

Easily Integrating Sync Generator Data into Custom Management Dashboards

A Practical Tutorial on Working with the SPG9000 HTTP API

Introduction

In nearly every facility in the media and entertainment industry more than one device is used simultaneously. This means that for every facility, whether small or large, in a building or a vehicle, for transmission or production, timing is critical. In the past, it was sufficient to simply genlock all the analog and/or SDI gear within a single room, building, or campus. Today, the world is far more complex. We get references from atomic clocks in constellations of satellites. We connect facilities over wide-area networks. We have hundreds of devices instead of dozens. We must provide timing for IP-based systems that are nondeterministic and no longer connected point-to-point. That is why extremely accurate timing is ever more crucial. Interestingly, while we have more standards than ever, we also have more uniqueness in requirements and system designs. It is often necessary to bring together multiple disparate systems into a single management dashboard. Having an easy-to-use API in an industry-standard format facilitates doing so. That's why an easy-to-use application programming interface is of such great benefit to systems integrators and solution providers. It's what motivated Telestream to support the SPG9000 with an intuitive API that can be tailored to meet myriad requirements. This technical brief explains the concepts necessary to integrate the instrument into your specific system.

In what follows, we will describe several ways to use the HTTP API of the SPG9000 to monitor the performance a PTP follower instance. We will periodically capture the "offsetFromMaster" value computed from the PTP instance, save it to a local database or an in-memory data structure, and then graphically display the value over time. In one example we will use the tools that comprise the Elastic Stack software for data management. In the second example we will create a similar dashboard using Python 3 and some of the many available libraries that comprise it's data analysis ecosystem.

An Easy-to-Understand Application Programming Interface (API)

The new SPG9000 has an HTTP-based API for configuration and status monitoring. Because this is a common way of communicating with network devices, it is intuitive and will be easy to understand for any modern software developer. If a programmer has written code to GET (i.e., query, receive, and parse) information from, or PUT (e.g., set the value of a variable) it out to an HTTP endpoint, then the SPG9000 API calls will be familiar and stylistically as expected. The two examples in this technical brief will explain and demonstrate that. As we explore this API, several things will look very familiar. First, note that the paths to the data we need are standard HTTP endpoints. Second, the data structure is JSON. If you have ever written code to read and write JSON using HTTP, you will understand how to use this API immediately.



HTTP Endpoints

This paper assumes that you are familiar with HTTP endpoints and have interacted with them before. For those new to these concepts, here is an overview:

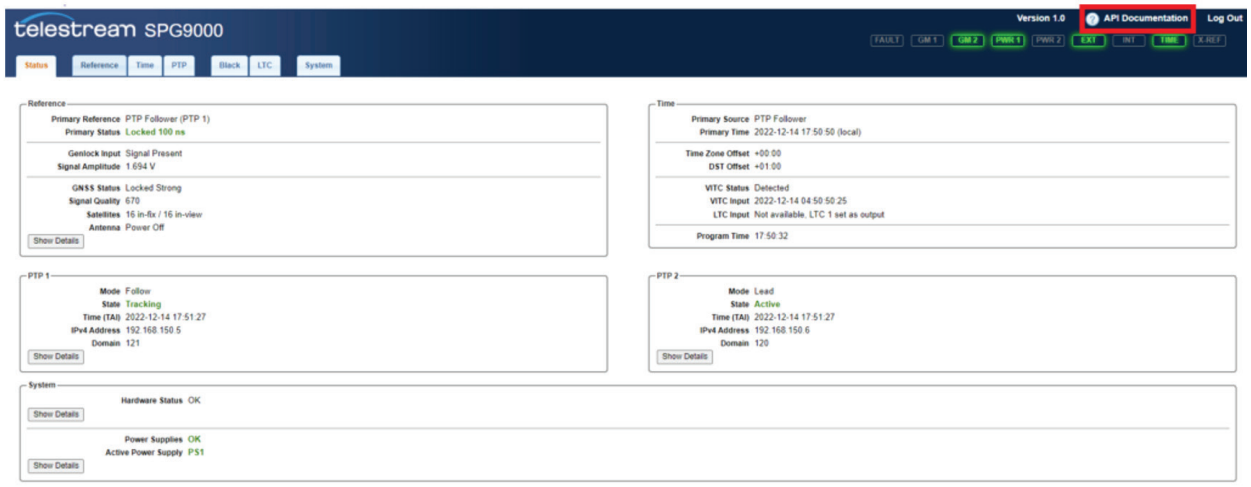
An API endpoint is a network location where an API can receive requests concerning some resource on the device being communicated with. Endpoints are generally URLs pointing to the location of the resource.

Applications that integrate with each other on a network are divided into clients, which send a request to another API and servers, which respond. In the SPG9000 examples, we will demonstrate a client application that runs on a computer, the server is on the SPG9000, and the client sets or retrieves a value from the server.

When the SPG9000 receives a request, the API performs some action and replies with a response back. In our examples, we will be asking an endpoint to iteratively return a value that is measured within the instrument.

What is JSON

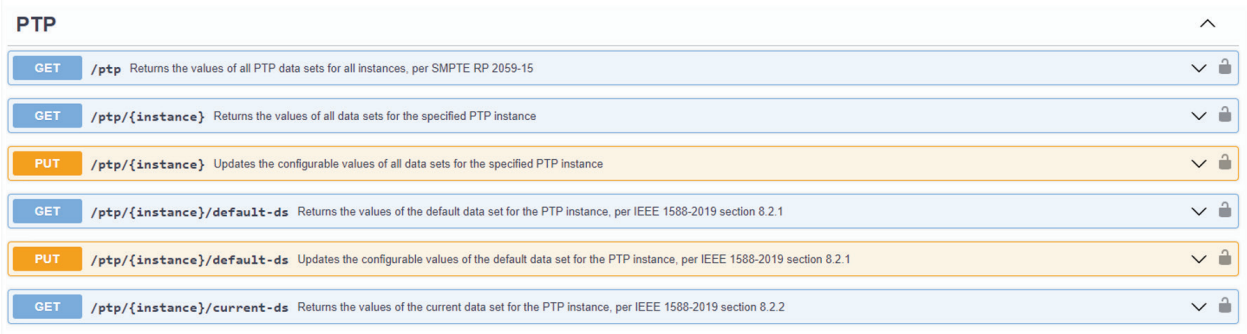
JSON is an acronym that stands for JavaScript Object Notation. It is a freely available standard for data storage and interchange. Although naming conventions are not always intuitive, JSON is generally human-readable. The fundamental concept is that of name–value pairs but these need not be single values. JSON is a public format with many uses and is the format in which the SPG9000 API interchanges data with client applications. As we hope you will see from the multiple examples in this paper, JSON is language-independent. It originated with JavaScript, but we use Elastic Stack and Python; our examples could just as well be implemented in C, C++, R, C#, Scala, or any other modern language.



Finding SPG9000 API Documentation

When you log into an SPG9000 you will see a user interface like earlier products but with several new features. In this document we are going to focus on the API functions and you will find a link to the API documentation in the upper right corner.

Clicking that link will take you to the API documentation where you will find such things as the possible PTP endpoints and the GET and PUT calls available. The documentation is organized by functional area and then by endpoint.



You can see the available data by clicking on a specific endpoint to dive one level deeper.



The example data in the image above describes the JSON that will be returned by the API call and offers a "sneak peek" into the example we will present below. Whenever you need to understand the behavior of an endpoint, the online documentation is the first place to visit. There you can also see examples of the structure of any given command and can execute it in a terminal application (provided you have an API Key) before you begin writing code.

The Data We Will Monitor

Consider a use case where you are building a dashboard and wish to plot a graph displaying how far your PTP timing deviates from the leader you are following. Because the SPG9000 is an accurate PTP timing device, evaluating the accuracy of a follower's offset from the PTP leader seems an appropriate monitoring example. Both of our examples will demonstrate this.

The measurement we need to make relies on the offsetFromMaster measurement which is defined in section 11.2 of IEEE Std 1588-2019. It represents the difference between a timestamp on a PTP follower and the corresponding timestamp on its PTP leader. Ideally this value would be 0, but realistically it will typically vary by several nanoseconds to microseconds due to the network architecture, dynamic network conditions, and other physical factors of the PTP implementation.

If we want to evaluate PTP timing accuracy, then monitoring this value over time is a useful metric for assessing network performance.

In the SPG9000, all PTP-related messages and data objects are based on the SMPTE RP 2059-15 YANG model, which is in turn based on the data sets defined in section 8 of IEEE Std 1588-2019. Specifically, section 8.2.2 defines the "currentDS" (current data set), which includes the "offsetFromMaster" value. In the SPG9000 API, sending a GET message to the /ptp/{instance}/current-ds endpoint will return a JSON object with the following structure (you saw this in the documentation screenshot above):

```
{
  "steps-removed": 1,
  "offset-from-master": 393216,
  "mean-delay": 501743616
}
```

The offset-from-master and mean-delay values use the TimeInterval type, defined in section 5.3.2 of IEEE Std 1588-2019. It is a time interval expressed in nanoseconds, multiplied by 216 (65536). Therefore, the offset-from-master value in the above example is actually 6 ns and the mean-delay is 7.656 µs.

Example 1: The "Minimal-Coding" Elastic Stack Implementation

To understand this example, please first familiarize yourself with the Elastic Stack topology, then install the software. After the Elastic Stack software (Logstash, Elasticsearch, Kibana) has been installed on the monitoring system, it must be configured to fetch the necessary data. Begin by editing the logstash configuration file (normally saved in the /etc/logstash/conf.d directory on a Linux system) as follows:

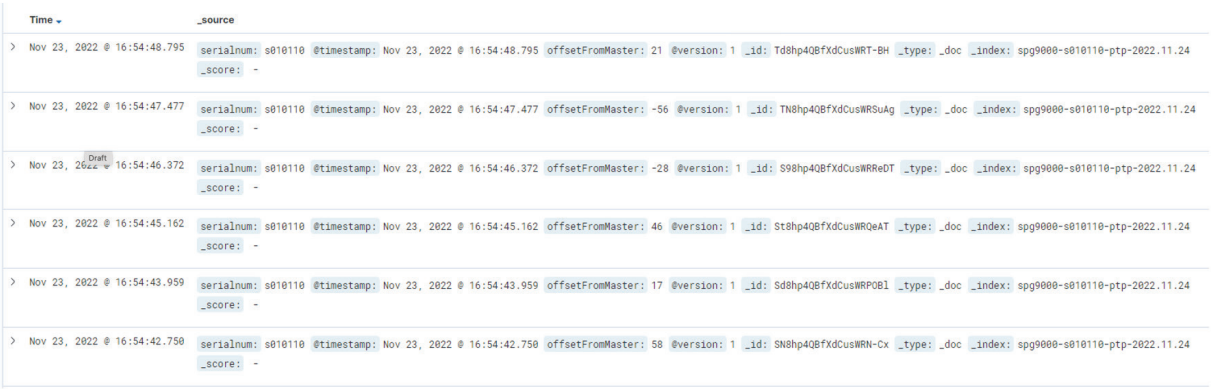
input {
  http_poller {
  urls => {
  "spg9000-s010110" => {
  method => get
  url => http://<YOUR SPG9000>/api/v1.0/ptp/1/current-ds
  headers => {
  Accept => "application/json"
  "X-API-Key" => "myApiKey"
  }
  }
  }
  schedule => { every => "1s" }
  codec => "json"
  }
}
	The input section uses the Http_poller plugin to query the SPG9000. The name chosen for the url parameter is the hostname of the SPG9000. The url value is the complete path for the API endpoint, including the "/api/v1.0" prefix. The "/ptp/1/current-ds" message will return the data for the PTP 1 instance on the SPG9000; use /ptp/2/current-ds for the PTP 2 instance. Next, the API requires a security key, sent via the X-API-Key header in the HTTP message. Lastly, the http poller plugin is configured to send this message every second and to expect a JSON response.

filter {
  json { source => "message" }
 	
  mutate {
  remove_field => ["steps-removed", "mean-delay"]
  add_field => { "serialnum" => "s010110" }
  rename => { "offset-from-master" => "offsetFromMaster" }
  }
 	
  ruby {
  code => "event.set('offsetFromMaster', event.get('offsetFromMaster') / 65536.0)"
  }
}
	The filter section is used by logstash to modify the raw response so that it conforms to our expected use. The mutate filter first removes the steps-removed and mean-delay values since we aren't using them in this example. This will reduce the size of each database record. Next we add a "serialnum" field to identify this particular SPG9000. This is useful if we want to monitor multiple machines, as we will need to identify the source for each data record. Lastly, we rename the value from offset-from-master (as it is named in the YANG model and the API) to offsetFromMaster (as it is named in 1588-2019) because the hyphen character must be removed for the next step.
	
	There is an available Ruby plugin that we will use to transform the value from the TimeInterval format to a plain value of nanoseconds, by dividing the value by 65536.

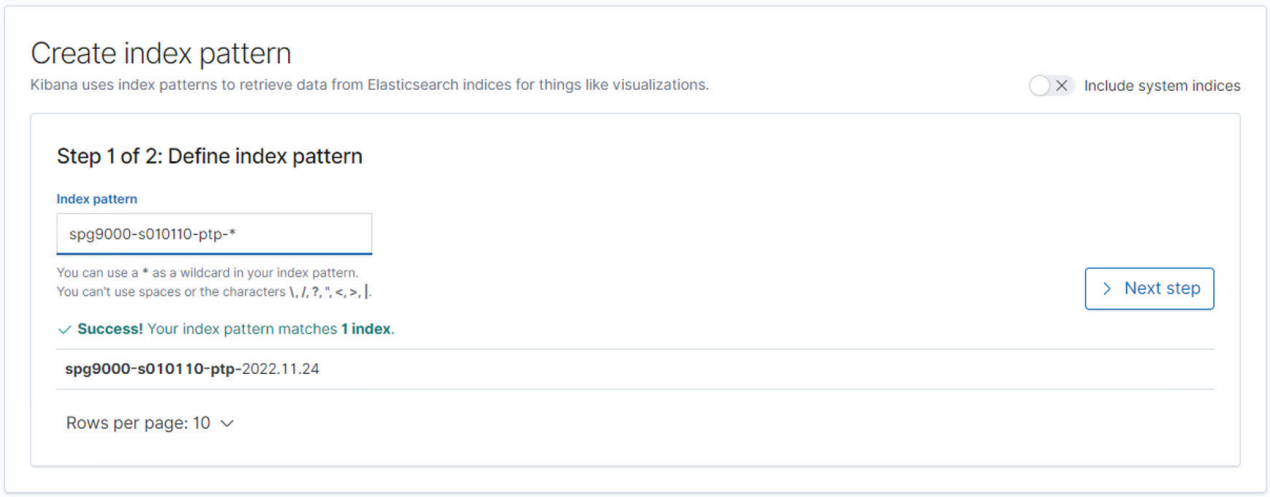
output {
  elasticsearch {
  hosts => ["http://10.101.3.202:9200"]
  index => "spg9000-%{serialnum}-ptp-%{+YYYY.MM.dd}"
  }
}
	The output section simply sends the transformed data record to Elasticsearch. The index name chosen here includes the serial number (added in the filter section above) and the current date, to make it easier to search for this data later.

1. Note that the IEEE-1588 offsetFromMaster is simply a difference between timestamps and does not necessarily account for network asymmetry.
2. offsetFromMaster does not assess the timebase of the instrument relative to its PTP subsystem. So, for example, when recovering from a holdover, it may temporarily be mistimed even if the PTP subsystem is well aligned.

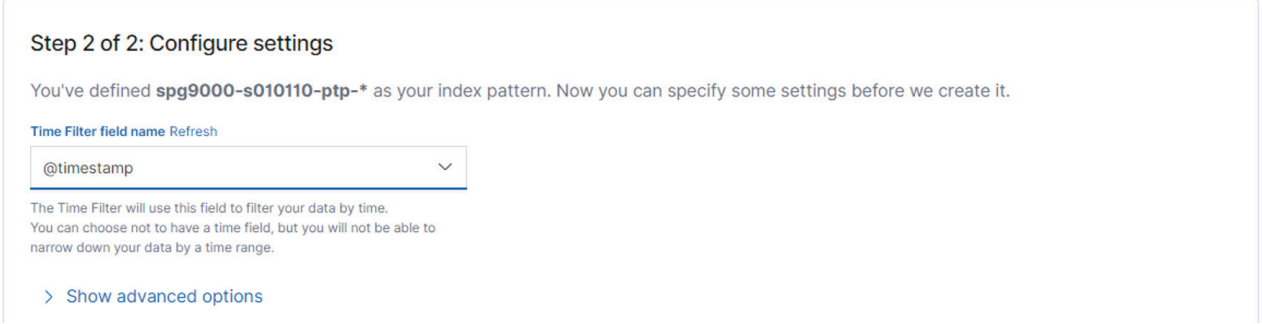
When Logstash is started with this configuration file, new indexes and data records will be added to the database. You can see these with the Discover page on the Kibana page. You will see the records with the modified offsetFromMaster field and the serialnum field that was also added:



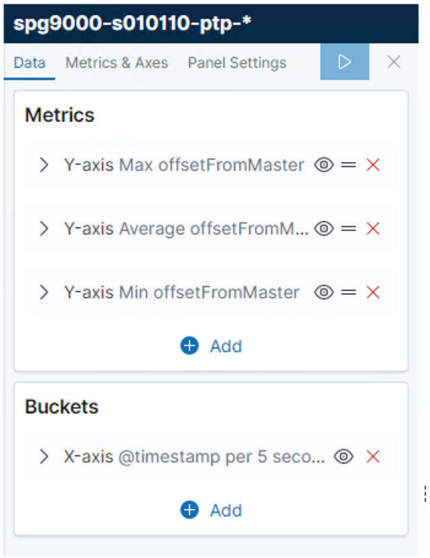
To visualize this data, we use the Kibana software in Elastic Stack. First, we will need to create a new index pattern. In this case, we use the prefix chosen in the logstash configuration above, with "*" to match all dates.



In the process of creating the index pattern, we next set the Time Filter to use the automatically-generated @timestamp value.



Once the index pattern has been created, we can create a new Visualization from the Kibana page. For this example, we will add three Y-axis metrics using the Max, Average, and Min aggregation types for the offsetFromMaster field. We will also use the Date Histogram aggregation for the X-axis, using the @timestamp field.



Our result will be a dynamic, continually updating graph like the following. This can then be added to a Kibana Dashboard alongside any other visualizations you care to create.



What you have seen in this example is a method for obtaining the offset-from-master value from the PTP current-ds data set with little need to write code. Clearly, we could use a similar approach to monitor any other data from the SPG9000, such as GNSS signal quality, SPG reference status, or system diagnostics. In other words, with a minimal need to write software, it is possible to create an arbitrarily complex dashboard for integrating the SPG9000 into custom solutions.

Example 2: A Simple Python Program

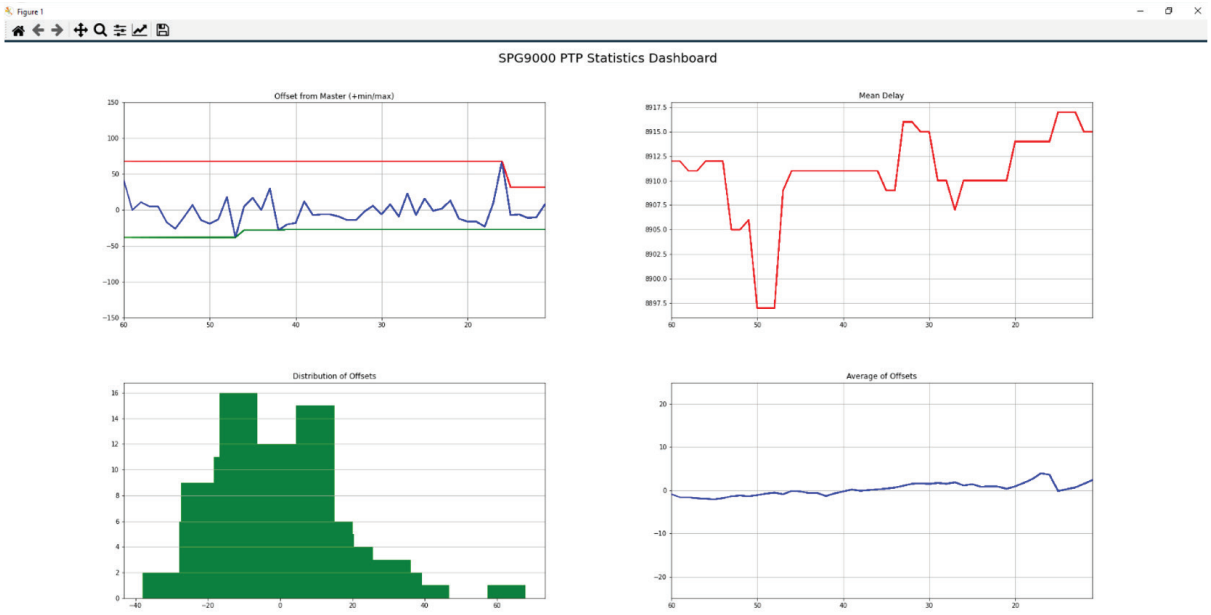
In our second example, we will use the Python "requests" and "json" libraries to extract the JSON object from the endpoint used above. Then we'll plot the graphs on our dashboard using matplotlib.pyplot. If you have done any data analysis in Python, you have likely done this before.

%matplotlib qt5 import matplotlib.pyplot as plt import matplotlib.animation as animation import requests import json	We will begin by allowing matplotlib to run interactively and then importing the Python libraries we need. In this example we will be creating graphs using the matplotlib.pyplot library and allowing them to update dynamically with the animation library. The requests library permits us to query an endpoint and the JSON library to parse the result.
offsetFromMaster = [] averageOffset = [] minOffset = [] maxOffset = [] meanDelay = [] counter = [] i=25	Next we will define some lists to hold the ever-growing data that we will collect and graph. If you are a Python programmer, you'll realize that this could be accomplished with a multi-dimensional array, dictionary, or Pandas Dataframe. Here we are using individual lists so it's not necessary to understand more complex data structures.
currentds_URL= "http://<YOUR SPG9000>/api/v1.0/ptp/1/current-ds"	Next, set the current dataset as the endpoint to query. This is similar to what we did in the previous example but with Python syntax.
fig, axs = plt.subplots(2, 2, figsize=(9, 8), linewidth=5, edgecolor="#04253a") fig.suptitle('SPG9000 PTP Statistics Dashboard', fontsize=20) #Grid Style axs[0,0].grid() axs[0,1].grid() axs[1,0].grid(axis='y') axs[1,1].grid() #Subplot Titles axs[0,0].set_title('Offset from Master (+min/max)') axs[0,1].set_title('Mean Delay') axs[1,0].set_title('Distribution of Offsets') axs[1,1].set_title('Average of Offsets') #Cleanup spacing plt.subplots_adjust(left=0.1, bottom=0.1, right=0.9, top=0.9, wspace=0.3, hspace=0.3)	Setup a figure with 4 subplots and give them titles and gridlines, axis limits and styles. For more information on syntax and the available options (of which there are many) consult the matplotlib.pyplot documentation at: https://matplotlib.org/3.5.3/api/_as_gen/matplotlib.pyplot.html

def graphPTPStats(i):
 	
    # Increment the counter
    counter.append(i)
 	
    # Get the PTP data from the endpoint
    currentds_result = requests.get(currentds_URL, headers={“X-API-Key”:”<YOUR API KEY>”}).json()
 	
    # Add the PTP data the the end of the lists
    meanDelay.append(currentds_result[‘mean-delay’])
    offsetFromMaster.append(currentds_result[‘offset-from-master’]/65536)
 	
    # Compute some basic statistics from the contents of the lists
    averageOffset.append(np.average(offsetFromMaster))
    minOffset.append(np.min(offsetFromMaster))
    maxOffset.append(np.max(offsetFromMaster))
 	
    # Plot the offset with Min and Max lines
    axs[0,0].plot(counter, offsetFromMaster, ‘b’)
    axs[0,0].plot(counter, minOffset, ‘g’)
    axs[0,0].plot(counter, maxOffset, ‘r’)
    axs[0,0].set_xlim(counter[i],i-50)
 	
    # Plot the delay
    axs[0,1].plot(counter, meanDelay, ‘r’)
    axs[0,1].set_xlim(counter[i],i-50)
 	
    # Plot the distribution and average of the offsets but wait for a couple iterations
    if(i>=2):
        axs[1,0].hist(offsetFromMaster, bins = 10, color = “green”)
        # Plot the average offset
        axs[1,1].plot(counter, averageOffset, ‘b’)
        axs[1,1].set_xlim(counter[i],i-50)
	Finally, to create a graph that updates continuously we will define a function that gets data from the endpoint, adds it to the lists, and plots it. Unlike our earlier example we will graph both the “offsetFromMaster” and the “meanDelay”. Then, just for the sake of diversity, we’ll create a histogram of the offsets. Note that, as in example 1, we divide offsetFromMaster by 65536.

ani = animation.FuncAnimation(fig, graphPTPStats, interval=500) plt.show()	Finally, we will use the animation library to continually call the plotting function and will create a graph that updates every 500 milliseconds

If you copy everything from the left column of the previous page's table into a Python IDE, a Jupiter Notebook, or a Python 3 REPL, and you include the actual hostname or IP address of your instrument, the result will be a dashboard which updates twice per second, that looks something like this:



Sophistication Can Be Simple

We hope these examples have made it clear that the new SPG9000 API is intuitive enough to empower anyone with a bit of modern coding knowledge to use and deploy it. This is just one reason, among many, that the instrument is easy to integrate into a sophisticated system. Of course, you can meet all your timing and synchronization needs without ever writing software to the API. This is simply one more way that the SPG9000 exceeds expectations.

Learn More

To learn more about the SPG9000, click here.

www.telestream.net | info@telestream.net | tel +1 530 470 1300

Specifications subject to change without notice. Copyright © 2023 Telestream, LLC. Telestream, CaptionMaker, Episode, Flip4Mac, FlipFactory, Flip Player, Lightspeed, ScreenFlow, Switch, Vantage, Wirecast, GraphicsFactory, MetaFlip, and Split-and-Stitch are registered trademarks and PRISM, MacCaption, e-Captioning, Tempo, VOD Producer, and Vidchecker are trademarks of Telestream, LLC. TitlerPro is a trademark of NewBlue, Inc. All other trademarks are the property of their respective owners.
February 2023