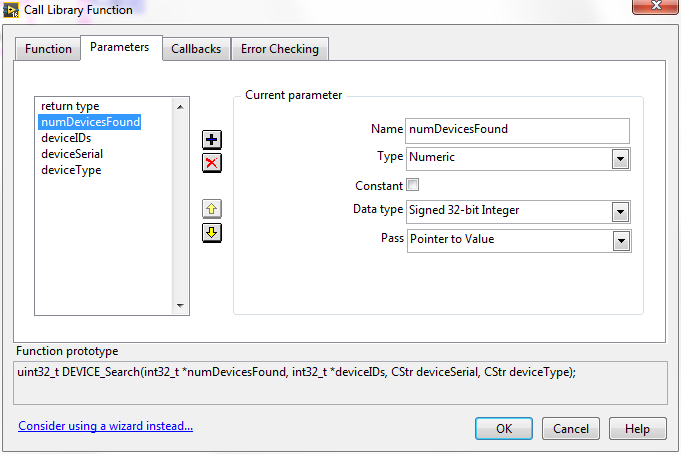
 

Technically we should give it some inputs, and this would be required in a different language, but LabVIEW automatically initializes the arguments for each function with either empty strings, NULL pointers, or the number 0 depending on the data type. That’s the case here and so all we need to do is provide the function outputs with the appropriate string indicators. Look at the function in more detail by double clicking the VI. It will bring up the front panel of that VI with all its built-in controls and indicators. You can even run the VI all by itself, the results of which are shown in the Front Panel screenshot below.

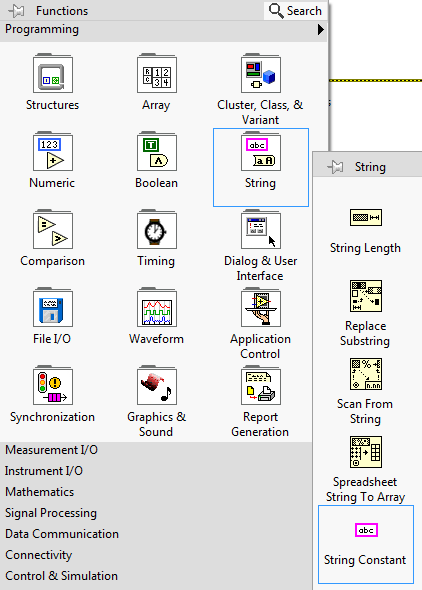


Bring up the Block Diagram for *Device Search*. It shows the inputs and outputs, their various data types, and the all-important *Call Library Function*, which is what LabVIEW uses as the wrapper for all the functions in RSA\_API.dll. If you double-click on *Call Library Function*, it brings up a dialog box with settings for this function. You won’t be doing anything special with *DEVICE Search*, but this process will be important for other functions later. Take a look at the Parameters tab in this dialog box, and you’ll see function parameters with information about them. The Type, Data type, and Pass dropdown menus will be important for other functions in the future. These menus indicate how LabVIEW has translated the argument variables from C data types to LabVIEW data types. The default configuration is just fine for most functions, but you’ll have to make some changes for a couple functions later.

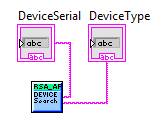
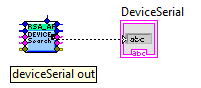




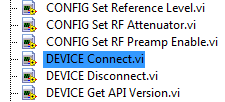
Now that we have way too much information about our function, we can create the appropriate controls & indicators and use it in our larger program. There are two outputs we care about right now, deviceSerial and deviceType. Both are strings, so we should create a string indicator for each. Right click on the Block Diagram and select String>String Constant.



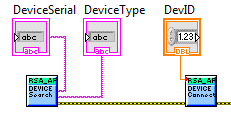
Right click on the string, select Change to Indicator, and name it either deviceSerial or deviceType. Make a copy of this string indicator and name it appropriately or just create a new one. You’ll be wiring these indicators to the appropriate outputs as shown below.



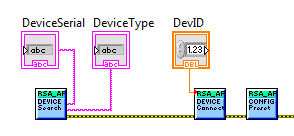
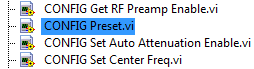
We’ll be using the Error as our control flow in this VI, so we’ll connect the ErrorOut of this function to the ErrorIn of the next function. Speaking of the next function, drag *DEVICE Connect* from the Project Explorer to the Block Diagram.

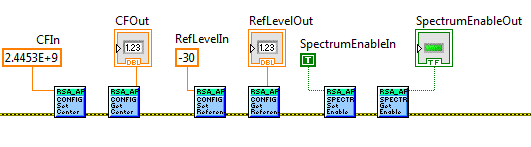
*DEVICE Connect* requires one input to select which instrument it connects to: deviceID. This is just an integer, and you can either create a constant or a control to manage which device it uses. Generally there will be just one RSA that is automatically enumerated as device 0 and you can plan accordingly. I’ve created the DevID control and wired it to the deviceID input of *DEVICE Connect*. I’ve also connected ErrorOut from *DEVICE Search* to the ErrorIn of *DEVICE Connect* to manage control flow.



Next we use the all-important *CONFIG Preset* function, which initializes the device to its default values (center frequency 1 GHz, reference level 0 dBm, IQ and spectrum bandwidth 40 MHz, free running, etc.) so we have a well-defined state from which to begin. It has no notable inputs or outputs so we don’t really need to do anything aside from making sure it executes.



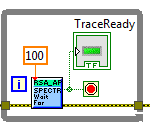
Next we will be setting and reading the center frequency & reference level of the instrument and enabling the spectrum functions, so we need *CONFIG Set Center Freq*, *CONFIG Get Center Freq*, *CONFIG Set Reference Level*, and *CONFIG Get Reference Level,*  *SPECTRUM Set Enable*, and *SPECTRUM Get Enable* from the Project Explorer. As you may have guessed, *CONFIG Set Center Freq* and *CONFIG Set Reference Level*, and *SPECTRUM Set Enable* each have one input: CFIn, RefLevelIn, and SpectrumEnableIn, respectively. *CONFIG Get Center Freq*, *CONFIG Get Reference Level*, and *SPECTRUM Get Enable* also similarly each have one output: CFOut, RefLevelOut, and SpectrumEnableOut, respectively. We’ll set the inputs as controls or constants and the outputs as indicators.

The next three functions do not have inputs or outputs (much like *CONFIG Preset*), so they are very simple. *SPECTRUM Set Default* sets the spectrum consumer to default settings (40 MHz span, 801 trace points, 300 kHz RBW, etc.), *DEVICE Run* tells the RSA to begin acquiring data (you should see the green light on the RSA begin to blink at this point in the VI),and *SPECTRUM Acquire Trace* tells the API to process a single spectrum trace.

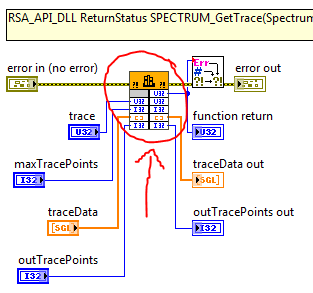


The next step is to have our VI wait until the data is ready to be queried. If you try to get the data before it’s finished processing, it will just return garbage data. There is a function that does this checking for us and it is called *SPECTRUM Wait For Trace Ready*. It has a **numeric** input (TimeoutMsec) and a **Boolean** output (TraceReady). We’ll put this function in a while loop that exits when TraceReady is true. I made TimeoutMsec a static numeric input of 100 ms. I also made TraceReady a Boolean indicator so we could see its status from the Front Panel.



Finally we need to get the trace and plot it. This is where we need to make some changes to the function and be a bit more thorough in our definition of input and output parameters. Let’s start by grabbing the *SPECTRUM Get Trace* VI, opening up its block diagram, and going to the *Call Library Function* settings Parameters tab.

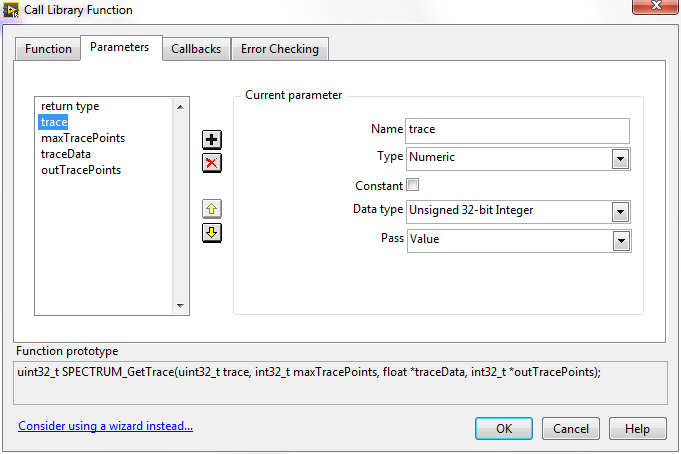
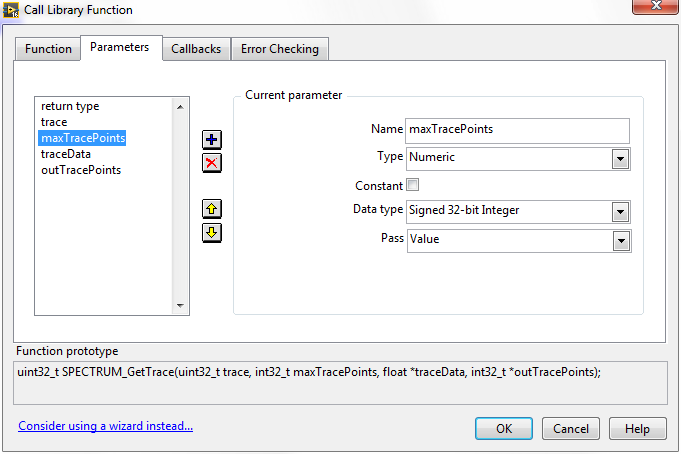
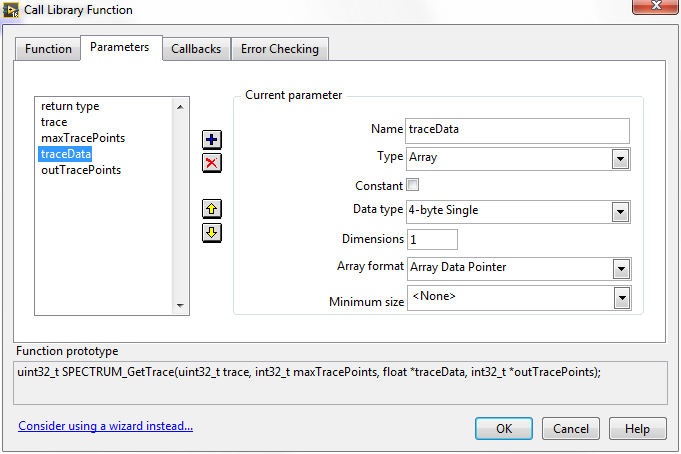


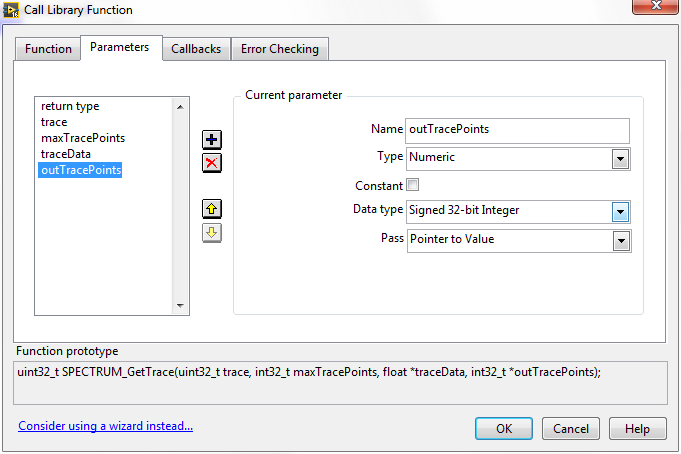
Let’s talk about the parameters before we change anything. The first parameter is trace, which specifies which trace is being acquired (the RSA can display up to 4 traces simultaneously). With the default settings, trace should be set to 0. You can leave it alone on the Parameters tab.

maxTracePoints specifies, you guessed it, the maximum number of trace points returned by the function. I set this to 801 because our default settings set the number of trace points to 801. Leave this alone in the Parameters tab as well.

traceData is the tricky one. traceData is an array of amplitude values. If you look at the function prototype, traceData is listed as a pointer to a float, which is the way C represents an array of floating point values. The default configuration for traceData in the *SPECTRUM Get Trace* VI is just a single numeric value, which is not going to give us an array. Keep in mind we have to translate the C data type to a data type that LabVIEW can understand and use. Since LabVIEW doesn’t understand pointers the same way C does, we need to change the Type from Numeric to Array. We can leave the data type as a 4-byte Single (aka float). It’s a single dimensional array, so Dimension is fine as well. We want the array format to be Array Data Pointer. Minimum size is up to you, but the smallest array that can be returned by *SPECTRUM Get Trace* is 801 points in length.

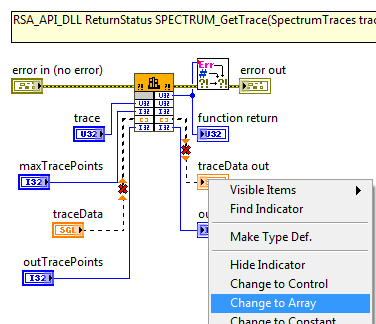
outTracePoints provides the number of trace points being returned by the function. Make sure it’s Pass parameter is Pointer to Value.

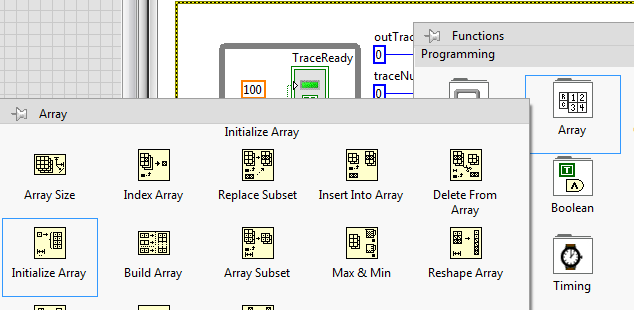


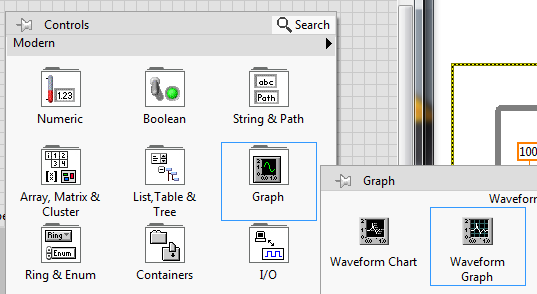
This nonsense needs to be done because of the concept of passing arguments to a function by value vs by reference. The way C passes values by reference is by using a pointer. When passing by value, the function can only use the variable’s value, but it can’t make any changes to the variable itself since it doesn’t have access to its location in memory. When passing by reference (using a pointer) the function can manipulate the variable itself because it actually has access to the variable’s memory location. We’re passing traceData and outTracePoints by reference (using pointers) in this function so that it can make changes to that memory and provide us with meaningful data when it’s finished.

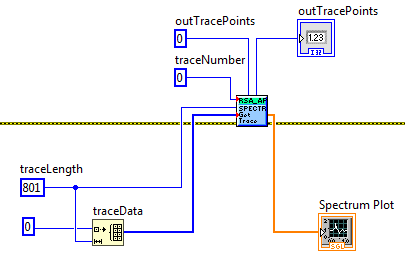
We also need to change the data types of the inputs and outputs on the *SPECTRUM Get Trace* Block Diagram after we make these changes. Right click on traceData and traceData out and select Change to Array.



Now that we’ve got that out of the way, we need to configure all the LabVIEW inputs and outputs. Let’s make an input control constant for traceNumber that selects the first trace (a value of 0). Wire this to the trace input. We also need to create a variable for outTracePoints. Its value doesn’t matter since it’s going to be changed by *SPECTRUM Get Trace*. I made it a constant and gave it a value of 0. Wire this to outTracePoints. I made a variable for the maxTracePoints input called traceLength and gave it a constant numeric value of 801. Wire this to maxTracePoints. And now for traceData. I made an array using LabVIEW’s *Initialize Array* function. This function takes in a size variable and an element variable. elementis the data type and value that all of the elements in the array will contain. I used the value from traceLength as my size input and just created a numeric constant with a value of 0 for my element input. Wire the output of this array to traceData. I also created an indicator for outTracePoints to make sure I have the right number of values returned by the function. Wire this to outTracePoints out. Finally we have traceData out, which we will wire directly to a Waveform Graph, which we will create on the Front Panel.

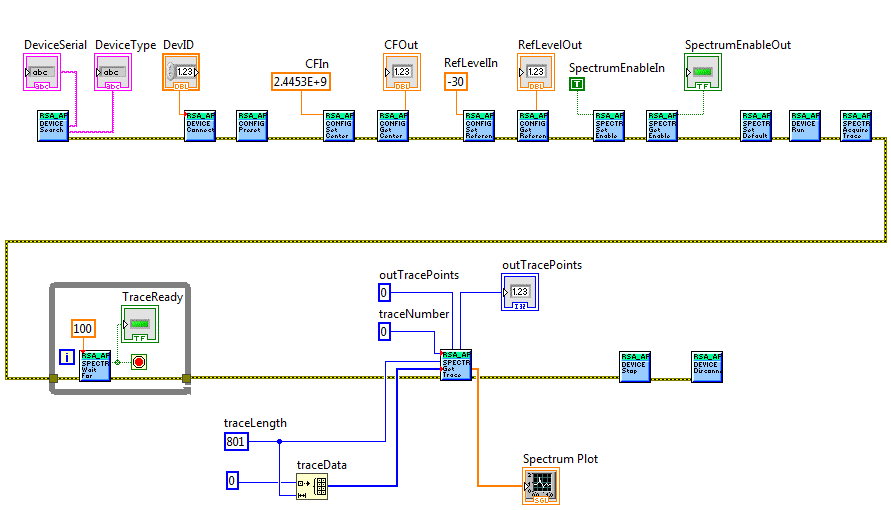


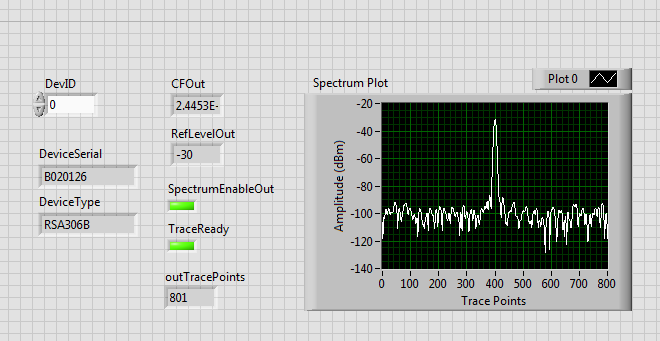




After we’ve gotten that straightened out, we can send the *DEVICE Stop* and *DEVICE Disconnect* commands. Neither of these have inputs or outputs, which should be a relief after setting up *SPECTRUM Get Trace*.

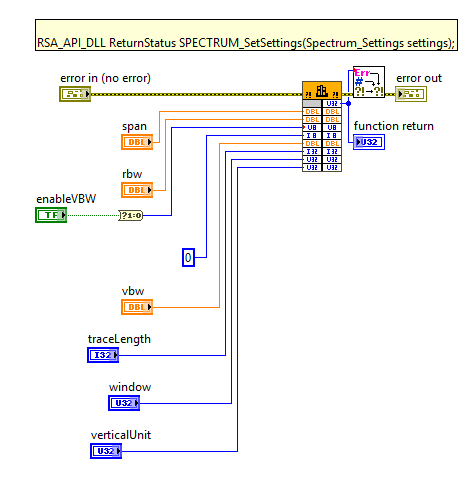
The final product should look like this:





***Spectrum\_setSettings()* configuration**

When using 32-bit LabView, the function *Spectrum\_setSettings()* requires to manually change the configuration like shown below. The default configuration doesn’t work and returns an error 1101 (errorParameterTraceLength). In 64-bit LV it can be used straight forward, no changes required.



Congratulations! You’ve successfully used the RSA API in LabVIEW and are now an expert. 