introspect technology

TECHNICAL INSIGHTS

Coordinator Utility

Published in Canada on January 11, 2021

INTROSPECT.CA

Table of Contents

Introduction

This document will explain how to coordinate actions across multiple Introspect hardware boxes using the Introspect ESP Software. This is done via the "Coordinator" facility which allows you to:

- Interoperate between RX and TX boxes
- Control a remote box (in the lab, or anywhere on the Internet)
- Coordinate actions on two or more boxes to provide more channels

To coordinate the actions of two or more instances of Introspect ESP Software (on one or more computers), the software instances communicate with each other via messages sent over a server. These messages can tell the targeted software to perform an action (by invoking a component method or by executing arbitrary Python code), and to return the result. For example, a TX box could start a transmission and then tell the RX box to read incoming data and return the results of data analysis.

The Coordinator Server

The communication between the boxes is done via the "Coordinator Server". You can run the Coordinator Server on the same computer that you are using for controlling one of the Introspect hardware boxes, or you can run it on a different computer. The Coordinator Server is usually started from the menu item "Coordinator Server" in the Tools menu of the Introspect ESP Software. It can also be started from the command-line. But note that you only want one instance of the server running at any time.

Coordinator Clients

After the server has been started, you create an instance of the Coordinator component on each of the instances of the Introspect ESP Software that will be controlling the hardware. You can have several instances of the Introspect ESP Software on one computer, or you can have the Introspect ESP Software running on separate computers. The Coordinator component is in the "utility" section of the Add Component list.

You tell each of the Coordinator components the host name (or IP address) and port number of the Coordinator Server (via the 'serverHostName' and 'serverPort' properties) so it knows where to send messages. And you specify a unique 'clientName' for each of the Coordinator components – usually you name them according to the purpose of the hardware that will be used by the Introspect ESP Software. For example, if you have two boxes, one of which is a transmitter, and the other a receiver, you might name the Coordinator components "tx_box" and "rx_box" respectively. Each Coordinator component acts as a client of the Coordinator Server – it sends messages (requests) to the server and gets back responses.

All interactions between the Coordinator components (clients) and the server (and hence other Coordinator components) is done via Coordinator component method calls. See the Coordinator component documentation (available from the Help menu) for details of the available methods.

Examples

The following examples illustrate some of the various ways that the Coordinator facility may be used. The first example uses the methods for specifying that a Coordinator client is in a certain state, and for querying the state of other Coordinator clients. This example also shows the log messages that give information about what activity is seen by the Coordinator server, and hence can be used for debugging Coordinator issues.

Subsequent examples show the use of the Coordinator facilities that allow a Coordinator client to request execution of code on another client, and they show the use of message passing for information sharing.

Many of the examples are runnable without access to hardware since they don't do any measurements. In your actual usage, you would of course be doing things that use the hardware. There are some examples that do a BertScan, requiring hardware that supports that operation, but these can easily be modified for the operations supported by that hardware that you are using.

The easiest way (conceptually, and programmatically) to use the Coordinator facility is to have one of the clients be the "master" and the other be the "slave", where the slave waits for requests/commands from the master, and then executes them and returns results to the master. We will show examples that use this paradigm. But it is also possible to have the Coordinator clients on a more equal footing, where they take turns doing the required operations.

BASIC EXAMPLE

Start by launching two separate instances of the Introspect ESP Software on the same computer. (You do this by double-clicking on the Introspect ESP Software icon a second time after the first instance has started, or by using the menu item "New Application Instance" from the "File" menu.) You should get the startup dialog for each of the two instances. For this example, it doesn't matter what form factor you choose – use whichever one is most familiar to you. Choose the option to create a new test in each of the two Introspect ESP Software instances. To make these two software instances easier to tell apart, save the Test in each of them, giving easily distinguishable names – e.g. "CoordinatorTestA" and "CoordinatorTestB". Your screen should now look like this:

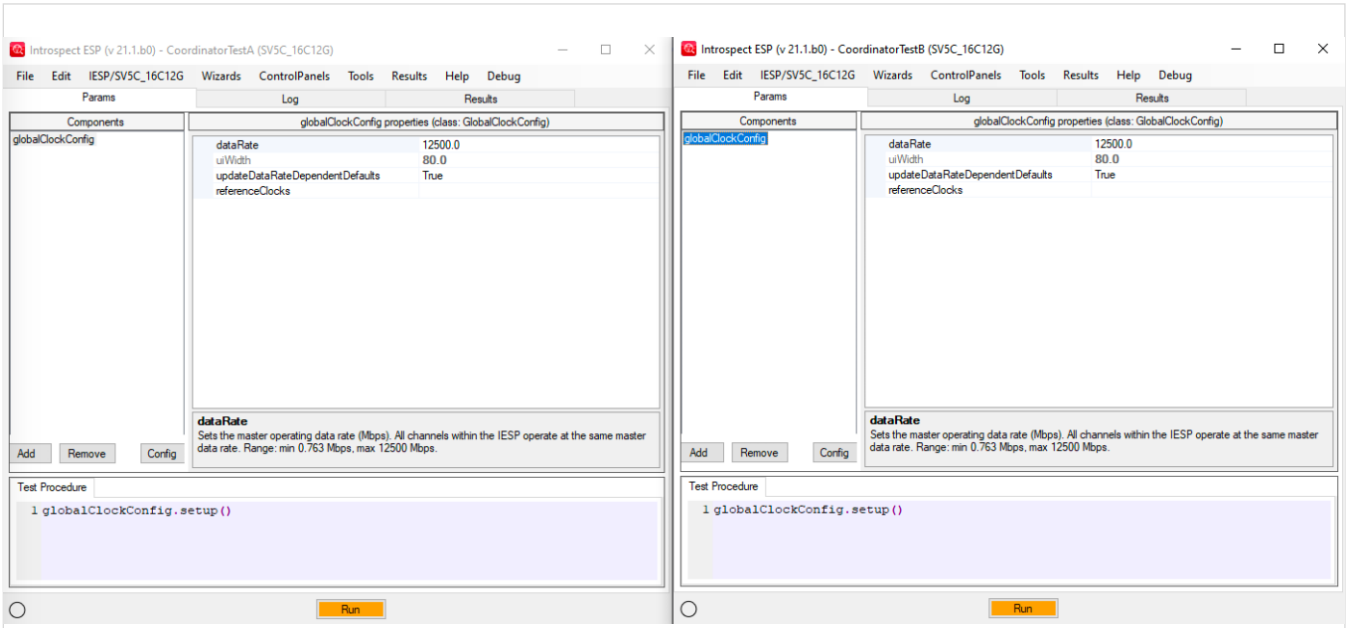

Figure 1: Two Test Procedures in the basic example of this document.

Now start a Coordinator Server on one of the two instances (it doesn't matter which one) by using the "Coordinator Server" menu item from the "Tools" menu. You will get a window like this:

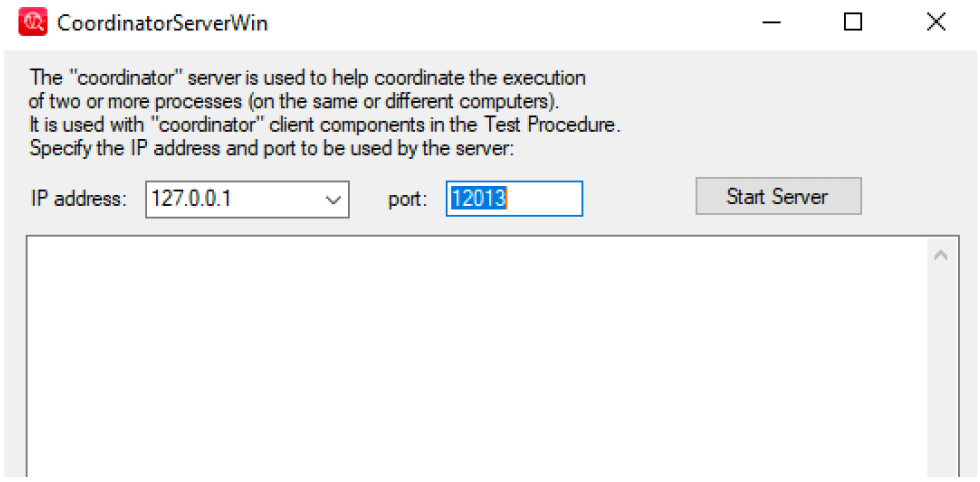

Figure 2: Coordinator server.

For this example, leave the "IP address" and "port" at their default values. These specify that the server will use the "localhost" address (127.0.0.1) of the computer and the (arbitrarily chosen) port number 12013. Click on the "Start Server" button. You should see some initial messages from the server in the text area at the bottom of that window. You can leave the Coordinator Server window open if you want to continue to see the messages from the server – sometimes these are useful for debugging. You can stop the server (when finished using the Coordinator facility) by using the "Stop Server" button. The server will be automatically stopped when you quit the Introspect ESP Software.

Now create an instance of the Coordinator component in each of the two Introspect ESP Software instances. (Note that the Coordinator component is in the "utility" section of the Add Component list). Leave the coordinator component properties 'serverHostName' and 'serverPort' at their default values which should match those that were specified when you started the server. Rename the two Coordinator components to be "AAA" and "BBB". (Note: to rename a component, you click once on its name in the list and then the name becomes editable.) You can leave the 'clientName' property of each of the two Coordinator components blank – the clientName will then be the name of the component ("AAA" or "BBB").

Delete the "globalClockConfig" components – we don't need them in this example.

A common problem when using the Coordinator facility is to get the coordinator clients known to the server before trying to communicate between them. Just creating a Coordinator component doesn't make it known to the server – the component needs to issue a command to make itself be known to the server.

For this example, we will have each of the Coordinator components issue a command to tell the server its "state".

Add the following code in the Test Procedure of "CoordinatorTestA":

```
AAA.setState("ready")
```

And add the following code in the Test Procedure of "CoordinatorTestB":

```
BBB.setState("ready")
```

Now click on the "Run" button in "CoordinatorTestA", and then in "CoordinatorTestB". Look in the Coordinator Server window (you can open it again from the "Tools" menu if you had closed it before). You should see messages like the following about the two coordinator clients being registered with the server, then their "state" being set, and finally the two clients being unregistered. That means that the coordinator clients are only known to the server while the Test is running. Once the Test has finished, the coordinator clients are unregistered, so they are once again unknown to the server – even though the server continues to run.

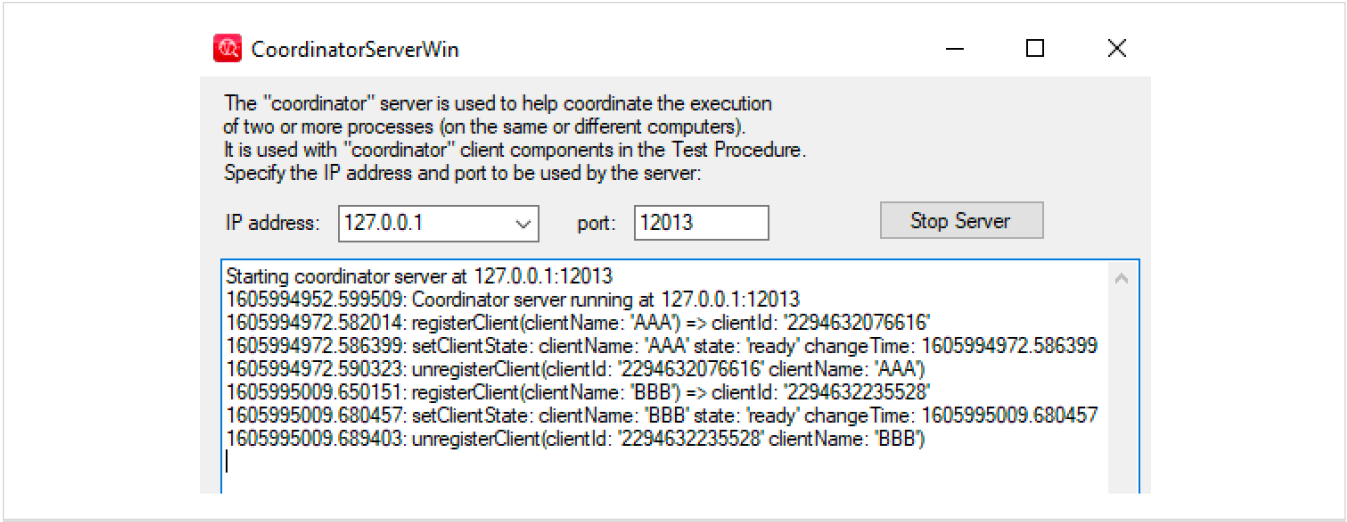

Figure 3: Coordinator server log window.

If we want to communicate between two Coordinator components, we need the Test to stay running until the communication is finished. So, add the following lines to the Test Procedure of "CoordinatorTestB":

```
for i in range(40):
    sleepMillis(1000)
```

This will keep the Test running for 40 seconds after the Coordinator component "BBB" sets its state. This provides time for the other Test to communicate with it.

Now add the following lines to the Test Procedure of "CoordinatorTestA":

```
waitForClientToExist("BBB", timeout = 60)
stateBBB = AAA.getState("BBB")
print("stateBBB: %s" % str(stateBBB))
```

Your screen should look like this:

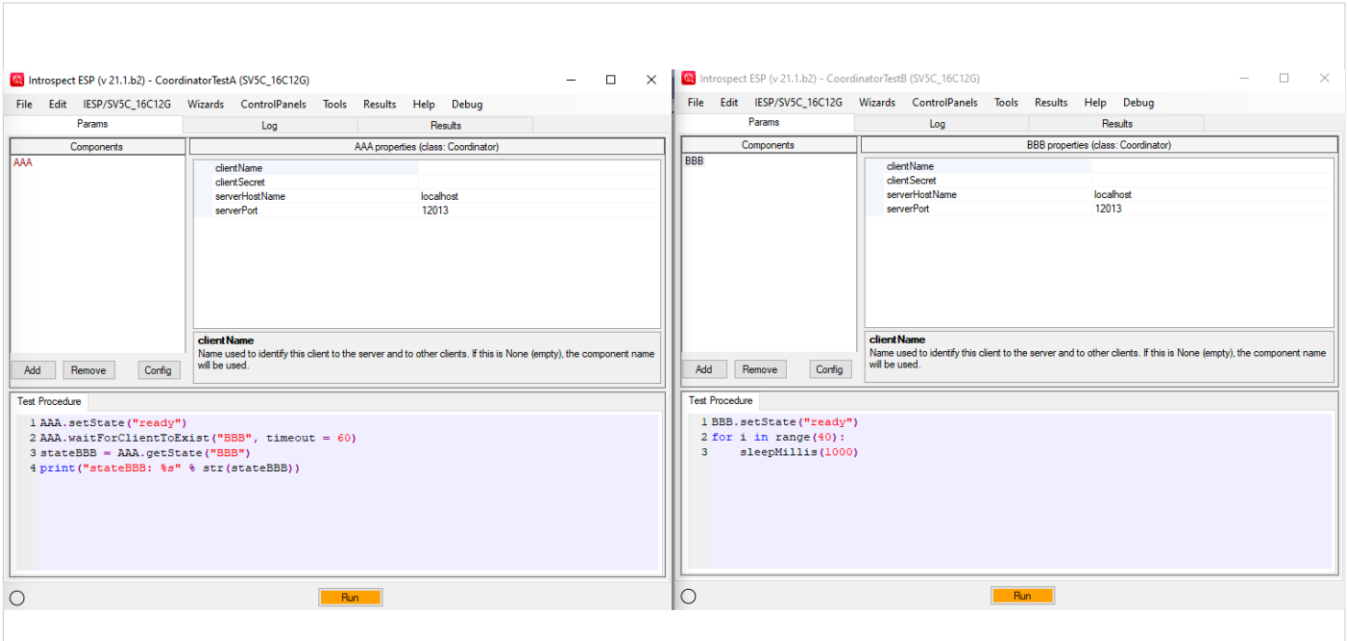

Figure 4: Two Test Procedures with code for client AAA to query the state of client BBB through the coordinator.

Now click on the "Run" button in "CoordinatorTestA", and then in "CoordinatorTestB". You should see something like the following in the "Log" tab of "CoordinatorTestA":

```
Components used by Test Procedure: [AAA]
IESP not used by Test Procedure
stateBBB: ('ready', 1606084136.3461993)
Test finished
```

This shows that the state ("ready") of the Coordinator component "BBB" was correctly queried from Coordinator component "AAA". The number after the "ready" is a timestamp giving the time that the state of "BBB" was set. Note that the state of "BBB" was queried while the Test in "CoordinatorTestB" was sleeping – this works because the server knew the state of "BBB" so the only communication was between "AAA" and the server. If you look in the Coordinator Server window, you will see the messages sent back and forth from the two Coordinator components.

Now we'll make the Coordinator component "BBB" change to various different states, and have the component "AAA" wait for "BBB" to be in a particular state, and then send it a message.

Change the Test Procedure of "CoordinatorTestB" to be:

```
BBB.setState("ready")
sleepMillis(13000)
BBB.setState("alpha")
sleepMillis(13000)
BBB.setState("beta")
messageInfo = BBB.waitForMessage()
(timestamp, senderName, msg) = messageInfo
print("Message from %s: %s" % (senderName, msg))
sleepMillis(13000)
```

And add the following lines to the Test Procedure of "CoordinatorTestA":

```
AAA.waitForClientState("BBB", "beta", timeout = 30)
AAA.sendMessage("BBB", "Hello from AAA")
```

So, component "BBB" will change state from "ready", to "alpha", and then to "beta". When component "AAA" detects that the state of "BBB" is "beta", it will send it a message, which "BBB" will wait for, and then print. Your screen should look like this:

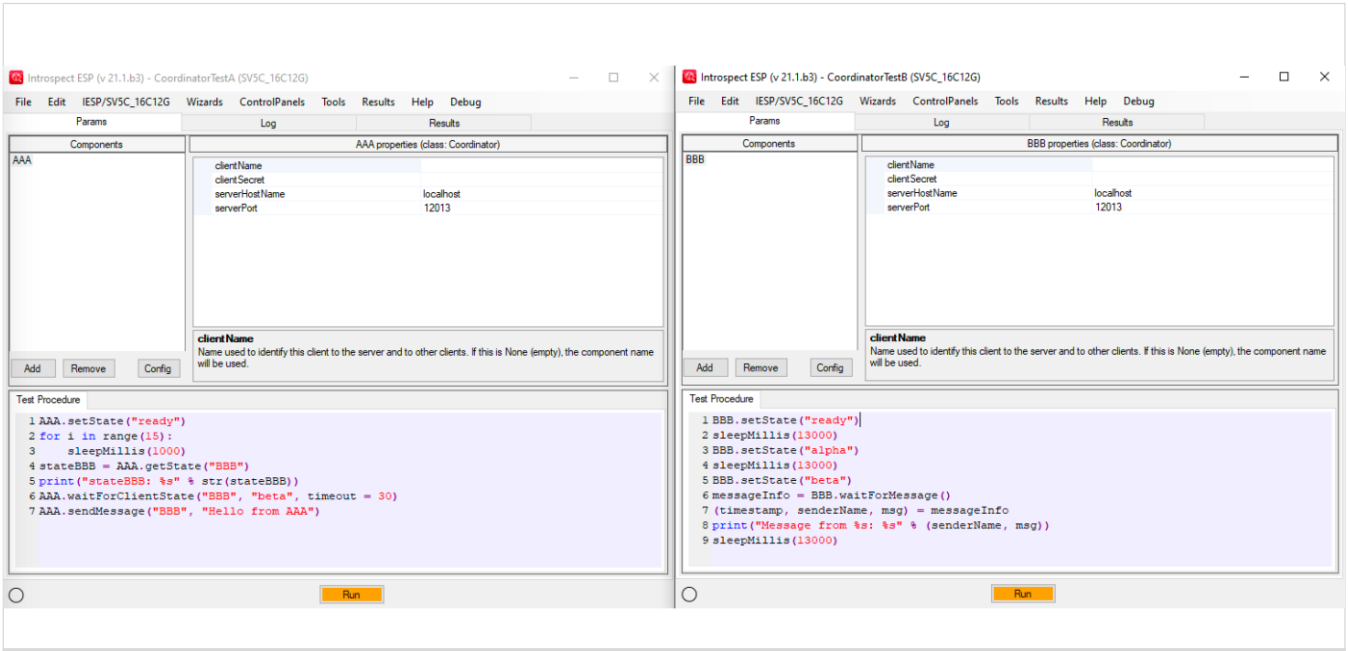

Figure 5: Two Test Procedures where client AAA waits for client BBB to reach a certain state before communicating with it.

Now click on the "Run" button in "CoordinatorTestA", and then in "CoordinatorTestB". You should see something like the following in the "Log" tab of "CoordinatorTestB":

```
Components used by Test Procedure: [BBB]
IESP not used by Test Procedure
Message from AAA: Hello from AAA
Test finished
```

MASTER/SLAVE EXAMPLE: DIGITS OF PI

In this example, we'll have one of the Tests (the "slave") being completely passive, waiting for commands from the "master".

Quit both of the instances of Introspect ESP Software (from the previous example) and start fresh. Then, as in the previous example, launch two separate instances of Introspect ESP Software, each with a new Test, and start the Coordinator Server in one of those instances. Create an instance of the Coordinator component in each of the two instances of Introspect ESP Software (naming them as before "AAA" and "BBB"). Delete the two instances of GlobalClockConfig. And to make things easier to refer to, save the two Tests as "CoordinatorTestA2" and "CoordinatorTestB2".

In "CoordinatorTestB2", which will be the "slave", make the Test Procedure be:

```
from dftm.funUtil import piDigitGenerator
piDigits = piDigitGenerator()
BBB.loopRunningCodeFromClient("AAA")
sleepMillis(10000)
```

The function 'piDigitGenerator' is an undocumented utility function which generates the digits of π. Here we have set up "CoordinatorTestB2" to be completely passive – it loops waiting for some Python code to be sent to it from the other Coordinator client, and executes that Python code. This loop will continue forever – or until the client "AAA" stops sending code to be executed. We added a 10 second sleep at the end of the Test Procedure so that the client "BBB" stays around until "AAA" finishes processing the results.

In "CoordinatorTestA2", which will be the "master", make the Test Procedure be:

```
numDigitsDesired = 20
for i in range(numDigitsDesired):
    codeStr = "next(piDigits)"
    codeResult = AAA.waitForCodeToBeRun("BBB",
                                        exprToEval = codeStr)
    (timestamp, digit) = codeResult
    print(digit)
    sleepMillis(300)
```

Here, we loop over 'numDigitsDesired', sending the Python expression "next(piDigits)" to be evaluated by the client "BBB". Note that the variable 'piDigits' exists in the Test Procedure of "CoordinatorTestB2" only. The 'next' function acting on this variable gets the next value from the list that is produced by the piDigitGenerator function (which again only ran on "CoordinatorTestB2").

The 'waitForCodeToBeRun' method returns a tuple in which we store the variable 'codeResult'. The second item in that tuple is the result of evaluating the expression that was sent to "BBB" – i.e. the next digit of π.

Your screen should look like this:

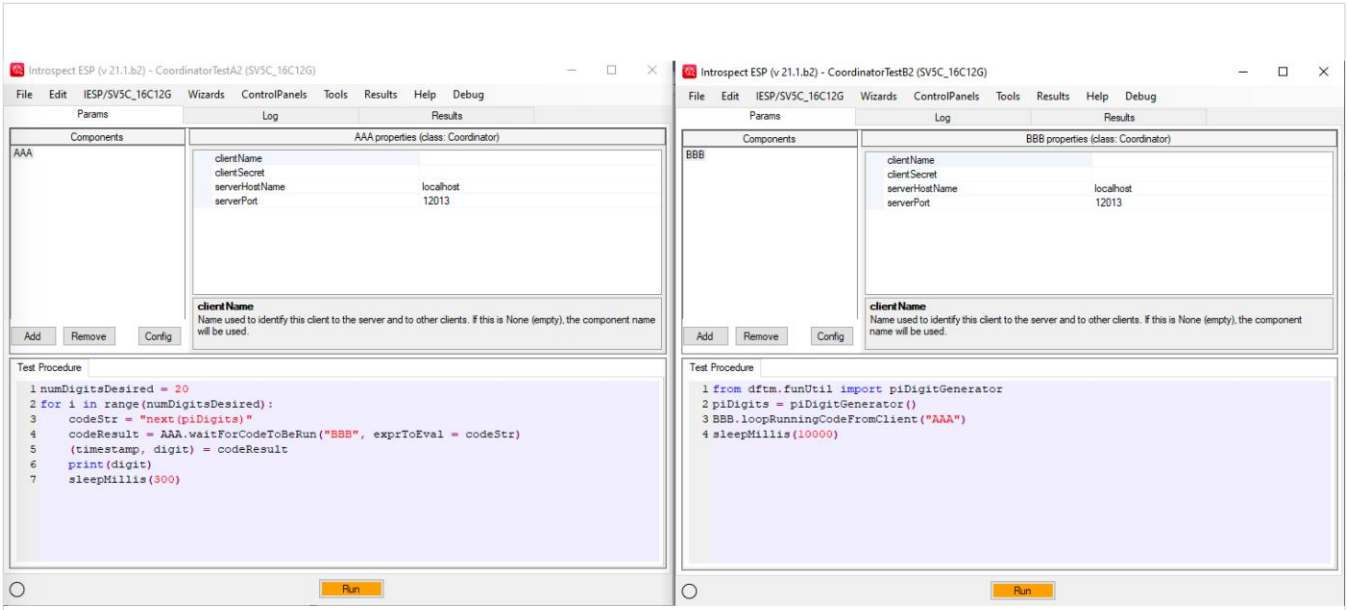

Figure 6: An example showing how to make AAA send code to be executed within the BBB Test Procedure.

Now click on the "Run" button in "CoordinatorTestB", and then in "CoordinatorTestA". You should see something like the following in the "Log" tab of "CoordinatorTestA":

```
Components used by Test Procedure: [AAA]
IESP not used by Test Procedure
3
1
4
1
5
9
2
6
5
3
5
8
9
7
9
3
2
3
8
4
Test finished
```

So what we've got here is a "slave" that knows how to calculate digits of π, and a "master" that issues repeated requests for the slave to calculate the next digit. Of course, the slave could equally well be

doing measurements of some kind and analyzing the results which are sent back to the master. That's what we'll do in the next example.

But before going on, note that the method 'waitForCodeToBeRun' also has an optional argument 'codeToExec'. We used the argument 'exprToEval' above because we just wanted to evaluate an expression, but both these arguments can be used in one call to 'waitForCodeToBeRun'. For example, we might want to run a BertScan, and then get the RJ from the result of that BertScan. So, we might use something like this:

```
codeStr1 = "bertScanResult = bertScan1.run()"
codeStr2 = "bertScanResult.getAnalysisValue(channel, 'RJ')"
codeResult = AAA.waitForCodeToBeRun("BBB",
                                    codeToExec = codeStr1,
                                    exprToEval = codeStr2)
(timestamp, rj) = codeResult
```

This would execute the code "bertScanResult = bertScan1.run()" and then evaluate the expression "bertScanResult.getAnalysisValue(channel, 'RJ')" to get the RJ value. We will do something similar in the next example, but by predefining a Function on the slave and then calling that Function.

MASTER/SLAVE EXAMPLE: BERTSCAN

In this example we will have the slave do a BertScan upon request and return the estimated RJ (random jitter) to the master. The master will be transmitting a pattern and changing the dataRate for each measurement. The code for this example (with a TxChannelList on the master and a BertScan on the slave) is applicable to the non-MIPI hardware – for MIPI hardware, you could use a MIPI Generator component and do a PertMeasurement on the RX side.

Quit both of the instances of Introspect ESP Software (from the previous example) and start fresh. Then, as in the previous example, launch two separate instances of Introspect ESP Software (specifying the form factors appropriately for the TX and RX sides), and create a new Test in each. Start the Coordinator Server in one of those instances. Create an instance of the Coordinator component in each of the two instances of Introspect ESP Software (naming them as before "AAA" and "BBB"). This time we'll keep the two instances of GlobalClockConfig – we will be using them to set the dataRate. Finally, to make things easier to refer to, save the two Tests as "CoordinatorTestA3" and "CoordinatorTestB3".

In "CoordinatorTestB3", which will be the "slave", create an instance of the RxChannelList component. Create an instance of the BertScan component. The 'rxChannelList' property of the BertScan component should have been automatically set to point to the RxChannelList component that was created just before. Set the 'saveResults' property of the BertScan component to False – we don't want the results saved to CSV files here since we will be programmatically extracting analysis results.

Create an instance of the Function component (it is in the 'utility' section of the Add Component list), rename it to "getRjFromBertScan" and set the 'args' of the "getRjFromBertScan" Function to:

```
dataRate, patternName
```

And set the code of the "getRjFromBertScan" Function to:

```
globalClockConfig.dataRate = dataRate
globalClockConfig.update()
rxChannelList1.expectedPatterns = [patternName]
bertScanResult = bertScan1.run()
channel = 1
rj = bertScanResult.getAnalysisValue(channel, 'RJ')
return rj
```

The Function "getRjFromBertScan" first changes the dataRate (on the RX side) to the value passed to the function as the first argument.

Then it sets the 'expectedPatterns' property of the RxChannelList component (assumed to be 'rxChannelList1') to a list with the pattern passed to the function as the second argument. Here we are taking advantage of the auto-convert facility of the software to convert the pattern name to a Pattern object.

Finally, it runs the BertScan (which uses the RxChannelList) and gets the RJ value for channel 1 from the BertScan analysis. The return value of the Function is the RJ value.

Your Function component should look like this:

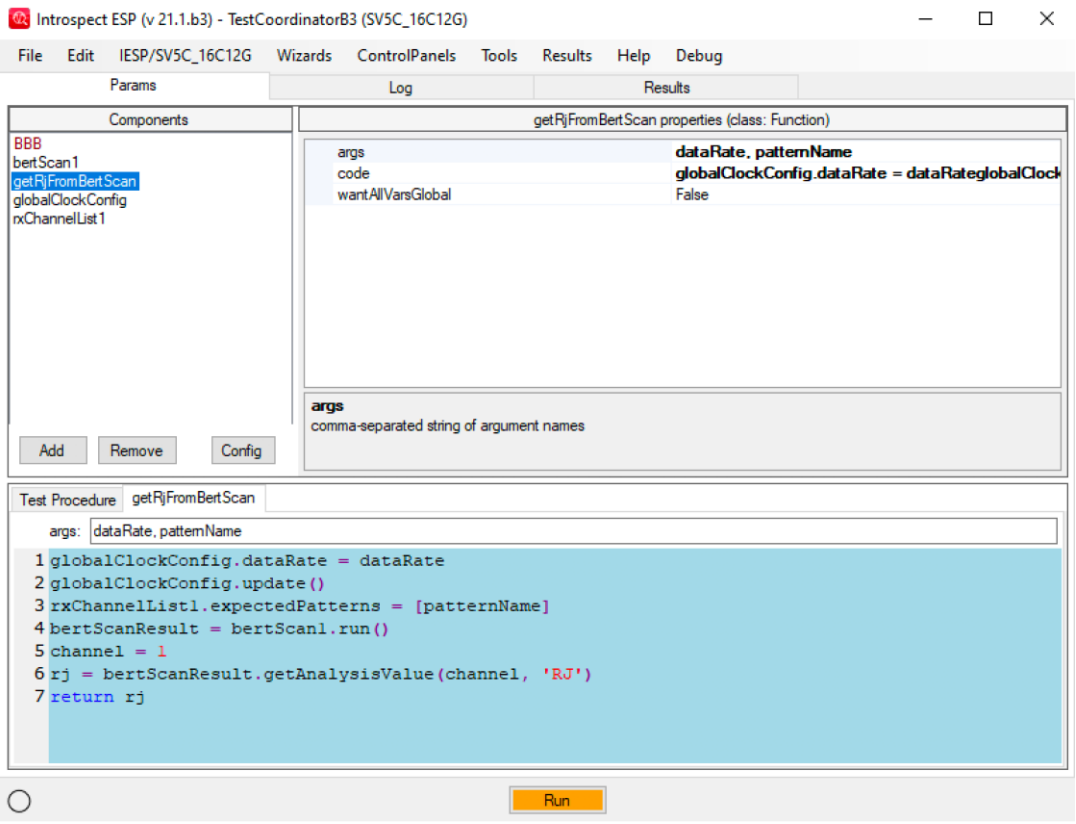

Figure 7: Function component declaration in the current example.

Now edit the Test Procedure of "CoordinatorTestB3" so it is like this:

```
globalClockConfig.setup()
BBB.loopRunningCodeFromClient("AAA")
sleepMillis(10000)
```

In "CoordinatorTestA3", which will be the "master", create an instance of the TxChannelList component.

Now edit the Test Procedure of "CoordinatorTestA3" so it is like this:

```
globalClockConfig.setup()
pattern = PAT_PRBS_7
txChannelList1.patterns = [pattern]
dataRates = [2000, 3000, 4000, 5000]
for dataRate in dataRates:
    globalClockConfig.dataRate = dataRate
    globalClockConfig.update()
    txChannelList1.setup()
    funcName = "getRjFromBertScan"
    funcArgs = [dataRate, pattern.name]
    codeResult = AAA.waitForFuncToBeRun("BBB", funcName, funcArgs)
    (timestamp, rj) = codeResult
    print("dataRate: %g RJ: %s" % (dataRate, rj))
    sleepMillis(300)
```

This loops, changing the dataRate, setting up the TxChannelList to transmit the pattern, then asks the "BBB" component to execute the Function "getRjFromBertScan" and print the RJ value that is returned. The current dataRate and the name of the TX pattern are passed as arguments to that function so that the RX side can match the dataRate and expected pattern.

Your screen should look like this:

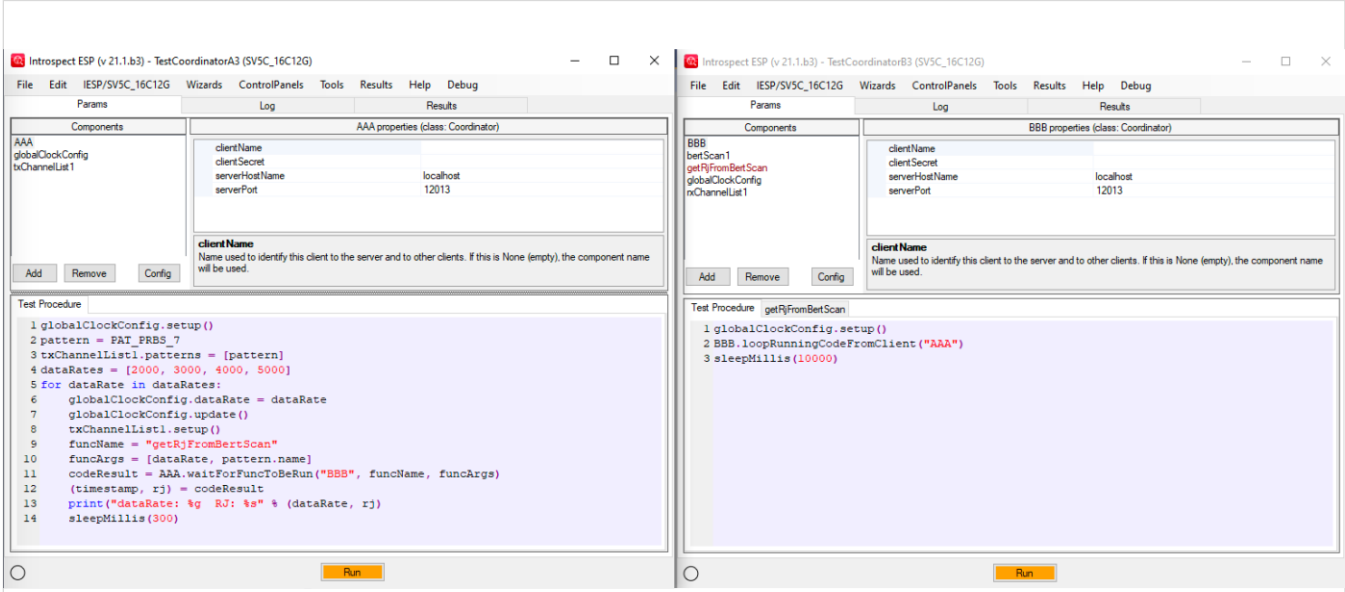

Figure 8: Generating a pattern on AAA and asking BBB to measure its random jitter and return the result.

Now click on the "Run" button in "CoordinatorTestB3", and then in "CoordinatorTestA3". You should see the RJ values for the various dataRates in the "Log" tab of "CoordinatorTestA3".

In this example, we had pre-defined the Function "getRjFromBertScan". This makes things neater, but it is also possible to execute arbitrary strings of Python code. And it's possible to create a Function on the fly on the slave side using the Coordinator component method 'defineFunctionOnClient', and then call it like we did with the pre-defined Function. Please refer to the Help menu for all the features supported by the Coordinator component.

Timeouts

Most of the Coordinator method calls have an optional 'timeout' argument. You've seen this argument being specified in some of the examples above. If you don't specify the 'timeout' argument, you get the default timeout which is usually 20 or 30 seconds. The default is adequate in most situations where the thing that you are waiting for takes much less time than this. But if you needed to walk down the hall to click on the "Run" button of one of the Introspect ESP Software instances, you likely want to specify a longer timeout. Similarly, if you are requesting that the other client perform an action which takes a long time, then you might want a longer timeout. Note that the default behaviour if an operation times out is to issue an error message (which aborts the Test Procedure). An alternative is to check the return values of the methods (see the Help documentation) to determine if there was a timeout, and then handle the situation in some other way.

There is another kind of timeout that might be relevant in some situations – this is the timeout for network operations. If your network is slow, you might need to specify a longer timeout via the Coordinator component method 'setCommunicationTimeout'.

It's not a timeout issue, but if you forget to start the Coordinator server before using Coordinator component methods that interact with the server, you will get error messages that you might attribute to timeout issues. To guard against this kind of forgetfulness, you might want to call the Coordinator component method 'isServerRunning' at the beginning of your Test Procedure.

Security

When you have a Coordinator server running, any instance of the Introspect ESP Software can send it messages/requests. If the name of a Coordinator client is known to an instance of Introspect ESP Software, that instance can send messages/requests to that client. Normally this is not an issue because you are in a friendly environment, but if the Coordinator server is accessible over the Internet, security might be an issue. The Coordinator facility provides a degree of security via the 'clientSecret' property of the Coordinator component. If you specify a value for 'clientSecret', then the other Coordinator clients will need to specify this as the optional argument 'securityKey' in Coordinator method calls like 'waitForCodeToBeRun'.

Running from Scripts

The Coordinator component can be used exactly the same way in scripts (outside of the Introspect ESP Software GUI). The Coordinator server needs to be running somewhere – it can be started via a script (coordinator/coordinatorServerScript.py), or the Tools menu item of the Introspect ESP Software GUI could be used just for convenience. Note that if the server is started via a script, you need to take care of stopping the server when no longer needed.

Conclusion

The Coordinator component is a very powerful utility that allows for controlling multiple Introspect hardware boxes and for communicating between multiple Introspect ESP Software instances. This document introduced the fundamental concepts behind the Coordinator component. It also showed useful examples of how Test Procedures running on two different instances can be made to communicate effectively. The examples shown in this document illustrate how complex automation environments (using hybrid Introspect boxes) can be created.

Revision Number	History	Date
1.0	Document Release	January 11, 2021

The information in this document is subject to change without notice and should not be construed as a commitment by Introspect Technology. While reasonable precautions have been taken, Introspect Technology assumes no responsibility for any errors that may appear in this document.

Published in Canada on January 11, 2021

INTROSPECT.CA