HtmlReportWriter Tutorial

This tutorial will show you how to use the "HtmlReportWriter" component in Introspect IESP. The HtmlReportWriter component provides an easy way to create HTML-format reports (with graphics) to show your Test results.

The Basics

Let's start with a simple example to show the basic operation of the HtmlReportWriter component. Create a new Test in Introspect ESP and then remove the "globalClockConfig" component if one has automatically been created. (We will not make use of the IESP hardware at all in the first part of this tutorial – this allows you to proceed without access to the hardware.) Save this Test as "HtmlReportWriterTutorial".

Then use the "Add" button to add an instance of the "HtmlReportWriter" component to the Test (it will be called 'htmlReportWriter1') and then rename it to 'reportWriter1' (just to make it have a shorter name). Note that there is no 'run' method for this component. The HtmlReportWriter component class provides various methods for programmatic use. It doesn't do anything by itself. That is, you will need to write some simple Python code to generate the HTML report.

To provide a place to put the Python code and to get a Result folder where the report file will be generated, we will use a "FunctionWithResults" component. This component has a 'run' method that will create a new folder for holding result files and then will execute the Python code you supply. Use the "Add" button to add an instance of the "FunctionWithResults" component to the Test (it will be called 'functionWithResults1') and then rename it to 'createReport'. Your Test should now look like Figure 1.

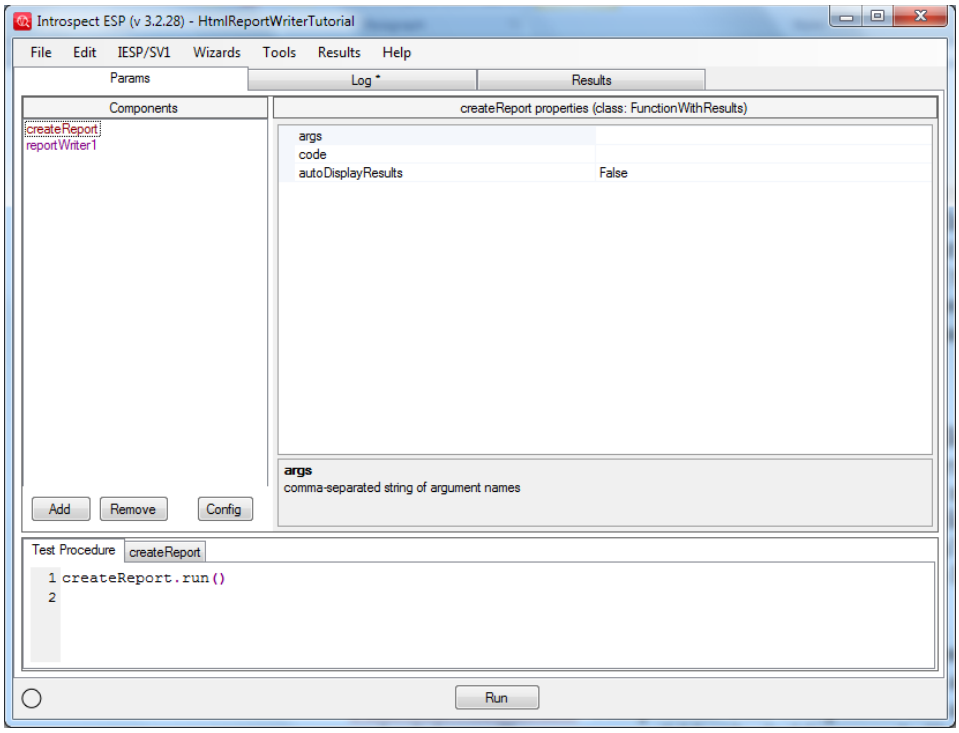


Figure 1 Test showing a FunctionWithResults component called createReport.

Notice that there is a call to the 'run' method of the 'createReport' component in the Test Procedure. Now click on the tab for 'createReport' and enter the following lines into the 'code' area there:

```
folderPath =
self.createRunResultFolder("StatusReport",
                                                "HtmlReport")
reportWriter1.start(folderPath)
reportWriter1.addHeading(1, "Status Report")
reportWriter1.addParagraph("This is a status report
on my progress.")
reportWriter1.createFile()
```

The first line of the above code uses the 'self' variable – this refers to the 'createReport' component (where we are entering the code). That line calls the 'createRunResultFolder' method with two strings as arguments. The first string ("StatusReport") is arbitrary – it specifies the name of the folder to be created. The second string ("HtmlReport") specifies the desired type of run result folder. The type of a run result folder determines its icon and what happens when you double-click that icon in the Results area of Introspect ESP. Here we specify the type as "HtmlReport" so that double-clicking the icon will open any HTML files in that folder. The 'createRunResultFolder' method returns the full path to the

folder that it creates. See the documentation for the "FunctionWithResults" class in "svt.html" for more details.

The last four lines of code above invoke methods on the 'reportWriter1' component. The first of these lines calls the 'start' method to tell the 'reportWriter1' component which folder should be used to store the generated HTML file and support files. We pass it the path to the folder that will get created by the call to 'createRunResultFolder'. It is a requirement that you call the 'start' method before calling other methods on 'reportWriter1'.

The 'addHeading' method tells 'reportWriter1' to add the specified string as a heading (of level 1) in the HTML, and the 'addParagraph' method tells it to add the specified string as a paragraph in the HTML. Finally, the 'createFile' method tells 'reportWriter1' to create the HTML file (in the folder you specified in the call to 'start'). The name of the HTML file will be "report.html" since that is the default value of the property 'fileName' of 'reportWriter1' and we didn't change it.

Your Test should now look like Figure 2.

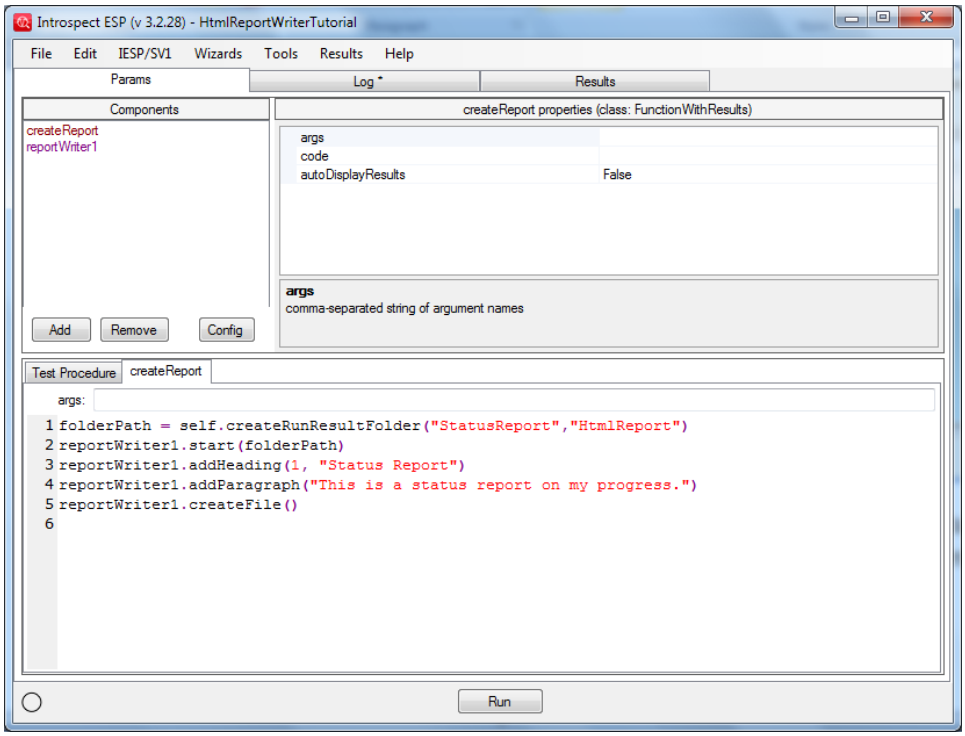


Figure 2 Functional code added in order to declare an html report output.

Now click the “Run” button to run this Test and you should get a Result icon labeled “StatusReport” (the name of the folder that we specified in the call to ‘createRunResultFolder’). Double-clicking on this icon will open a web page viewer and display the generated HTML. It should look like Figure 3.

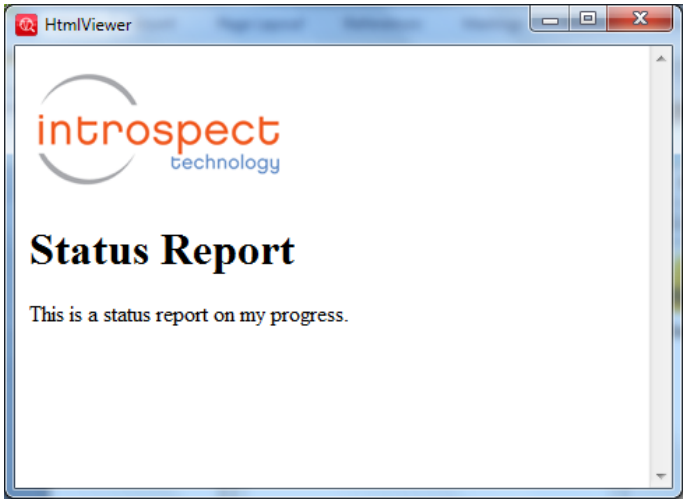


Figure 3 Html result viewer showing the generated report.

You can of course, add multiple headings (of various levels) and multiple paragraphs of text to your HTML report by just calling the ‘addHeading’ and ‘addParagraph’ methods repeatedly. There is also a general ‘addHtml’ method which can be used to add an arbitrary string of HTML code to your report. (You would likely only use this for HTML features which are not supported by the specialized methods of the HtmlReportWriter class.)

Adding Tables of Data

Now let’s look at how to add tables of data to your HTML reports. As a starting example, we’ll add a table that shows the number of days in each month of the year. This simple example allows us to concentrate on the HtmlReportWriter methods for adding table elements without getting distracted by where the data is coming from.

In the 'code' area of 'createReport', add the following lines just before the call to 'createFile':

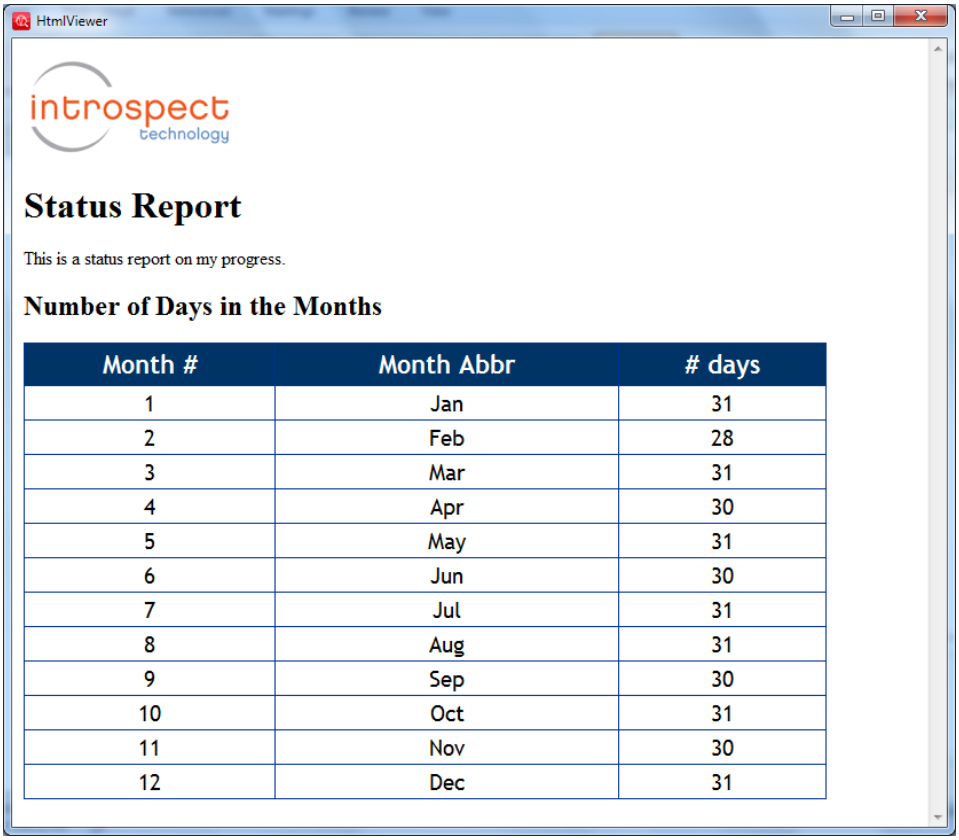
```
reportWriter1.addHeading(2, "Number of Days in the Months")
monthInfo = [('Jan',31), ('Feb',28), ('Mar',31), ('Apr',30),
             ('May',31), ('Jun',30), ('Jul',31), ('Aug',31),
             ('Sep',30), ('Oct',31), ('Nov',30), ('Dec',31)]
reportWriter1.addTableStart()
reportWriter1.addTableRowStart()
reportWriter1.addTableHeaderCell("Month #")
reportWriter1.addTableHeaderCell("Month Abbr")
reportWriter1.addTableHeaderCell("# days")
reportWriter1.addTableRowEnd()
for i in range(12):
      monthNum = i + 1
      (monthAbbr, numDays) = monthInfo[i]
      reportWriter1.addTableRowStart()
      reportWriter1.addTableCell(monthNum)
      reportWriter1.addTableCell(monthAbbr)
      reportWriter1.addTableCell(numDays)
      reportWriter1.addTableRowEnd()
reportWriter1.addTableEnd()
```

The above lines of code start an HTML table ('addTableStart'), then add a header row with table cells "Month #", "Month Abbr" and "# days", then loops through the months, adding a table row for each month with the month number, the month abbreviation and # of days from 'monthInfo'. Finally, the HTML table is ended with 'addTableEnd').

If you Run the Test now, the generated HTML report should look like Figure 4.

That was a lot of lines of code just to create one table because we spelled out how to create each row with a separate method call. That is the most general and most flexible way to do it, but for common cases there are much more concise ways. For example, if we reformulated our 'monthInfo' data as a Python dictionary (with the keys being the month numbers) then we could use the method 'addTableFromDictOfListValues'. To try this, replace the above lines that you added for creating the table with the following lines:

```
reportWriter1.addHeading(2, "Number of Days in the Months")
monthInfoDict = {1:('Jan',31), 2:('Feb',28), 3:('Mar',31),
4:('Apr',30),
             5:('May',31), 6:('Jun',30), 7:('Jul',31), 8:('Aug',31),
             9:('Sep',30), 10:('Oct',31), 11:('Nov',30), 12:('Dec',31)}
labels = ["Month #", "Month Abbr", "# days"]
                        reportWriter1.addTableFromDictOfListValues(mont
                        hInfoDict, labels)
```



Month #	Month Abbr	# days
1	Jan	31
2	Feb	28
3	Mar	31
4	Apr	30
5	May	31
6	Jun	30
7	Jul	31
8	Aug	31
9	Sep	30
10	Oct	31
11	Nov	30
12	Dec	31

Figure 4 Html report with a table declaration.

This is a lot more concise, but if you now run the Test, you should get the same HTML report as before. The documentation for the "HtmlReportWriter" class in "svt.html" gives more details about this and other 'addTableFrom…' methods.

Adding Tables of Data from Measurement Components

Now let's look at a more realistic example. To set this example, please add the globalClockConfig back into the Test Procedure since we will be performing real tests with the hardware. Also, please add a txChannelList1 component with as many channels as you have physically connected to the hardware. Finally, assuming that there are receiver channels connected, run the bertScan wizard and create a bertScan1 component. The resulting Test window should look like Figure 5, where we have deleted the bertScan1.run() command from the main Test Procedure. This will be added later inside the createReport function.

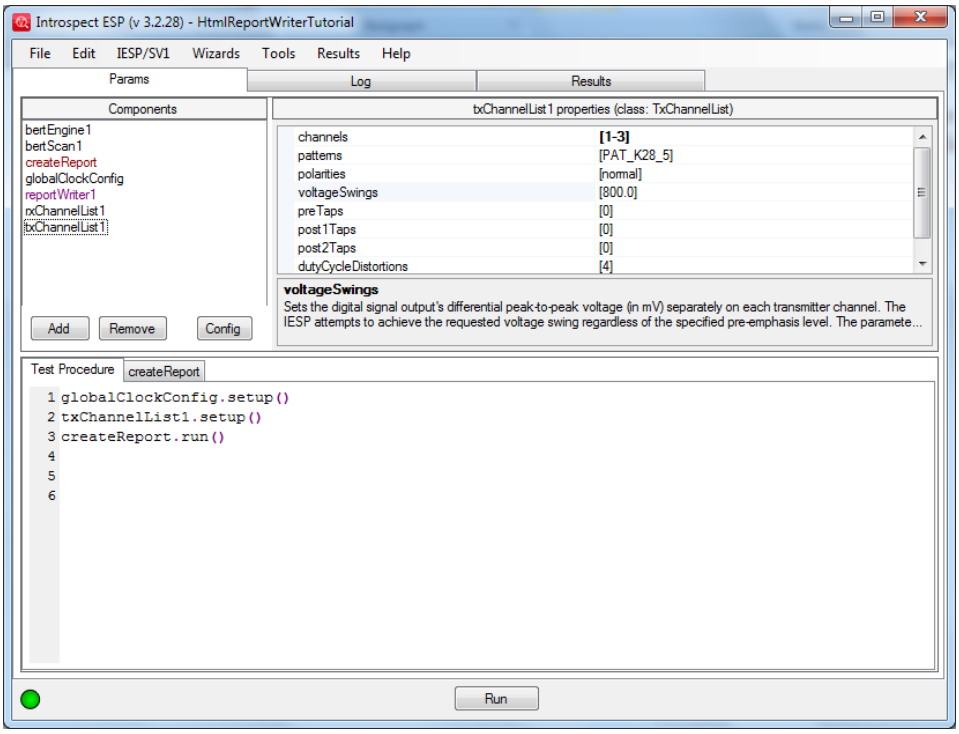


Figure 5 More complete Test containing actual IESP components.

We would like to add the bertScan1.run command into the createReport function. The 'run' method of the BertScan component returns a tuple of values that can be used for further analysis or report generation. (See documentation for the BertScan class in "svt.html".) Usually the Test Procedure doesn't capture the return value from the 'run' method since it is usually sufficient to just look at the results using the BertScan viewer. But if you want to do a report, change the line in the Test Procedure to something like the following:

```
(calibDelays, errCounts, jitterAnalysisResults) = bertScan1.run()
```

In our case, we would like to place the above command in the createReport function and not in the main Test Procedure, since the createReport function is the one that makes use of the bertScan1 return values.

Then, for the purpose of this tutorial, we will concentrate on the jitter analysis results out of bertScan1. We get a dictionary 'jitterStats' from the 'jitterAnalysisResults' as follows:

```
jitterStats = jitterAnalysisResults['jitterStats']
```

The 'jitterStats' dictionary has entries for each channel that give: 'rj', 'dj', 'tj', 'berLevel', 'eyeCenter' and we can make a table with these values with the following code:

```
labelsDict = dict(rj = "Delta-Delta RJ (ps)",
                  dj = "Delta-Delta DJ (ps)",
                  tj = "Delta-Delta TJ (ps)",
                  berLevel = "Extrapolated BER at Eye Center",
                  eyeCenter = "Measured Eye Center (ps)")
reportWriter1.addTableFromDictOfDictValues(jitterStats,
                                           "Channel",labelsDict)
```

The 'labelsDict' variable is a dictionary that specifies what the labels should be for each of the 'rj', dj', etc entries. The second argument to 'addTableFromDictOfDictValues' is the label to be used for the first column of the table (that shows the keys of the 'jitterStats' dictionary (here the keys are channel numbers).

The resulting createResults function is shown below in Figure 6. As can be seen, we kept the original heading and paragraph text; and we simply added the code to execute the bertScan, extract its jitter statistics, and then map the jitter statistics into tabular format.

Test Procedure | createReport

args:

```
1  folderPath = self.createRunResultFolder("StatusReport","HtmlReport")
2  reportWriter1.start(folderPath)
3  reportWriter1.addHeading(1, "Status Report")
4  reportWriter1.addParagraph("This is a status report on my progress.")
5
6  (calibDelays, errCounts, jitterAnalysisResults) = bertScan1.run()
7  jitterStats = jitterAnalysisResults['jitterStats']
8  labelsDict = dict(rj = "Delta-Delta RJ (ps)",
9                    dj = "Delta-Delta DJ (ps)",
10                   tj = "Delta-Delta TJ (ps)",
11                   berLevel = "Extrapolated BER at Eye Center",
12                   eyeCenter = "Measured Eye Center (ps)")
13 reportWriter1.addTableFromDictOfDictValues(jitterStats,
14                                           "Channel",
15                                           labelsDict)
16
17 reportWriter1.createFile()
```

Figure 6 Illustration of code within the createReport function.

The table generated via the above code would look like Figure 7. One issue that is apparent in the above table is that the numbers are displayed with far too many decimal places. To fix this, you can use the 'setNumSignifFigures' method. For example, adding a line like the following before the call that generates the table would make all table entries show only 3 significant digits:

```
reportWriter1.setNumSignificantFigures(3)
```

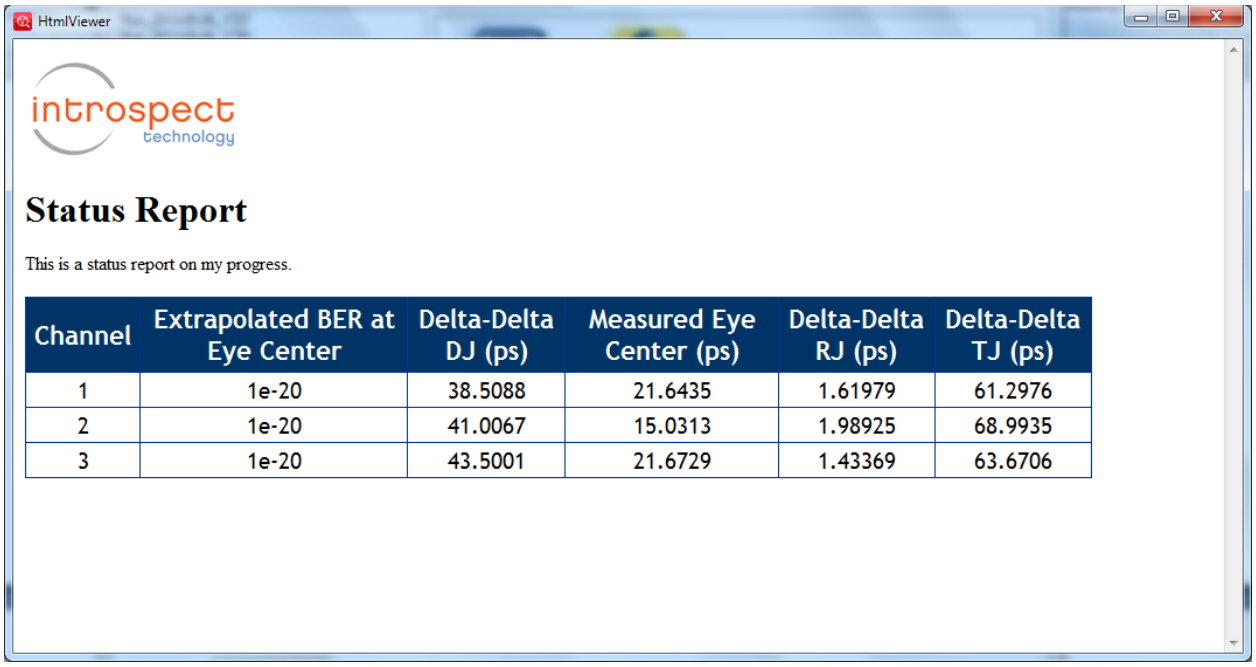


Figure 7 Output of latest execution of Test.

Adding Images

The HtmlReportWriter component can add images to your HTML report and automatically display a gallery of thumbnail images with the full-size images available upon mouse-click. For example, let's suppose that your Test uses the PlotCreator component to create plots of some measurement data. The 'run' methods of PlotCreator and PlotCreatorBasic return the full path to the image files that they create. Suppose that your Test Procedure had stored these image file paths in a variable called 'myImageFilePaths' which is a Python dictionary (keyed by the image titles). Then you could add these images to your HTML report with the following code:

```
reportWriter1.addImageGalleryStart()
for title in sorted(myImageFilePaths.keys()):
      imageFilePath = myImageFilePaths[title]
      reportWriter1.addImage(imageFilePath, title)
reportWriter1.addImageGalleryEnd()
```

Here's a more concrete example of adding images to your HTML report. Please run the EyeScan wizard to create a component called eyeScan1. Just like in the previous bertScan1 example, place the eyeScan1.run() command inside the createReport function. After execution of the eyeScan1.run() operation, we will extract graphics. Specifically, you can use the EyeScan method 'createGraphicsForLastRun' to create image files with screen

captures of the EyeScan viewer with the data from the most recent run of the EyeScan component. The following lines of code would add these screen captures into your HTML report:

```
imageFilePaths = eyeScan1.createGraphicsForLastRun()
channels = sorted(imageFilePaths.keys())
reportWriter1.addImageGalleryStart()
for channel in channels:
      imageFilePath = imageFilePaths[channel]
      title = "EyeScan channel%d" % channel
      reportWriter1.addImage(imageFilePath, title)
reportWriter1.addImageGalleryEnd()
```

The resulting createReport() function is shown in Figure 8 where we have placed the eyeScan1.run() command right after the code for generating a table of jitter statistics from the previous bertScan run.

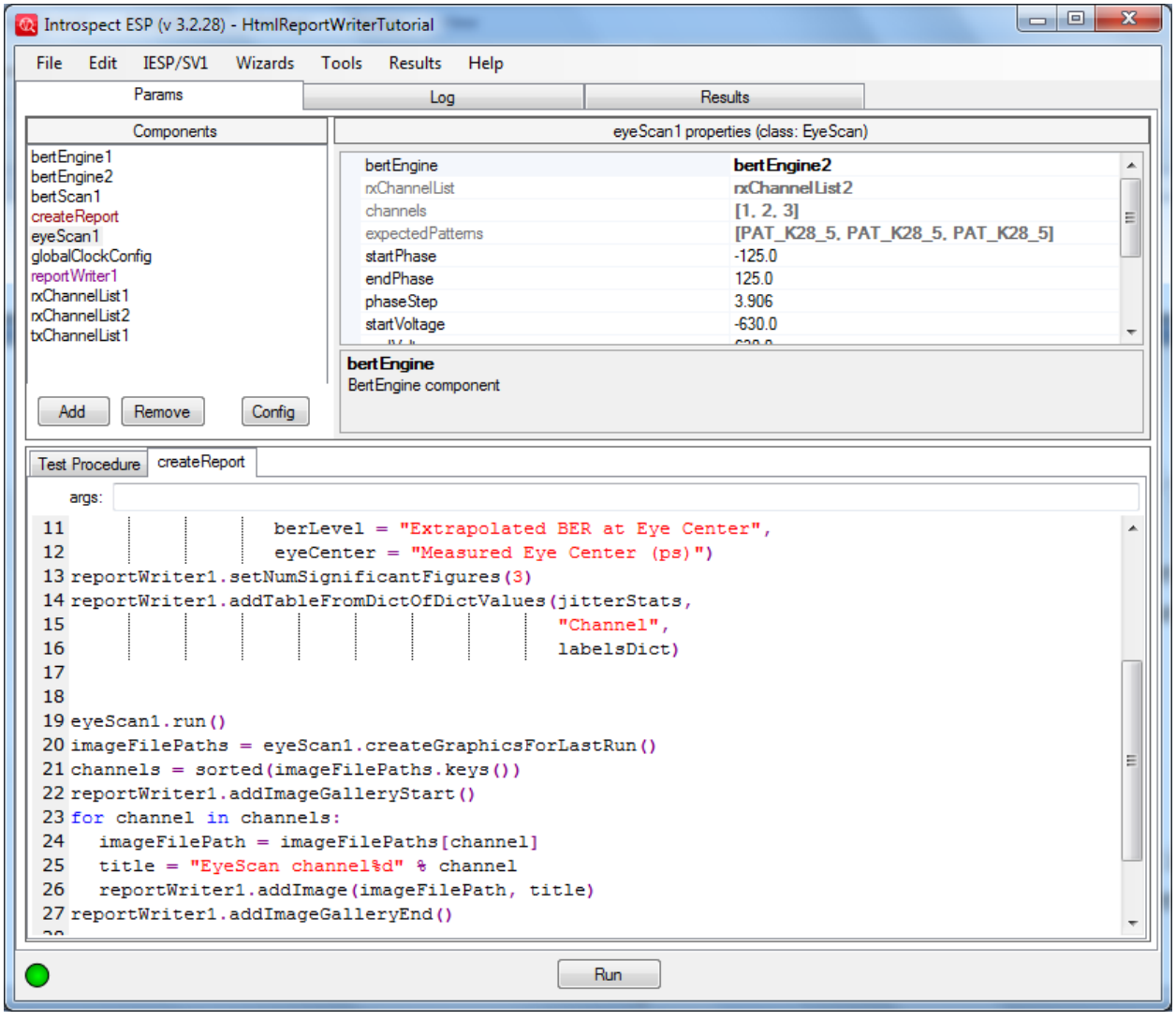


Figure 8 CreateReport function with eyeScan1 component and results gallery.

See the documentation for the “HtmlReportWriter” class in “svt.html” for more details on the ‘addImage’ method. In particular, note that you can specify the size of the thumbnail images if desired. At an rate, executing the code of Figure 8 generates the report in Figure 9.

By default, the image files are copied locally (into an “images” sub-folder) so that the folder containing the HTML report is self-sufficient – i.e. so you can send someone just that one folder and all the required image files (and other support files) will be in that folder. But if you don’t want to make local copies of the images (e.g. if you are creating an HTML report afterwards and using image files that you have manually copied into a local folder), you can tell the ‘addImage’ method not to do any copying.

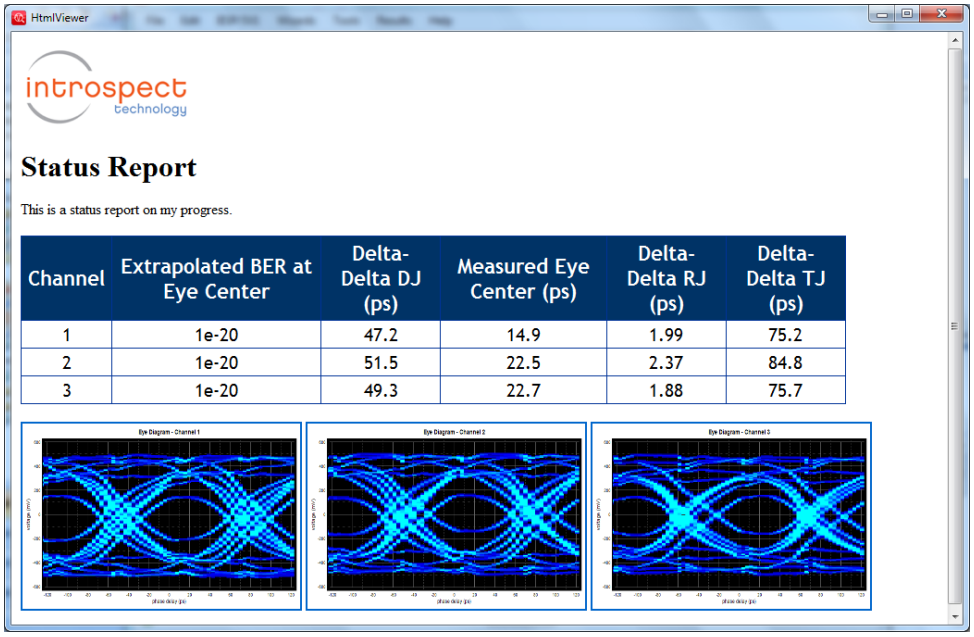


Figure 9 Generated report with integrated image galleries from eyeScan1

There are some convenience methods for the more common image gallery scenarios that can make your code much more concise. The last example could make use of the ‘addImageGalleryFromDictOfImages’ method:

```
imageFilePaths = eyeScan1.createGraphicsForLastRun()
reportWriter1.addImageGalleryFromDictOfImages(imageFilePaths,
                                              "EyeScan channel")
```

The first argument is the dictionary of image file paths. The second argument is a string that will be prepended to each dictionary key (here the channel numbers) to make the image titles.

Now, the code is much simpler as shown in Figure 10.

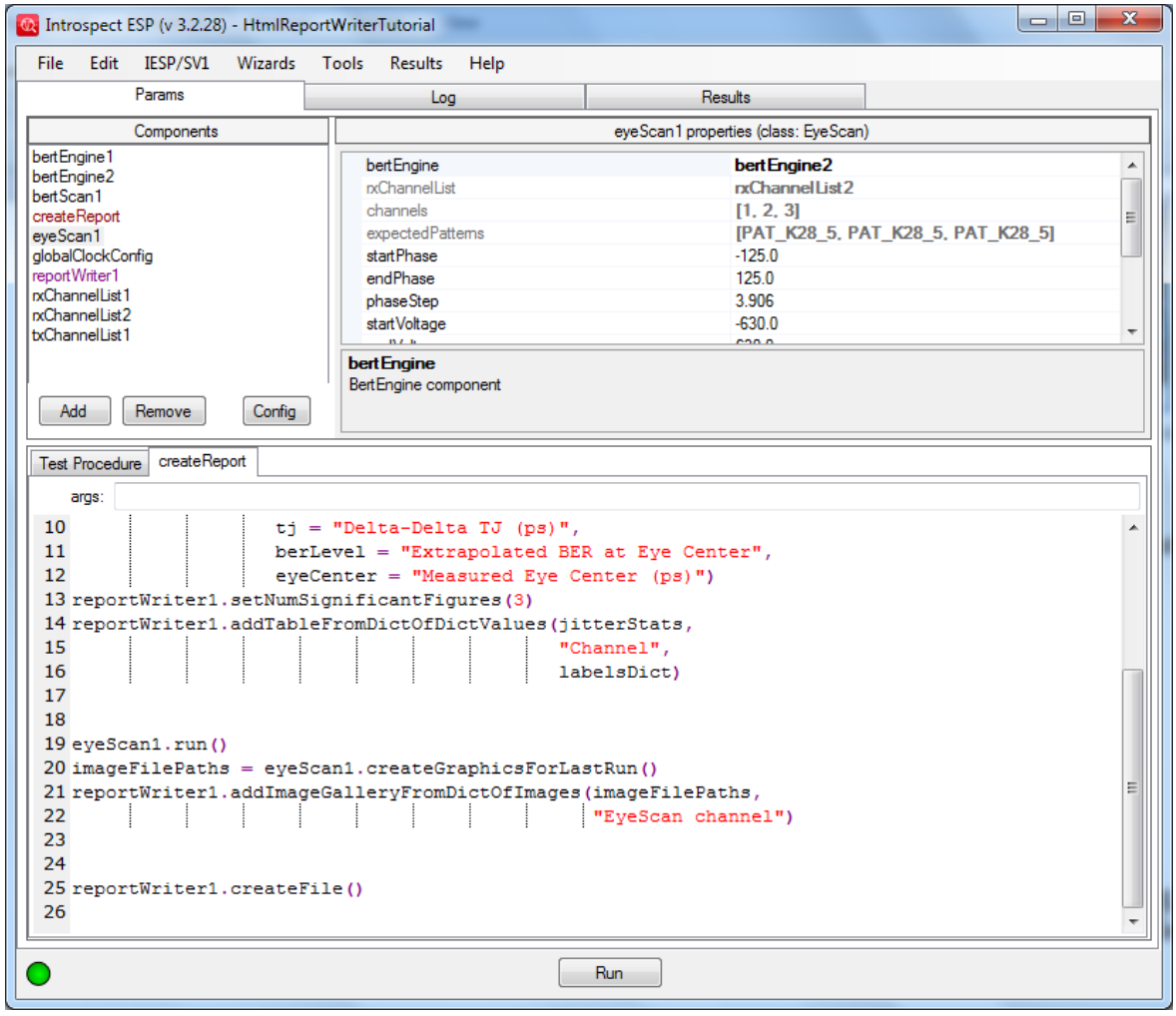


Figure 10 Simplified code for image gallery generation after eyeScan execution.

Creating Off-Line Reports

Usually the HtmlReportWriter is used to create reports from data measured during the course of the current Test Procedure. I.e. the data is measured, and a report is generated as part of the same Test run. But it is also possible to use the HtmlReportWriter to generate reports using data and/or images that you have gathered from previous Test runs (or other sources). We might refer to these as "offline" reports since the report generation is done at a later time, with no connection to the hardware being needed.

If you are doing this kind of thing, it is convenient to create a new Test for the purpose of creating the report, and then to copy all of the data and images that will be used for the report into the "Params" sub-folder of the Test folder. (This Test will use the data and images as input for its procedure, so you can think of the data and images as being like "parameters" for the report-generating-Test.)

Let us go ahead and create a blank Test called "offlineHtml". The Test will be completely empty as in Figure 11, since the first step will be to copy a bertScan1 result folder into the Params folder of this test.

As you have seen in the examples above, you will need to be able to pass the full path to the image files to the HtmlReportWriter methods. Similarly, you will need to pass the full paths to the data files or folders if you are going to have your Test Procedure read these data files using methods like the 'readResultFiles' methods of EyeScan, BertScan, etc. To get the path of a file that is in the "Params" sub-folder of the currently running Test, you can use the SVT function 'getTestParamsFilePath'. For example:

```
filePath1 = getTestParamsFilePath("eyeScanCapture1.png")
```

This same function can be used (despite its name) to get the full path to a sub-folder of the "Params" folder – e.g.:

```
folderPath = getTestParamsFilePath("eyeScanResultsFolder")
```

Note that while it is convenient to have the data files inside the Test that is generating the report, it isn't necessary – if you don't want to copy the data files, you can just refer to them using a full path to their current location.

Once you have the paths to the data or image files needed for your report, you can proceed exactly as in the examples above.

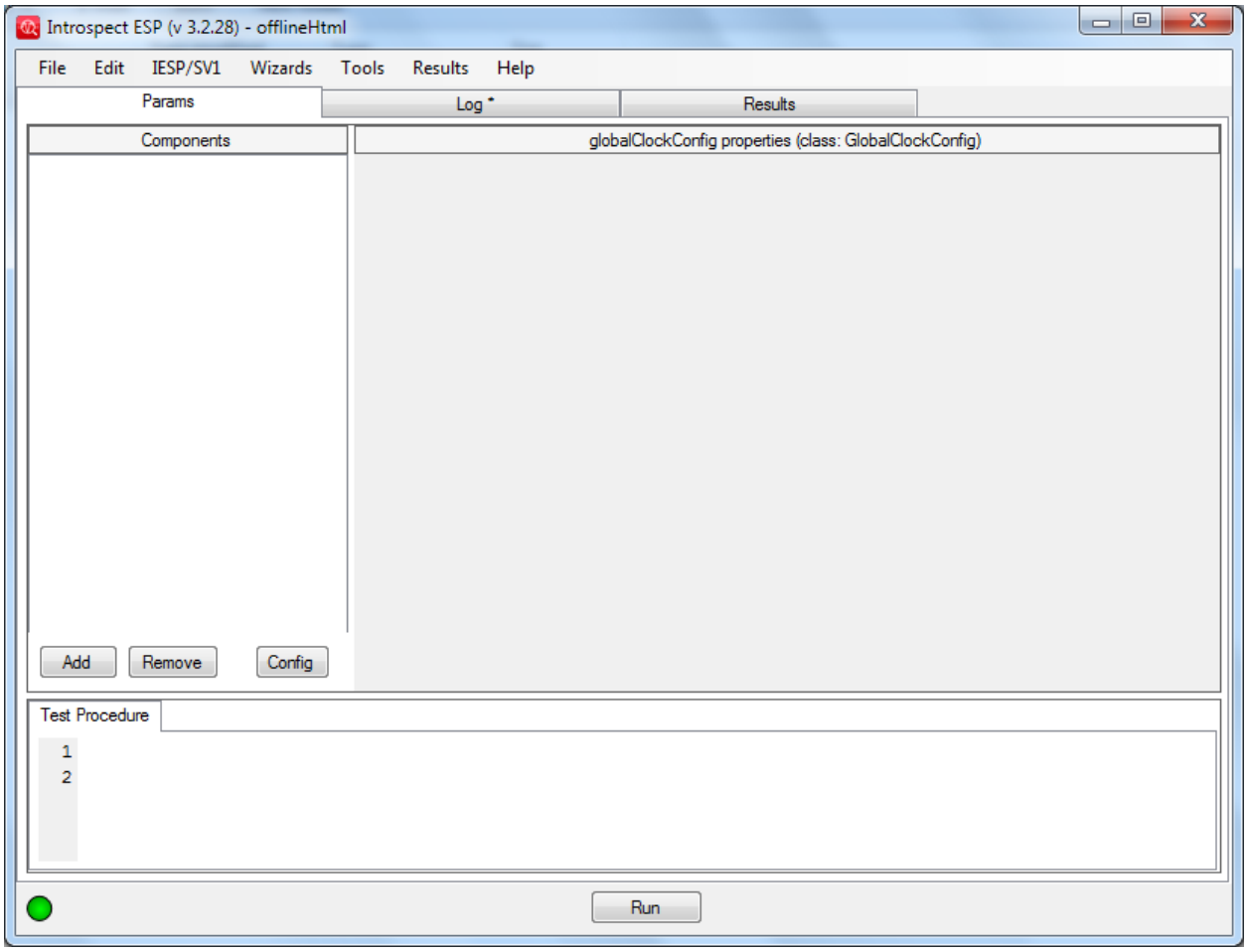


Figure 11 Empty Test that is created for the purpose of storing bertScan1 results inside the Params folder.

Here's an example that reproduces the report on jitter from a BertScan that we did above. Suppose we had previously run a Test that did a BertScan and now we want to produce an "offline" report from those results. To make things convenient, we create a new Test, save it, and then copy the "bertScan1" results folder from the previous Test into the "Params" sub-folder of the new Test. We remove the 'globalClockConfig' component (since we won't be using the hardware at all). And we add an HtmlReportWriter component (renaming it to 'reportWriter1' as before) and a FunctionWithResults component (renaming it to 'createReport').

We will use the BertScan method "readResultFiles" to read the data files into the same type of variables that we would have been returned from the 'run' method. (See the documentation on SvtBertScan in "svt.html" for details on the 'readResultsFiles' method.) Here's the code for the 'createReport' function that will create the same report as we saw earlier:

```
# Read the results from the "bertScan1" folder
# that was copied into the "Params" folder
dataFolderPath = getTestParamsFilePath("bertScan1")
(metadata, results) = SvtBertScan.readResultFiles(dataFolderPath)
(calibDelays, errCounts, jitterAnalysisResults) = results
jitterStats = jitterAnalysisResults['jitterStats']

# now generate an HTML report from 'jitterStats'
folderPath = self.createRunResultFolder("StatusReport",
                                         "HtmlReport")
reportWriter1.start(folderPath)
reportWriter1.addHeading(1, "Status Report")
reportWriter1.addParagraph("This is a status report on my progress.")
labelsDict = dict(rj = "Delta-Delta RJ (ps)",
                  dj = "Delta-Delta DJ (ps)",
                  tj = "Delta-Delta TJ (ps)",
                  berLevel = "Extrapolated BER at Eye Center",
                  eyeCenter = "Measured Eye Center (ps)")
reportWriter1.addTableFromDictOfDictValues(jitterStats,
                                           "Channel",
                                           labelsDict)
reportWriter1.createFile()
```

The final Test Procedure looks like Figure 12. Again, note that we have defined a FunctionWithResults component, called it createReport, and then entered the code inside this function. This is done so that Introspect ESP is able to generate a result icon after execution and to organize the html output in a manner that is consistent with all component executions.

Test Procedure | createReport

args:

```
# Read the results from the "bertScan1" folder
# that was copied into the "Params" folder
dataFolderPath = getTestParamsFilePath("bertScan1")
(metadata, results) = SvtBertScan.readResultFiles(dataFolderPath)
(calibDelays, errCounts, jitterAnalysisResults) = results
jitterStats = jitterAnalysisResults['jitterStats']

# now generate an HTML report from 'jitterStats'
folderPath = self.createRunResultFolder("StatusReport",

    "HtmlReport")
reportWriter1.start(folderPath)
reportWriter1.addHeading(1, "Status Report")
reportWriter1.addParagraph("This is a status report on my progress.")
labelsDict = dict(rj = "Delta-Delta RJ (ps)",
                  dj = "Delta-Delta DJ (ps)",
                  tj = "Delta-Delta TJ (ps)",
                  berLevel = "Extrapolated BER at Eye Center",
                  eyeCenter = "Measured Eye Center (ps)")
reportWriter1.addTableFromDictOfDictValues(jitterStats,
                                           "Channel",
                                           labelsDict)
reportWriter1.createFile()
```

Figure 12 Function to read offline data and generate an html report from this data.