Using Introspect ESP Python Modules in External Python Scripts

This document discusses the use of the Python modules that are supplied with Introspect IESP in external Python scripts that you write.

The Python modules that are supplied (under the "Python/dftm" folder) with Introspect ESP are intended for use by the Test Procedures that you create in the Introspect ESP GUI. These Python modules are used when you run the Test (via the "Run" button) in the GUI and they are used when you run the Test (via the "runSvtTest.py" script) from the command line or from another command-line environment like a script or a batch file. (See separate document "Running Tests from the Command Line".)

It is, however, possible to use the supplied Python modules in external Python scripts that you write (using some text editor, outside the context of Introspect ESP) as long as you supply the context that the Python components expect. This might be useful, for example, if you have an existing Python script that controls other hardware and you want to intersperse calls to control the IESP hardware. (An alternative way would be to use the Introspect ESP facilities to control this other hardware from within the Test Procedure that you write in the Introspect ESP GUI – that is the recommended way.)

Python Version

The first requirement to be able to use the Python modules supplied with Introspect ESP in external Python scripts is that your Python scripts must be using the same version of Python as that supported by Introspect ESP. The current version of Introspect ESP uses Python 2.7 and so your external scripts must be using some 2.7.x version of Python. If you get an error message about a "bad magic number", this means that you are using an incompatible version of Python. You can check which version of Python is the default in your command-line environment by executing the command:

```
python –version
```

Python Module Search Path

In order to be able to use the Python modules supplied with Introspect ESP, the Python interpreter must be able to find them. Python looks for modules in the folder where the script is, in the folders specified in the PYTHONPATH variable, and in the Python installation folders. The search path used when importing modules is in the Python variable 'sys.path' – this is initialized from the folders mentioned above when the script is started. You can print out the value of 'sys.path' in your script as a debugging aid if modules are not being found. To ensure that the Python modules supplied with Introspect ESP are found when running your script, you can either set the PYTHONPATH environment variable to include the parent folder of the "dftm" folder, or append this folder to 'sys.path' at the top of your script.

For example, if your Introspect ESP installation folder is "C:\MyStuff\IntrospectESP" and the "dftm" folder with the Introspect ESP Python modules is in the "Python" sub-folder of that installation folder, then you could set the PYTHONPATH environment variable with the following command in a DOS window:

```
set PYTHONPATH=C:\MyStuff\IntrospectESP\Python
```

or you could add the following line near the top of your Python script:

```
sys.path.append(r"C:\MyStuff\IntrospectESP\Python")
```

Note the use of the 'r' prefix to get a raw string – otherwise the backslashes would cause problems.

Context assumed by Python component classes

The Python component classes from Introspect ESP generally assume that they are being used in the context of a Test – that there is a Test object (of class SvtTest) which "owns" all of the component objects. The component classes that "talk" to the IESP hardware (subclasses of SvtIespComponent) assume that a connection to the IESP hardware has already been established. The runnable components (subclasses of SvtRunnableComponent) assume that their 'run' method will be called from the 'run' method of the Test object.

Configuring for the IESP Hardware Form Factor

The IESP hardware comes in various form factors. Your script will need to configure the IESP Python component for the form factor that you are using. To use the IESP hardware in your script, you will need to specify the form factor via lines like the following:

```
from dftm.iespCore import IESP

iesp = IESP.getInstance(formFactorName)
```

where 'formFactorName' is a variable containing the name of the form factor. The available form factors are represented by Python classes in the files in the "dftm/iespFormFactors" folder. To see the names of the available form factors, put a line like the following in your script:

```
print IESP.getAvailableFormFactors()
```

A form factor can have one or more "sub-parts". For example, the "DV1600" form factor has sub-parts named "moduleA" and "moduleB". Your script needs to specify which subparts are to be enabled. You do this via lines like the following:

```
iesp.enableSubPart(subPartName)
```

where 'subPartName' is the name of a sub-part, and 'iesp' is the IESP instance that was obtained earlier.

Some IESP hardware supports multiple firmware personalities. If you are using a form factor that supports multiple personalities, your script needs to specify which of these firmware personalities to use. You do this via a line like the following:

```
iesp.setDefaultFirmwarePersonality(personalityName)
```

where 'personalityName' is the name of a firmware personality, and 'iesp' is the IESP instance that was obtained earlier.

For more info on the IESP methods mentioned above, see the HTML documentation in the file "iesp.html".

Connecting to the IESP Hardware

To connect to the IESP hardware from your external script, you need to add a line like the following:

```
connected = iesp.connectViaFtdi()
```

where 'iesp' is the IESP instance that was obtained earlier. The 'connectViaFtdi' method returns a boolean and you should check

this return value in your script to see if the connection was successful.

Some IESP hardware requires some initialization after connection. To do this initialization, your script should have a line like:

```
status = iesp.initHardware()
```

where 'iesp' is the IESP instance that was obtained earlier. The 'initHardware' method returns a boolean and you should check this return value in your script to see if the hardware initialization was successful.

For more info on the IESP methods mentioned above, see the HTML documentation in the file "iesp.html". Note that it is possible to use the other IESP methods documented in that file to communicate directly with the IESP hardware, but it is usually easier to use the higher-level methods provided by the SvtComponent classes.

Creating SvtComponent Instances

The components used by Introspect ESP are implemented via Python modules in the files under the "dftm/components" folder. The base class for all components is SvtComponent. The main sub-classes of SvtComponent are:

SvtRunnableComponent:

- components that have a 'run' method (returns measurement results)

SvtIespComponent:

- components that "talk" to the IESP hardware

SvtRunnableIespComponent:

- components that "talk" to the IESP hardware and have a 'run' method

The available component classes are documented in the file "svt.html" in the "Doc" folder.

To use an SvtComponent sub-class in your script, you first need to initialize the component classes via the following lines:

```
from dftm.svtComponent import SvtComponent

SvtComponent.initComponents(globals())
```

The above call to the 'initComponents' method also imports the names of all the SvtComponent sub-classes into your script's

namespace, so that it is possible to use these classes without any further imports.

To create an instance of an SvtComponent, you would add a call to the component class' constructor (the name of the class) with no arguments. For example, if you want to create an instance of the SvtGlobalClockConfig component (this is almost always needed), you would add a line like the following:

```
globalClockConfig = SvtGlobalClockConfig()
```

All of the SvtComponent sub-classes can be instantiated in a similar way. Creating a component instance does not communicate with the IESP hardware – it just creates a Python object. The component instance that is created has default values for all of its attributes. You can change the attribute values in the usual way – for example:

```
globalClockConfig.dataRate = 3000
```

This just changes the attributes of the Python object. It does not communicate those changes to the IESP hardware. To affect the IESP hardware, you need to call the 'setup' method:

```
globalClockConfig.setup()
```

To create an instance of the SvtTxChannelList component and set different voltage levels between channels 1 - 3, you would add lines like the following:

```
txChannelList1 = SvtTxChannelList()

txChannelList1.channels = [1, 2, 3]

txChannelList1.voltageSwings = [20.0, 100.0, 500.0]

txChannelList1.setup()
```

If you subsequently (later in your script) wanted to change the voltage values, you could add lines like:

```
txChannelList1.voltageSwings = [20.0, 20.0, 20.0]

txChannelList1.update()
```

The 'setup' method communicates all of the component attributes to the IESP hardware. The 'update' method communicates only those attributes which have changed since the last call to 'setup' or 'update', so the 'update' method is more appropriate when you are changing component attribute values after the initial setup.

Some component attributes are references to other component instances. For example, the 'patterns' attribute of the SvtTxChannelList class is a list of SvtPattern instances. So to set these attributes, your script would first need to create instances of

the appropriate class that can be referred to. For example, your script could create an instance of a user-defined pattern and then change the TxChannelList instance to refer to that pattern and one of the built-in patterns via the following lines:

```
pattern1 = SvtUserPattern()

pattern1.dataBinStr = "11100111010101010110111"

txChannelList1.patterns = [PAT_K28_5, pattern1]

txChannelList1.update()
```