Federal Agencies Digital Guidelines Initiative
ADCTester implementation details

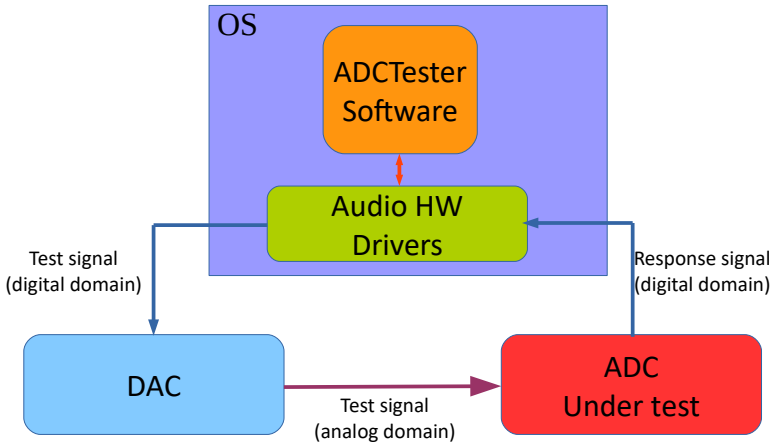
Christian Landone
301/08/2018

1. Overview.

This document details the implementation of an open source software audio test solution for the assessment of the performance of commercial Analogue to Digital Converters.

2. User scenario.

The figure below shows a typical ADC test set-up employing the ADCTester software test and measurement application running on a standard PC.



The Application generates a stimulus signal specific for the type of measurement and plays it through the PC's internal sound devices or an external DAC.
The resulting analogue signal is acquired via the ADC under test and analysed using the metric specific to the test procedure.
The result of the test are subsequently saved to file using an XML format.

Characterisation of the ADC's performance is carried out by multiple test procedures, as specified in the Guideline Document provided by AVPreserve, hence the application's internal signal generator is able to produce a variety of waveforms and the analysis section provides a choice of different measurement algorithms.

Characterisation of the ADC is carried out with minimal intervention from the Operator, apart from physical set-up of the test rig.

The user scenario can be outlined in the following steps.

- The external DAC is connected to one of the host PC's communication ports.

- The ADC under test is connected to one of the host PC's communication ports.

- The Test Application is started.
- On instantiation, the Application's UI shows a list of test procedures that will be performed on the ADC under examination. The parameters for each test procedure are pre-defined, as specified in the ADC performance Guidelines document.

- The Operator selects both the DAC and ADC from a list provided by the Application; the list can is displayed on a modal dialogue window activated by a menu item.

- The chosen hardware set-up is calibrated using the procedure outlined in the pop-up dialogue.

- The test procedures list is reviewed by the Operator and individual procedures are enabled or disabled.

- The Operator initiates the test procedure with a simple button press.

- The application automatically performs each test procedure in the list

- Once the end of the test procedures list is reached, results are saved to file both in machine and human-readable formats.

3. Functional requirements.

A set of functional requirements have been extracted from the user scenario outlined in section 2:

1) The application must be able to identify and enumerate the host platform's native Audio APIs and, for each API, identify and enumerate compatible audio devices connected to the host PC.

2) Discover audio device capabilities and default settings, e.g. supported sample rates.

3) Create and control input and output audio streams for each attached device, at the supported sample rates.

4) Sample-accurate measurements is guaranteed and CPU usage kept at a minimum during the playback and acquisition phase.

5) The application's internal signal generator is able to produce the following type of waveforms:

- Single frequency sine wave with variable frequency, amplitude and duration.
- Stepped frequency (n/Octave) sine waves with configurable amplitude, duration, guarding period and pause between steps.
- Multi-frequency tones with configurable frequencies and amplitudes.

6) The signal generator is able to target individual channels of the selected DAC.

7) For each test procedure, signal acquisition and recording from the ADC under test is initiated without the Operator's intervention.

8) The application's analysis engine is able to manage individual channels of the ADC under test.
9) The application's analysis engine includes FFT capabilities of at least 32K points with a variety of windowing functions including the Kaiser window specified in the Guideline document.
FFT size, overlap and averaging time are mandatory configurable parameters.

10) The measurement procedures in the application's analysis engine must include the test methods outlined in the Guideline document, namely:

- Stepped sine wave frequency response.
- Six point THD+N.
- Single point Signal to Noise.
- Three point Cross-talk.
- Twin tone LF IMD.
- Twin tone IF IMD.
- Single tone Spurious Inharmonic Signal.

11) Once the Operator initiates the performance characterisation procedure, all tests outlined in the Guideline document shall be performed sequentially and tests result document/file shall be generated without further Operator intervention.

12) Simple calibration procedure: the calibration procedure is restricted to a single Operator action prior to commencement of the ADC characterisation procedure.

13) If needed, the analysis engine is able to perform individual measurements directly on LPCM files provided by the Operator.

14) The list of the individual tests that are part of the ADC characterisation procedure is clearly displayed in the application's main UI.

15) The Operator can enable or disable individual tests from the ADC characterisation procedure. Limited parameter change for each test is possible by editing a human-readable XML file.

16) The Application's architecture permit the introduction of additional classes that extend the type of waveforms that can be produced by the application's internal signal generation engine, e.g. random noise (white/pink), chirps, etc.

16) The Application's architecture permit the introduction of additional classes that extend the type of tests can be performed by the application's analysis engine, e.g. Impulse response of the system.

17) Fully modular approach to the characterisation procedure.
Individual tests can be created or removed by an experienced Operator and added to the performance characterisation procedure.
Source type, test signal characteristics and measurement algorithm parameters can be fully configured for each newly created module.

4. Principle of operation.

In order to offer a flexible – and expandable – software tool, the ADC characterisation procedure is specified using an external procedures XML file :

```
−<FADGIProject version="1.0" title="TestProject" date="30/8/2018 at 18:56">
  −<procedures>
    −<test id="0" name="freqresp_left_ch" alias="left channel frequency response: 12 steps/oct" enabled="true">
       <audiorouting signal_ch_idx="0" response_ch_idx="0"/>
       <fileio signalfile="freqresp_left_ch_sig.wav" responsefile="freqresp_left_ch_resp.wav" resultsfile="freqresp_left_ch.xml" testtype="normal" workfolder="C:\Users\chris\Desktop\TestProject"/>
      −<config>
         <parameter name="signal" type="string" value="octsine" units="" alias="signal module" editable="false"/>
         <parameter name="analyser" type="string" value="stepfreq" units="" alias="analysis module" editable="false"/>
         <parameter name="freqstart" type="double" value="20" units="Hz" alias="start frequency" editable="true"/>
         <parameter name="freqstop" type="double" value="20000" units="Hz" alias="stop frequency" editable="true"/>
         <parameter name="octsteps" type="int" value="12" units="" alias="steps per octave" editable="true"/>
         <parameter name="detectionlevel" type="double" value="-70" units="dB" alias="signal detection threshold" editable="true"/>
         <parameter name="inttime" type="double" value="250" units="mS" alias="integration time" editable="true"/>
         <parameter name="transtime" type="double" value="200" units="mS" alias="transient time" editable="true"/>
         <parameter name="bursttime" type="double" value="100" units="mS" alias="burst pause" editable="true"/>
         <parameter name="level" type="double" value="-20" units="dBFS" alias="signal level" editable="true"/>
         <parameter name="outputfreqresponse" type="bool" value="1" units="" alias="output frequency response" editable="true"/>
         <parameter name="maxsigdev" type="double" value="1e-8" units="" alias="signal quality threshold" editable="true"/>
       </config>
      −<performancespecs>
         <spec name="freqrespdev" type="double" value="0.2" units="dB" criterion="lessthan"/>
       </performancespecs>
     </test>
    −<test id="1" name="freqresp_right_ch" alias="right channel frequency response: 12 steps/oct" enabled="true">
       <audiorouting signal_ch_idx="1" response_ch_idx="1"/>
       <fileio signalfile="freqresp_right_ch_sig.wav" responsefile="freqresp_right_ch_resp.wav" resultsfile="freqresp_right_ch.xml" testtype="normal" workfolder="C:\Users\chris\Desktop\TestProject"/>
      −<config>
         <parameter name="signal" type="string" value="octsine" units="" alias="signal module" editable="false"/>
         <parameter name="analyser" type="string" value="stepfreq" units="" alias="analysis module" editable="false"/>
         <parameter name="freqstart" type="double" value="20" units="Hz" alias="start frequency" editable="true"/>
         <parameter name="freqstop" type="double" value="20000" units="Hz" alias="stop frequency" editable="true"/>
         <parameter name="octsteps" type="int" value="12" units="" alias="steps per octave" editable="true"/>
         <parameter name="detectionlevel" type="double" value="-70" units="dB" alias="signal detection threshold" editable="true"/>
         <parameter name="inttime" type="double" value="250" units="mS" alias="integration time" editable="true"/>
         <parameter name="transtime" type="double" value="200" units="mS" alias="transient time" editable="true"/>
         <parameter name="bursttime" type="double" value="100" units="mS" alias="burst pause" editable="true"/>
         <parameter name="level" type="double" value="-20" units="dBFS" alias="signal level" editable="true"/>
         <parameter name="outputfreqresponse" type="bool" value="1" units="" alias="output frequency response" editable="true"/>
         <parameter name="maxsigdev" type="double" value="1e-8" units="" alias="signal quality threshold" editable="true"/>
       </config>
      −<performancespecs>
         <spec name="freqrespdev" type="double" value="0.2" units="dB" criterion="lessthan"/>
```

The procedures file is uniquely associated to a specific “project”: it is therefore generated for each new procedure created by the end user.
The top node contains general information such as the project’s name and creation date.
Additional information such as the operator’s name, type of DAC and ADC used during the procedure and any remarks entered by the operator are also contained in an XML node:

```
−<projectinfo>
    <info name="opsname" value="ChrisL"/>
    <info name="dacname" value="RME ADI 2Pro"/>
    <info name="adcname" value="RME ADI 2Pro"/>
  </projectinfo>
  <remarks content="Test procedure for RME ADI 2 Pro"/>
```

The procedures file defines individual tests, and their appropriate operational parameters, that must be performed sequentially in order to fully verify the compliance of the ADC under investigation.

The top node of each test procedure contains the following information:
The following properties are defined for each *”test”* node:

1) id
The test identification number. This must be incremental.

2) name
The test short-hand name.

3) alias
A human-readable brief description of the test procedure.

4) enabled
Specifies whether the specific test procedure is enabled.: if false, the test will be omitted from the procedure. This parameter can be changed by the user from the main UI.

The "audiorouting" child node specifies the channel routing used by the application for the test procedure:

1) signal_ch_idx
Channel index of the DAC hardware used in the test.

2) response_ch_idx
Channel index of the ADC hardware used in the test.

These parameters can be changed by the user from the main UI.

The "fileio" child node specifies the names of the files generated during each test procedure.

1) signalfile
Specifies the name of the audio file containing the stimulus signal.

2) responsefile
Specifies the name of the audio file containing the audio stream recorded from the ADC under *test.*

3) resultsfile
Specifies the name of the xml file containing all the measured performance metrics and pass/fail result.

4) testtype
Specifies whether a full test will be performed or if the analysis will be carried out on an already existing response file.

These parameters can be changed by the user from the main UI.

The "config" child node enumerates individual operational parameter for each performance test. A number of parameters are common to all test, in particular:

1) parameter name = "signal"
Specifies the type of stimulus signal used by the test procedure (e.g. sine wave, twin tone, etc)

2) parameter name = "analyser"
Specifies which analysis module will be used for the test.

3) parameter name = "inttime"
Specifies the minimum length in mS of the stimulus signal.

4) parameter name = "transtime"
Specifies a "guarding period" in mS that will be added at the beginning and end of the stimulus signal.
This period will be discarded by the analysis module in order to avoid the effect of sudden transients

5) parameter name = "bursttime"
If the test procedure requires multiple burst signals, this parameter specifies the pause period in mS between bursts.

6) parameter name = "outputfreqresponse"
Whenever the specific test generates a frequency plot during the performance metrics evaluation, if this parameter is enabled, the plot will be saved in the results file.

7) parameter name = "maxsigdev"
Specifies a threshold used for the evaluation of the quality of the response signal acquired during the test procedure.
This is used to detect playback/recording glitches and external noises that may occur during the test due to hardware issues or faulty cables.

Refer to the individual tests' documentation for an explanation of the other parameters.

The "performancespecs" node lists which performance metric will be used by the application to return a pass or fail result at the end of the specific test.
In the case of test # 0 in the above figure we have:

1) name = "freqrespdev"
This corresponds to the difference between the maximum and minimum values of the ADC's measured frequency response.

2) type = "double"
Variable type (either double or string), currently this specifier is not used by the application.

3) value = "0.2"
Limit value of *freqrespdev* for pass/fail decision.

4) criterion= "lessthan"
Measured value for *freqrespdev* must be less than *value* for a "pass" result.

Once the test procedure is initiated, the application will cycle through each enabled test described in the XML structure and perform the following action:

- Generate the stimulus signal specific to the test and save it to a LPCM Wav file using the path defined by the *"signalfile" value*.

- Playback the *signalfile* through the DAC specified in the “audio devices setup” UI window while simultaneously record the stream from the ADC. The recorded ADC response will be saved as a LPCM Wav file to the path defined by the *“responsefile”* value.

- Once the playback/recording procedure has ended, the application opens the *responsefile* and performs relevant performance metrics extraction using to the analysis module specified by the *“analyser”* parameter value in the test description properties.

- Test results are then serialised to an XML file to the path defined by the *“resultsfile” value*.
The results files typically have the following format:

```
− <FADGIResults title="right channel frequency response: 12 steps/oct" channelindex="1">
  − <dataset id="0">
    − <testmetrics>
        <parameter name="maxFreq" value="251.953" units="Hz"/>
        <parameter name="maxFreqLevel" value="-20.008" units="dB"/>
        <parameter name="minFreq" value="16148.4" units="Hz"/>
        <parameter name="minFreqLevel" value="-20.0841" units="dB"/>
        <parameter name="freqrespdev" value="0.0760721" units="dB"/>
      </testmetrics>
    + <freqresponse></freqresponse>
    </dataset>
  − <performancespecs>
      <spec name="freqrespdev" type="double" value="0.2" units="dB" criterion="lessthan"/>
    </performancespecs>
    <testoutcome value="pass"/>
  </FADGIResults>
```

The root node contains information regarding the test name, tested audio channel and measured performance parameters.
The above example is the result of a 12 steps/ocatve frequency response measurement carried out on the right channel of the ADCC under evaluation.

The *“testmetrics”* node displays the following measurement parameters:

parameter name = “maxFreq”
Frequency at which the maximum peak level has been measured.

parameter name = “maxFreqLevel”
Value in dB of the measured the maximum peak level.

parameter name = “minFreq”
Frequency at which the minimum peak level has been measured.

parameter name = “minFreqLevel”
Value in dB of the measured the minimum peak level.

parameter name = “freqRespDev”
Difference between the maximum and minimum measured peak levels in the entire frequency response.

The *“freqresponse”* node contains data points describing the frequency response measured during the procedure. If this data is not required, it can be omitted by setting the parameter *“outputfreqresponse”* to false in the test description node.

The response is saved as children nodes with the following format:

```
− <freqresponse>
    <point frequency="35.1563" level="-20.0472"/>
    <point frequency="58.5938" level="-20.0238"/>
    <point frequency="64.4531" level="-20.0202"/>
    <point frequency="82.0313" level="-20.0161"/>
    <point frequency="105.469" level="-20.0149"/>
    <point frequency="111.328" level="-20.0182"/>
    <point frequency="140.625" level="-20.0117"/>
    <point frequency="146.484" level="-20.0161"/>
    <point frequency="169.922" level="-20.0138"/>
    <point frequency="175.781" level="-20.0099"/>
    <point frequency="187.5" level="-20.0118"/>
```

Where each point is described by its frequency in Hz and amplitude in dB.

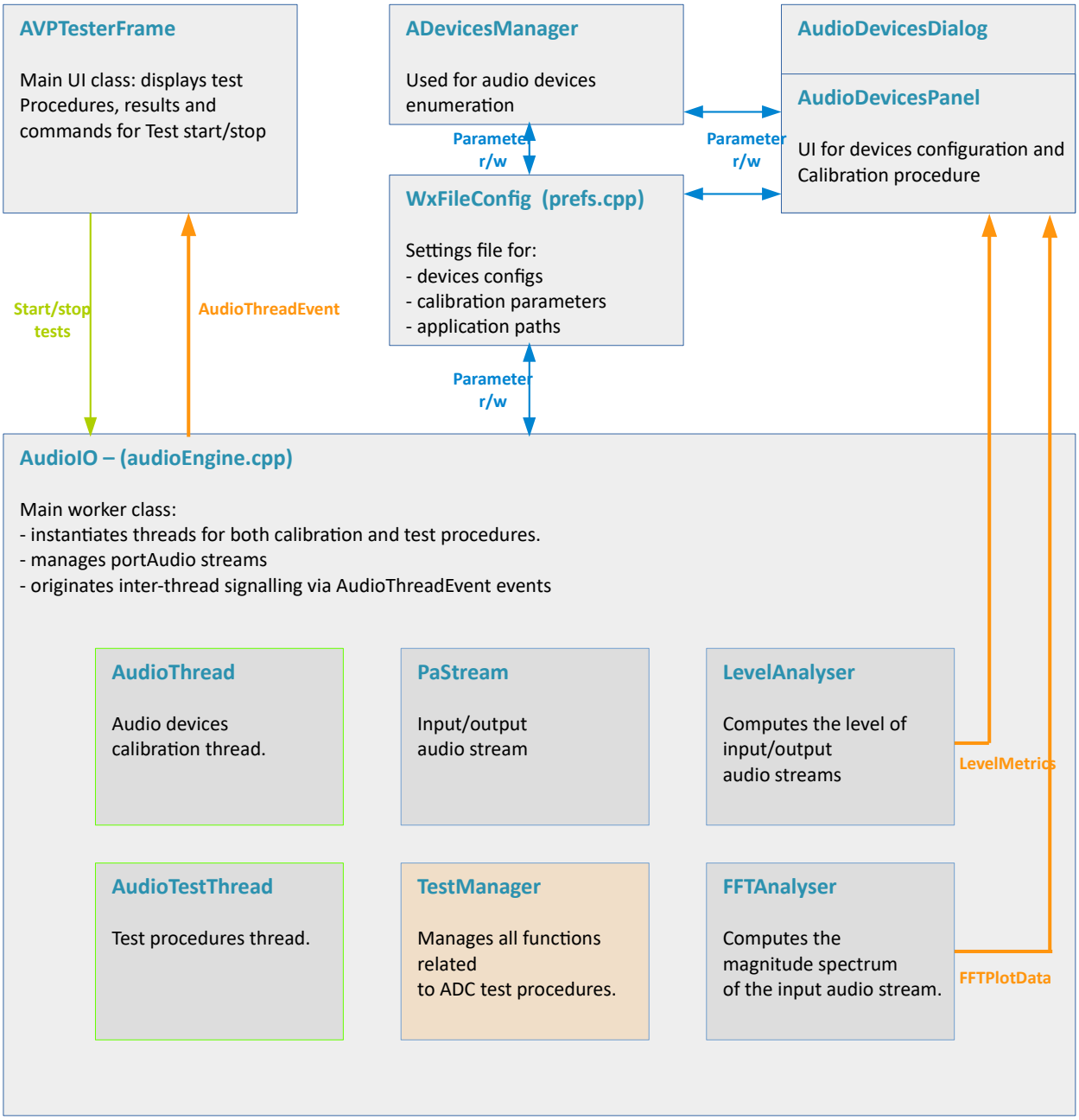
The node named *”performancespecs”* is copied for reference from the test description node in the procedures file.

The node named *“testoutcome”* simply indicates the outcome of the test, either *pass, fail* or *retest*.

5. Internal implementation.

The following sections describe the internal architecture of the ADCTester software application and the most relevant C++ classes.

5.1 Top level class view.



5.2 TestManager class.

This class is “in charge” of the majority of the application’s functionalities as it parses the test procedures file and instantiates individual signal generators and analysis modules.

It provides the following public methods:

```
bool OpenProject( wxString path );
```
Opens a procedures file specified by the user

```
bool SaveProject( wxString path = wxEmptyString);
```
Saves the current procedures structure to a specific path

```
void SetTestXml(wxXmlNode* tNode, wxString ProjectBasePath);
```
Sets the XML structure describing the test procedures

```
wxXmlNode* GetTestsXml() { return mTestsNode; }
```
Returns the XML structure describing the test procedures

```
std::vector<TestDescriptor> GetTestDescriptors() { return mDescriptors; }
```
Returns a vector containing string-only descriptions of the currently loaded test procedures. Used by the UI for display purposes.
Each element in the vector is a structure with the following format:

```
typedef struct TestDescriptor {
        wxString ID;
        wxString name;
        wxString alias;
        wxString enabled;
}TestDescriptor;
```

```
void EnableTest(wxString testID, bool enabled);
```
Enables/disable a single test with index testID

```
void SetTestParameter(int testIndex, wxString paramName, wxString paramValue);
```
sets the value of a specific test parameter

```
std::vector<TestParameter>  GetTestParameters(wxString testID);
```
Returns a vector containing string-only descriptions of the test with index testID. Used by the UI for display purposes.
Each element in the vector is a structure with the following format:

```
typedef struct TestParameter {
        wxString name;
        wxString type;
        wxString alias;
        wxString units;
        wxString value;
        wxString editable;
}TestParameter;
```

```
TestFileIOInfo GetTestFileIOInfo(wxString testID);
```
Returns a vector containing string-only descriptions of the file IO settings for test with index testID.
Each element in the vector is a structure with the following format:

```
typedef struct TestFileIOInfo{
        wxString testType;
        wxString signalFileName;
        wxString responseFileName;
        wxString resultsFileName;
        wxString workFolder;
}TestFileIOInfo;
```

TestAudioIOInfo GetTestAudioIOInfo(wxString testID);
Returns a vector containing string-only descriptions of the audio IO settings for test with identifier testID.

TestAudioIOInfo GetTestAudioIOInfo(int testIdx);
Returns a vector containing string-only descriptions of the audio IO settings for test with index testIdx.
Each element in the vector is a structure with the following format:

```
typedef struct TestAudioIOInfo {
        wxString signalChIdx;
        int signalChIdxNum;
        wxString responseChIdx;
        int responseChIdxNum;
}TestAudioIOInfo;
```

int GetNumberOfTest() { return mNumberOfTests; }
Returns the number of tests specified in the procedures file.

bool IsTestEnabled(int testIndex);
Test whether the test with index testnIdex is enabled.

int GetTestType(int testIndex);
Returns the type of test (full, file analysis only, etc)

int GenerateSignalFile(int testIndex, double sampleRate, int Channels, wxString&);
Generates the stimulus signal described in the test with id *testIndex* and saves it to file. *SampleRate* and *Channels* are parameters specific to the procedure set up that are set by the user in the “audio devices setup” UI window.

wxString GetSignalFilePath(int testIdx);
Returns the complete path of the Wav file containing the stimulus signal.

wxString GetResponseFilePath(int testIdx);
Returns the complete path of the Wav file containing the recorded response signal.

wxString AnalyseResponse(int testIndex);
Instantiates the analysis module specified in the test descriptor and calculates performance metrics using the response file.

wxString GetParameterValue(int testIndex, wxString parameterName);
Helper method: returns the value of a parameter belonging to the test with ID testIndex;

wxString GetParameterAlias(int testIndex, wxString parameterName);
Helper method: returns the alias of a parameter belonging to the test with ID testIndex;

wxXmlNode* GetParameterNode(int testIndex, wxString parameterName);
Returns the parameter xml description belonging to the test with ID testIndex;

void SetTestType(wxString testID, wxString testType);
Sets the type of test to be performed (normal or offline)

void SetTestResponseFileName(wxString testID, wxString fileName);
Sets the name of the Wav file containing the recorded response signal.

void SetTestSignalChannel(wxString testID, wxString chIdx);
Sets the index of the playback channel used in the test.

void SetTestResponseChannel(wxString testID, wxString chIdx);
Sets the index of the recording channel used in the test.

5.3 Devices calibration thread.

When the user activates the manual calibration procedure for the chosen DAC/ADC combination, the AudioIO object starts a thread named AudioThread, with entry point *doIODevicesCalibration().* Within this method, the following actions take place:

Pa_Initialize()
The PortAudio Library, allowing access to the host audio APIs, is initialised:

```
mInConfig = ADevicesManager::Instance()->getCurrentInputConfig();
mOutConfig = ADevicesManager::Instance()→getCurrentOutputConfig();
```

Current settings from the chosen DAC/ADC setup are loaded from the Preferences object.

mInputLevelMetric->Initialise(...)
Audio level analyser for the input audio stream is instantiated.

mFFTrta→initialiseFFT(...)
mFFTrta is a FFTAnalyser object.
The spectrum analyser for the input streams is initialised

OpenInputDevice(...)
Selected ADC device is opened, using the chosen sample rate and number of channels.

OpenOutputDevice(...)
Selected DAC device is opened, using the chosen sample rate and number of channels.

```
mThreadWrite = new AudioThreadWrite;
if (mThreadWrite->Run() != wxTHREAD_NO_ERROR)
{
        delete mThreadWrite;
        mThreadWrite = NULL;
}
```

Start the playback thread

```
while (mPlaybackDone == false)
```

Main loop for the calibration procedure. Carries on until the procedure is manually stopped by the user.
Within the loop, the following process occurs:

Test Signal generation
A 997Hz sine wave is generated by the playback thread and the signal is multiplied by the gain factor set in the UI.

Pa_ReadStream(...);
The signal is acquired from the input audio device (DUT).

mInputLevelMetric->process(...);
Input audio level is computed.

mFFTrta->doRTA(...);
Spectrum of input audio level is computed.

Loop is terminated, objects are deleted and Audio I/O is deinitialsed.

CloseInputDevice();

CloseOutputDevice():

Pa_Terminate();

5.4 Test procedures thread.

When the user activates the ADC test procedure fro the main UI ,the AudioIO object instantiates a thread named AudioTestThread, with entry point *doADCTest().*
Within this method, the following actions take place:

Pa_Initialize()
The PortAudio Library, allowing access to the host audio APIs, is initialised:

GetCurrentIOConfiguration(...)
Current settings from the chosen DAC/ADC setup are loaded from the Preferences object.

mTestManager→GetNumberOfTest(...)
mTestManager is a mTestManager object.
This call find out the number of tests procedures specified for the characterisation of the ADC.

for (testIndex = 0; testIndex < noTests; testIndex++)
For each of the test procedures perform the following:

> if(mTestManager→IsTestEnabled(testIndex))
> Test if the procedure is enabled
>
> mTestManager→GenerateSignalFile(...);
> Generate an appropriate signal file for the test procedure
>
> PlaybackAcquire(...);
> Playback the signal file whilst recording the audio stream from the ADC. The internal structure of this method is similar to the *doIODevicesCalibration* method described in the previous section.
>
> mTestManager→AnalyseResponse(...);
> Once playback and recording is terminated, open the ADC response file and perform the appropriate analysis.

All tests specified in the characterisation have been carried out.
Loop is terminated, objects are deleted and Audio I/O is deinitialsed.
Pa_Terminate();

5.5 Test procedures implementation.

In this section we present the implementation of the individual tests contained in the current characterisation procedure.
As mentioned in a previous section, the ADCTester application parses the XML node pertaining to the individual test procedure and then proceeds to generate a test file as a 32 bit floating point WAV file.
Once the test file is saved to disk, the application plays it back via the chosen DAC devices and simultaneously records to file the response recorded from the ADC under test.
At the end of this process, the appropriate analysis module is instantiated and performs extraction of the performance metrics specific to the test.

5.5.1 Signal generator modules.

During the signal generation procedure, the TestManager reads the XML specification for the required test and dynamically instantiates the module defined by the parameter named *"signal"*:

<parameter name="signal" type="string" value="octsine" units="" alias="octave stepped sine" editable="false"/>

In this case the signal generator module will create a file containing multiple sine waves with fractional octave spaces.

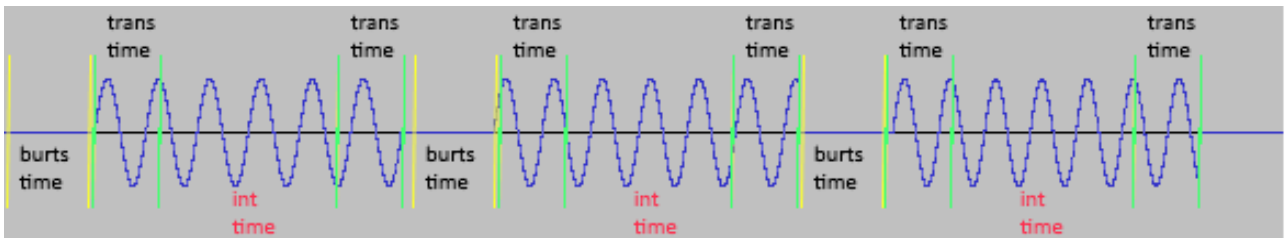
Apart from the test's specific parameters, such as level and frequencies, a number of common parameters are used by each module to create the appropriate test signal:

<parameter name="bursttime" type="double" value="100" units="mS" ..... />

<parameter name="transtime" type="double" value="50" units="mS" ..... />

<parameter name="inttime" type="double" value="250" units="mS" ..... />

The digram below shows the meaning of these parameters:



In practice *inttime* is the minimum signal length required to perform the analysis on the signal, whereas *bursttime* is a “silent” period used by the analysis stages to locate the exact beginning of the signal.

A guarding period, *transtime*, is also inserted in order to eliminate switch on/off transients from the calculations.

Tests that work primarily in the frequency domain usually specify the analysis period in terms of the length of the required FFT multiplied by the number of desired averages.
In these cases the *inttime* parameter is derived directly using the following entries in the XML descriptor:

<parameter name="fftlength" type="double" value="32768" units="" alias="fft size" … />

<parameter name="fftnoavg" type="double" value="16" units="" alias="number of fft averages"../>

In order to enable the use of input and output sound devices operating with different sample rates, the signal generator modules need both values to be specified on instantiation, e.g:

```
OctaveToneGenerator(double pbSampleRate, double recSampleRate, int channels);
```

5.5.2 Analyser modules.

During the response analysis procedure, the TestManager reads the XML specification for the required test and dynamically instantiates the module defined by the parameter named *“analyser”*:

<parameter name="analyser" type="string" value="thdn" units="" alias="total harmonic distortion plus noise" editable="false"/>

Apart from test-specific specifications, all analysis modules require a set of parameters that pertain to the signal’s time domain structure, in particular:

The target audio channel for the performance analysis.
<parameter name="chidx" type="int" value="0" units="" alias="test channel" editable="true"/>

The length of the transients guarding period:
<parameter name="transtime" type="double" value="250" units="mS" alias="transient time" editable="true"/>

The threshold level for the detection of a valid signal:
<parameter name="detectionlevel" type="double" value="-70" units="dB" alias="signal detection threshold" ../>

An onset detector is used in all analysis modules to locate the beginning of a valid signal, and the value of *transtime* is added to identify the actual analysis start point

The sample rate value is directly extracted from the Wav file metadata.

Due to a large functional commonality that exists between analysis modules, these are typically derived from a base class called *FadgiAnalyser.*

```
class FADGIAnalyser
{
public:
	FADGIAnalyser();
	virtual ~FADGIAnalyser();

	//mandatory function implemented by all derived analysis modules
	virtual int analyseSignal(wxXmlNode* testDescriptionNode)=0;


	////////////////////////////////////////////////////////
	// common member functions
	////////////////////////////////////////////////////////

	//extract analysis parameters from test xml description
	void setParameters(wxXmlNode* testDescriptionNode);

	//open .WAV file of recorded DUT response
	SNDFILE* openResponseFile();

	//close .WAV file of recorded DUT response
	void closeResponseFile();

	//analyse response and find beginnning of signal
	std::vector<size_t> getOnsets(SNDFILE* afile, int channelIndex, bool debug = false);


	//check pass or fail test condition
	bool checkTestSpecs(wxXmlNode* resultsNode);

	//serialise report to file
	bool writeResultsToFile(wxXmlNode* resultsNode);


	////////////////////////////////////////////////////////
	// helper functions
	////////////////////////////////////////////////////////

	//find max level in frequency range
	FreqPoint findPeakInRange(float startFreq, float endFreq, std::vector<FreqPoint> &frequencyResponse);

	//find min level in frequency range
	FreqPoint findMinInRange(float startFreq, float endFreq, std::vector<FreqPoint> &frequencyResponse);

	//get value of test configuration parameter
	wxString getTestParameterStringValue(wxString paramName, wxXmlNode* testParamsNode);

	//gets value of nammed parameter
	double getTestParameterValue(wxString paramName, wxXmlNode* testParamsNode);

	//get value of measured parameter from report
	double getResultValue(wxString paramName, wxXmlNode* resultsNode);

	//get pass/fail specification from guidelines
	double getSpecValue(wxString paramName, wxXmlNode* specsNode);

	//acquired response signal quality pass/fail check
	bool checkSignalQuality();
...
};
```

The object *FreqPoint* is a structure employed by all analyser classes to characterise an element in the frequency domain and a vector of these elements is the standard representation of a frequency response or FFT output:

```
typedef struct FreqPoint {
	int binNumber;
	double frequency;
	double peakValueLin;
	double peakValueLog;
}FreqPoint;
```

FADGIAnalyser::AnalyseSignal() is mandatory for all analyser units and its implementation typically follows the scheme:

```
int
DummyAnalyser::analyseSignal(wxXmlNode* testDescriptionNode)
{
	int result = TestErrorUnknown;

	//extract core common parameters, i.e. file paths, detection thresholds, guarding	periods,etc
	setParameters(testDescriptionNode);

	//recorded response file
	SNDFILE* mRespFile = openResponseFile();

	if (mRespFile)
	{
		//find valid audiosegments in file
		std::vector<size_t> onsets = getOnsets(mRespFile, mSelectedChannel, false);

		if (onsets.size() > 0)
		{
			//analyse detected audio segments
			analyseSegments(mRespFile, onsets);

			//check integrity of signal
			mSigQualityOK = checkSignalQuality();

			//pass, fail or retest outcome
			result = buildReport();
		}
		else
		{
			result = TestErrorRespSignal;
		}

		closeResponseFile();
	}
	else
	{
		result = TestErrorRespFile;
	}
	return result;
}
```

The “AnalyseSignal” method can return the following values:

```
enum FADGIAnalysisOutcome {

TestErrorUnknown = -10,
//unknown test outcome (for debug purposes only)

TestErrorRespFile,
//response audio file not found

TestErrorRespSignal,
//no valid audio segment has been detected in the response file

TestSigQualityFail,
//the response signal is corrupted

TestFail = 0,
//response of the device under test is not compliant with the published specifications

TestPass
//response of the device under test is compliant with the published specifications
};
```

If the signal quality check fails, ADCTest will alert the end user that the result is not safe and, therefore, the test should be repeated.
The quality check is performed by verifying that the total power of the signal remains constant throughout the measurement procedure.
If a sudden change occurs, this is likely caused by a signal drop due to a transient audio driver fault during the playback and acquisition process or by noise injected in the test set-up.

5.5.3 Frequency response test unit.

Signal generation module:

```
<parameter name="signal" type="string" value="octsine" units="" alias="octave stepped sine" editable="false"/>
```

The TestManager instantiate the class: *OctaveToneGenerator* declared in
./SigGen/*OctaveToneGenerator.h*

Signal generator’s parameters:

Start frequency:
```
<parameter name="freqstart" type="double" value="18" units="Hz" alias="start frequency" editable="true"/>
```

Stop frequency:
```
<parameter name="freqstop" type="double" value="21000" units="Hz" alias="stop frequency" editable="true"/>
```

Number of frequency steps per octave:
```
<parameter name="octsteps" type="int" value="12" units="" alias="steps per octave" editable="true"/>
```
Duration for each step:
```
<parameter name="inttime" type="double" value="250" units="mS" alias="integration time" editable="true"/>
```

Analysis module:

<parameter name="analyser" type="string" value="stepfreq" units="" alias="stepped frequency response" editable="false"/>

The TestManager instantiate the class: *StepsFrequencyResponse* declared in ./Analysers/*StepsFrequencyResponse .h*

Analyser's operational parameters:

Signal detection level:
<parameter name="detectionlevel" type="double" value="-70" units="dB" alias="signal detection threshold" editable="true"/>

Analysis procedure:

The stimulus signal consists in a sequence of discrete tones at frequencies corresponding to 1/12th octave spacing.

The LPCM Wave file containing the recorded response from the ADC under test is opened and the signal is analysed using the FADGIAnalyser::getOnsets(…) function.
A vector containing the position, in samples, of each discrete tone is returned by this method.

Each segment of the signal is processed in the *analyseSegment(…)* function and the following operations are performed:

- The transient guarding period is discarded from each segment.
- The dominant frequency and peak level are calculated on each segment. The process generates a FreqPoint element for each analysed segment.

- The frequency response (mFrequencyResponse) of the ADC is a vector of the above elements.

- The performance metric of the test mFreqRespDev is given by the difference between the maximum and minimum levels in the computed frequency response.

```
mMaxFreq = findPeakInRange(mStartFreq, mStopFreq, mFrequencyResponse);
mMinFreq = findMinInRange(mStartFreq, mStopFreq, mFrequencyResponse);
double frDv = (double)(mMaxFreq.peakValueLin / mMinFreq.peakValueLin);
mFreqRespDev =fabs(20 * log10(frDv));
```

The search range is defined by values obtained from the unit test's operational parameters:

```
double mStartFreq = getTestParameterValue(wxT("freqstart"), mParamsNode);
double mStopFreq = getTestParameterValue(wxT("freqstaop"), mParamsNode);
```

- The pass/fail outcome is obtained by comparing the value of mFreqRespDev with the published reference.

5.5.4 Total harmonic distortion + noise test unit.

Signal generation module:

<parameter name="signal" type="string" value="singlesine" units="" alias="single sine tone" editable="false"/>

The TestManager instantiate the class: *SingleSineToneGenerator* declared in ./SigGen/*SingleSineToneGenerator.h*

Signal generator's parameters:

Stimulus frequency:
<parameter name="tonefreq" type="double" value="41" units="Hz" alias="signal frequency" editable="true"/>

Stimulus level:
<parameter name="tonelevel" type="double" value="-1" units="dB" alias="signal level" editable="true"/>

Signal duration:
<parameter name="fftlength" type="double" value="32768" units="" alias="fft size" editable="true"/>
<parameter name="fftnoavg" type="double" value="16" units="" alias="number of fft averages" editable="true"/>

Analysis module:

<parameter name="analyser" type="string" value="thdn" units="" alias="total harmonic distortion plus noise" editable="false"/>

The TestManager instantiate the class: *THDNoise* declared in ./Analysers/THDNoise.*h*

Analyser's operational parameters:

Signal detection level:
<parameter name="detectionlevel" type="double" value="-70" units="dB" alias="signal detection threshold" editable="true"/>

FFT size:
<parameter name="fftlength" type="double" value="32768" units="" alias="fft size" editable="true"/>

Number of averages:
<parameter name="fftnoavg" type="double" value="16" units="" alias="number of fft averages" editable="true"/>

Type of averaging function:
<parameter name="fftavgtype" type="string" value="linear" units="" alias="averaging type" editable="true"/>

Bandwidth of notch used for exclusion of fundamental tonal component:
<parameter name="notchbw" type="double" value="200" units="hz" alias="fundamental notch bandwidth" editable="true"/>

Search range for each harmonic component:
<parameter name="harmsearchbw" type="double" value="20" units="hz" alias="search range of harmonics for THD calculations" editable="true"/>

Low-end exclusion zone for calculations:

<parameter name="lowerlimit" type="double" value="20" units="hz" alias="lowest frequency used in calculations" editable="true"/>

Analysis procedure:

The stimulus signal consists in a discrete tone at frequencies and levels specified by the guidelines.

The LPCM Wave file containing the recorded response from the ADC under test is opened and the signal is analysed using the FADGIAnalyser::getOnsets(…) function.
A vector containing the position, in samples, of the recorded signal is returned by this method.
In this case, only one segment will be detected.

The active segment of the signal is processed in the *analyseSegment(…)* function and the following operations are performed:

- The transient guarding period is discarded from the segment.

- The maximum peak level of the response signal is computed (mMaxSigValue).

- The segment is divided into the number of sub-segments specified by the *fftnoavg* parameter*;* Each segment has length specified by the parameter *fftlength* ;

- Each sub-segment is windowed using a Kaiser7 function and its magnitude frequency is computed from using an FFT algorithm.

- Linear averaging of the computed Magnitude FFT is performed.

- The vector mFrequencyResponse of FreqPoint elements containing the averaged magnitude response of the ADC in the frequency domain is obtained.

- The total signal power of the signal is computed by adding its magnitude squared elements (mTotalSignaPower).

- The fundamental frequency element mSigBin of the stimulus signal *Fc* is detected from the response spectrum.

- The noise and harmonic distortion spectrum is obtained by excluding frequencies below the value specified by the parameter *lowerlimit* and a notch around the fundamental frequency specified by the parameter *notchbw .*
The notch exclusion zone is defined as *(Fc – notchbw/2 : Fc + notchbw/2)*

- The power of the noise and harmonic distortion spectrum is obtained by adding it magnitude squared elements (mNoiseAndHDPower).

- The percentile and log values for the THD+N are calculated using, respectively :

```
mTHDpN_Pc = 100 * sqrt((mNoiseAndHDPower) / (mTotalSignaPower));
mTHDpN_Log = 20 * log10(mTHDpN_Pc / 100);
```

- The levels of up to the 6th order harmonics are computed.
For each nth order harmonic, a search is made in the response spectrum for the highest value in the range: *(n*Fc – harmsearchbw : n*Fc + harmsearchbw)*
The power of the harmonic distortion spectrum is obtained by adding the magnitude squared values of each detected harmonic (harmPower)

```
float sigPower = mSigBin.peakValueLin*mSigBin.peakValueLin;
float harmPower = 0;
for (size_t hIdx = 2; hIdx < 6; hIdx++) {
        float harmFreq = hIdx*mSigBin.frequency;
        float startFreq = harmFreq - mHarmonicsSearchBandwidth/2;
        float endFreq = harmFreq + mHarmonicsSearchBandwidth / 2;
        FreqPoint hh = findPeakInRange(startFreq, endFreq, mFrequencyResponse);
        harmPower += (hh.peakValueLin*hh.peakValueLin);
}
```

- The percentile and log values for the Total Harmonic Distortion are calculated using, respectively:

```
mTHD_Pc = 100*sqrt(harmPower/sigPower);
mTHD_Log = 20 * log10(mTHD_Pc / 100);
```

where sigPower is the power of the fundamental component of the response calculated as:
mSigBin.peakValueLin*mSigBin.peakValueLin

- The log value for the Dynamic Range is calculated using:

```
mSNR_Log = 20 * log10((mTHDpN_Pc/100) - (mTHD_Pc/100)) - fabs(mSigBin.peakValueLog);
```

where mSigBin.peakValueLog is the peak log10 value of the fundamental component of the response.

- The pass/fail outcome is obtained by comparing the value of mTHDpN_Pc with the published reference.

5.5.5 Dynamic range test unit.

Signal generation module:

```
<parameter name="signal" type="string" value="singlesine" units="" alias="single sine tone" editable="false"/>
```

The TestManager instantiate the class: *SingleSineToneGenerator* declared in ./SigGen/*SingleSineToneGenerator.h*

Signal generator's parameters:

Stimulus frequency:
<parameter name="tonefreq" type="double" value="997" units="Hz" alias="signal frequency" editable="true"/>

Stimulus level:
<parameter name="tonelevel" type="double" value="-40" units="dB" alias="signal level" editable="true"/>

Signal duration:
<parameter name="fftlength" type="double" value="32768" units="" alias="fft size" editable="true"/>
<parameter name="fftnoavg" type="double" value="16" units="" alias="number of fft averages" editable="true"/>

Analysis module:

<parameter name="analyser" type="string" value="thdn" units="" alias="total harmonic distortion plus noise" editable="false"/>

The TestManager instantiate the class: *THDNoise* declared in ./Analysers/THDNoise.*h*

Analyser's operational parameters:

Signal detection level:
<parameter name="detectionlevel" type="double" value="-70" units="dB" alias="signal detection threshold" editable="true"/>

FFT size:
<parameter name="fftlength" type="double" value="32768" units="" alias="fft size" editable="true"/>

Number of averages:
<parameter name="fftnoavg" type="double" value="16" units="" alias="number of fft averages" editable="true"/>

Type of averaging function:
<parameter name="fftavgtype" type="string" value="exponential" units="" alias="averaging type" editable="true"/>

Bandwidth of notch used for exclusion of fundamental tonal component:
<parameter name="notchbw" type="double" value="200" units="hz" alias="fundamental notch bandwidth" editable="true"/>

Search range for each harmonic component:
<parameter name="harmsearchbw" type="double" value="20" units="hz" alias="search range of harmonics for THD calculations" editable="true"/>

Low-end exclusion zone for calculations:
<parameter name="lowerlimit" type="double" value="20" units="hz" alias="lowest frequency used in calculations" editable="true"/>

Analysis procedure:

Same procedure as THD+N test unit with the notable differences:

- Averaging of the spectrum magnitude is exponential rather than linear;

- The pass/fail outcome is obtained by comparing the value of mSNR_Log with the published reference.

5.5.6 Cross-talk test unit.

Signal generation module:

<parameter name="signal" type="string" value="singlesine" units="" alias="single sine tone" editable="false"/>

The TestManager instantiate the class: *SingleSineToneGenerator* declared in ./SigGen/*SingleSineToneGenerator.h*

Signal generator's parameters:

Stimulus frequency:
<parameter name="tonefreq" type="double" value="20" units="Hz" alias="signal frequency" editable="true"/>

Stimulus level:
<parameter name="tonelevel" type="double" value="-1" units="dB" alias="signal level" editable="true"/>

Signal duration:
<parameter name="fftlength" type="double" value="32768" units="" alias="fft size" editable="true"/>
<parameter name="fftnoavg" type="double" value="16" units="" alias="number of fft averages" editable="true"/>

Analysis module:

<parameter name="analyser" type="string" value="xtalk" units="" alias="xtalk analyser class" editable="false"/>

The TestManager instantiate the class: *Crosstalk* declared in ./Analysers/Crosstalk.*h*

Analyser's operational parameters:

Signal detection level:
<parameter name="detectionlevel" type="double" value="-150" units="dB" alias="signal detection threshold" editable="true"/>

FFT size:
<parameter name="fftlength" type="double" value="32768" units="" alias="fft size" editable="true"/>

Number of averages:
<parameter name="fftnoavg" type="double" value="16" units="" alias="number of fft averages" editable="true"/>

Type of averaging function:
<parameter name="fftavgtype" type="string" value="exponential" units="" alias="averaging type" editable="true"/>

Analysis procedure:

The stimulus signal consists in a discrete tone at frequencies and levels specified by the guidelines.

The stimulus is applied on an audio channel different from the channel under test, as this procedure is aimed at detecting signal leakages.

The LPCM Wave file containing the recorded response from the ADC under test is opened and the signal is analysed using the FADGIAnalyser::getOnsets(…) function.
A vector containing the position, in samples, of the recorded signal is returned by this method.
In this case, only one segment will be detected.
The active segment of the signal is processed in the *analyseSegment(…)* function and the following operations are performed:

- The transient guarding period is discarded from the segment.

- The frequency and level of the stimulus signal is obtained from the test unit's configuration parameters:

```
mInputSignalLevel = (float)getTestParameterValue(wxT("tonelevel"), mParamsNode);
mInputSignalFrequency = (float)getTestParameterValue(wxT("tonefreq"), mParamsNode);
```

TODO: These values could be obtained from the channel where the stimulus signal is applied, this is preferable, as it yields a more accurate input level estimate.

- The segment is divided into the number of sub-segments specified by the *fftnoavg* parameter*;* Each segment has length specified by the parameter *fftlength* ;

- Each sub-segment is windowed using a Kaiser7 function and its magnitude frequency is computed from using an FFT algorithm.

- Exponential averaging of the computed Magnitude FFT is performed.

- The vector mFrequencyResponse of FreqPoint elements containing the averaged magnitude response of the ADC in the frequency domain is obtained.
- The element with the highest level found in a limited range near the stimulus frequency is detected (FreqPoint hh)

- The value for the crosstalk is calculated as:

```
xTalkValue_Log = hh.peakValueLog + mInputSignalLevel;
```

- The pass/fail outcome is obtained by comparing the value of xTalkValue_Log with the published reference.

5.5.7 Low frequency intermodulation distortion test unit.

Signal generation module:

<parameter name="signal" type="string" value="dualsine" units="" alias="dual tone" editable="false"/>

The TestManager instantiate the class: Dual*SineToneGenerator* declared in ./SigGen/Dual*SineToneGenerator.h*

Signal generator's parameters:

Stimulus 1 frequency:
<parameter name="tonefreq1" type="double" value="41" units="Hz" alias="signal 1 frequency" editable="true"/>

Stimulus 2 frequency:
<parameter name="tonefreq2" type="double" value="7993" units="Hz" alias="signal 2 frequency" ../>

Total signal level:
<parameter name="tonelevel" type="double" value="-1" units="dB" alias="total signal level" editable="true"/>

Level ratio of stimuli:
<parameter name="levelratio" type="double" value="4" units="dB" alias="signals level ratio" editable="true"/>

Signal duration:
<parameter name="fftlength" type="double" value="32768" units="" alias="fft size" editable="true"/>
<parameter name="fftnoavg" type="double" value="16" units="" alias="number of fft averages" editable="true"/>

Analysis module:

<parameter name="analyser" type="string" value="lfimd" units="" alias="high or low frequency imd" .. />

The TestManager instantiate the class: *IMD* declared in ./Analysers/IMD.*h*

Analyser's operational parameters:

Signal detection level:
<parameter name="detectionlevel" type="double" value="-70" units="dB" alias="signal detection threshold" .. />

FFT size:
<parameter name="fftlength" type="double" value="32768" units="" alias="fft size" editable="true"/>

Number of averages:
<parameter name="fftnoavg" type="double" value="16" units="" alias="number of fft averages" editable="true"/>

Type of averaging function:
<parameter name="fftavgtype" type="string" value="linear" units="" alias="averaging type" editable="true"/>

Intermodulation distortion calculation method:
<parameter name="imdtype" type="string" value="smpte" units="" alias="imd method" editable="true"/>

Analysis procedure:

The stimulus signal consists in a twin tone signal at frequencies and levels specified by the guidelines.

The LPCM Wave file containing the recorded response from the ADC under test is opened and the signal is analysed using the FADGIAnalyser::getOnsets(…) function.
A vector containing the position, in samples, of the recorded signal is returned by this method.
In this case, only one segment will be detected.

The active segment of the signal is processed in the *analyseSegment(…)* function and the following operations are performed:

- The transient guarding period is discarded from the segment.

- The stimulus signal's frequencies are obtained from the test uint's operational parameters:

```
mInputSignalFrequencyLow = (float)getTestParameterValue(wxT("tonefreq1"),      mParamsNode);
mInputSignalFrequencyHigh = (float)getTestParameterValue(wxT("tonefreq2"),      mParamsNode);
```

- The segment is divided into the number of sub-segments specified by the *fftnoavg* parameter*;* Each segment has length specified by the parameter *fftlength* ;

- Each sub-segment is windowed using a Kaiser7 function and its magnitude frequency is computed from using an FFT algorithm.

- Linear averaging of the computed Magnitude FFT is performed.

- The vector mFrequencyResponse of FreqPoint elements containing the averaged magnitude response of the ADC in the frequency domain is obtained.

- The levels of the response at the stimulus signal's fundamental frequencies is measured, the search is performed in the immediate neighbourhood of the components to allow for limitations in the FFT resolution.
This operation returns two elements of type FreqPoint: lowPeak and highPeak.

- For this type of test the "SMPTE" method for intermodulation distortion calculations is selected, therefore we measure the power of the side-bands around the highest stimulus tone; the search is performed in the immediate neighbourhood of the components to allow for limitations in the FFT resolution:

```
double DSum = 0;
for (size_t sbIdx = 1; sbIdx < 3; sbIdx++)
{
        //upper sidebands
        double startFreqU = (mInputSignalFrequencyHigh + sbIdx*mInputSignalFrequencyLow) -      binResolution * 2;
```

```
        double endFreqU   = (mInputSignalFrequencyHigh + sbIdx*mInputSignalFrequencyLow) +      binResolution * 2;
        FreqPoint FU = findPeakInRange(startFreqU, endFreqU, mFrequencyResponse);

        //lower sidebands
        double startFreqL = (mInputSignalFrequencyHigh - sbIdx*mInputSignalFrequencyLow) -      binResolution * 2;
        double endFreqL   = (mInputSignalFrequencyHigh - sbIdx*mInputSignalFrequencyLow) +      binResolution * 2;

        FreqPoint FL = findPeakInRange(startFreqL, endFreqL, mFrequencyResponse);

        double sbSum = (FU.peakValueLin + FL.peakValueLin);
        DSum += ((sbSum*sbSum) / (highPeak.peakValueLin*highPeak.peakValueLin));
}
```

- Finally, we obtain the percentile and log value for the intermodulation distortion as:

```
mIMD_Percentile = 100 * sqrt(DSum);
mIMD_Log = 20 * log10(mIMD_Percentile/100);
```

- The pass/fail outcome is obtained by comparing the value of mIMD_Log with the published reference.

5.5.8 High frequency intermodulation distortion test unit.

Signal generation module:

```
<parameter name="signal" type="string" value="dualsine" units="" alias="dual tone" editable="false"/>
```

The TestManager instantiate the class: Dual*SineToneGenerator* declared in ./SigGen/Dual*SineToneGenerator.h*

Signal generator's parameters:

Stimulus 1 frequency:

```
<parameter name="tonefreq1" type="double" value="18000" units="Hz" alias="signal 1 frequency" editable="true"/>
```

Stimulus 2 frequency:

```
<parameter name="tonefreq2" type="double" value="20000" units="Hz" alias="signal 2 frequency" editable="true"/>
```

Total signal level:

```
<parameter name="tonelevel" type="double" value="-1" units="dB" alias="total signal level" editable="true"/>
```

Level ratio of stimuli:

```
<parameter name="levelratio" type="double" value="1" units="dB" alias="signals levels ratio" editable="true"/>
```

Signal duration:

```
<parameter name="fftlength" type="double" value="32768" units="" alias="fft size" editable="true"/>
<parameter name="fftnoavg" type="double" value="16" units="" alias="number of fft averages" editable="true"/>
```

Analysis module:

<parameter name="analyser" type="string" value="lfimd" units="" alias="high or low frequency imd" .. />

The TestManager instantiate the class: *IMD* declared in ./Analysers/IMD.*h*

Analyser's operational parameters:

Signal detection level:
<parameter name="detectionlevel" type="double" value="-70" units="dB" alias="signal detection threshold" .. />

FFT size:
<parameter name="fftlength" type="double" value="32768" units="" alias="fft size" editable="true"/>

Number of averages:
<parameter name="fftnoavg" type="double" value="16" units="" alias="number of fft averages" editable="true"/>

Type of averaging function:
<parameter name="fftavgtype" type="string" value="linear" units="" alias="averaging type" editable="true"/>

Intermodulation distortion calculation method:
<parameter name="imdtype" type="string" value="ccif" units="" alias="imd method" editable="true"/>

Analysis procedure:

Same procedure as the LFIMD test unit with the notable differences:

- The "CCIF" method is used for calculation of the intermodulation distortion, therefore second and third order difference frequency distortions (DFD2 and DFD3) need to be measured from the averaged spectrum.
Search ranges include the immediate neighbourhood of the difference components to allow for limitations in the FFT resolution:

```
//2nd order Difference frequency
double df2_start = (mInputSignalFrequencyHigh - mInputSignalFrequencyLow) - binResolution * 2;
double df2_end   = (mInputSignalFrequencyHigh - mInputSignalFrequencyLow) + binResolution * 2;
FreqPoint DF2 = findPeakInRange(df2_start, df2_end, mFrequencyResponse);


//2nd order Difference frequencies
double df3a_start = (2*mInputSignalFrequencyHigh - mInputSignalFrequencyLow) - binResolution * 2;
double df3a_end   = (2*mInputSignalFrequencyHigh - mInputSignalFrequencyLow) + binResolution * 2;
FreqPoint DF3A = findPeakInRange(df3a_start, df3a_end, mFrequencyResponse);

double df3b_start = (2 * mInputSignalFrequencyLow - mInputSignalFrequencyHigh) - binResolution * 2;
double df3b_end   = (2 * mInputSignalFrequencyLow - mInputSignalFrequencyHigh) + binResolution * 2;
FreqPoint DF3B = findPeakInRange(df3b_start, df3b_end, mFrequencyResponse);

mDFD2_Percentile = 100 * (DF2.peakValueLin / (highPeak.peakValueLin + lowPeak.peakValueLin));
mDFD3_Percentile = 100 * ((DF3A.peakValueLin+DF3B.peakValueLin)/(highPeak.peakValueLin + lowPeak.peakValueLin));
```

- Finally, we obtain the percentile and log value for the intermodulation distortion as:

```
mIMD_Percentile = mDFD2_Percentile + mDFD3_Percentile;
mIMD_Log = 20 * log10(mIMD_Percentile / 100);
```

- The pass/fail outcome is obtained by comparing the value of mIMD_Log with the published reference.

5.5.9 Spurious inharmonic signals test unit.

Signal generation module:

<parameter name="signal" type="string" value="singlesine" units="" alias="single sine tone" editable="false"/>

The TestManager instantiate the class: *SingleSineToneGenerator* declared in ./SigGen/*SingleSineToneGenerator.h*

Signal generator's parameters:

Stimulus frequency:
<parameter name="tonefreq" type="double" value="997" units="Hz" alias="signal frequency" editable="true"/>

Stimulus level:
<parameter name="tonelevel" type="double" value="-1" units="dB" alias="signal level" editable="true"/>

Signal duration:
<parameter name="fftlength" type="double" value="32768" units="" alias="fft size" editable="true"/>
<parameter name="fftnoavg" type="double" value="32" units="" alias="number of fft averages" editable="true"/>

Analysis module:

<parameter name="analyser" type="string" value="spis" units="" alias="spurious iharmonic analyser" .. />

The TestManager instantiate the class: *SpIS* declared in ./Analysers/SpIS.*h*

Analyser's operational parameters:

Signal detection level:
<parameter name="detectionlevel" type="double" value="-50" units="dB" alias="signal detection threshold" .. />

FFT size:
<parameter name="fftlength" type="double" value="32768" units="" alias="fft size" editable="true"/>

Number of averages:
<parameter name="fftnoavg" type="double" value="32" units="" alias="number of fft averages" editable="true"/>

Type of averaging function:
<parameter name="fftavgtype" type="string" value="linear" units="" alias="averaging type" editable="true"/>

Bandwidth of notch used for exclusion of fundamental tonal component:

<parameter name="notchbw" type="double" value="100" units="hz" alias="fundamental notch bandwidth" editable="true"/>

Search range for each harmonic component:
<parameter name="harmsearchbw" type="double" value="20" units="hz" alias="search range of harmonics for THD calculations" editable="true"/>

Low-end exclusion zone for calculations:
<parameter name="lowerlimit" type="double" value="20" units="hz" alias="lowest frequency used in calculations" editable="true"/>

High-end exclusion zone for calculations:
<parameter name="higherlimit" type="double" value="20000" units="hz" alias="highest frequency used in calculations" editable="true"/>

Analysis procedure:

The stimulus signal consists in a discrete tone at 997 Hz @ -1dBFS

The LPCM Wave file containing the recorded response from the ADC under test is opened and the signal is analysed using the FADGIAnalyser::getOnsets(…) function.
A vector containing the position, in samples, of the recorded signal is returned by this method.
In this case, only one segment will be detected.

The active segment of the signal is processed in the *analyseSegment(…)* function and the following operations are performed:

- The transient guarding period is discarded from the segment.

- The segment is divided into the number of sub-segments specified by the *fftnoavg* parameter*;*
Each segment has length specified by the parameter *fftlength* ;

- Each sub-segment is windowed using a Kaiser7 function and its magnitude frequency is computed from using an FFT algorithm.

- Linear averaging of the computed Magnitude FFT is performed.

- The vector mFrequencyResponse of FreqPoint elements containing the averaged magnitude response of the ADC in the frequency domain is obtained.

- The element corresponding to the stimulus fundamental tone is located in the response signal's spectrum.

FreqPoint **stimPnt = findPeakInRange(lowestFrequency, highestFrequency, mFrequencyResponse);**
The search range is defined by values obtained from the unit test's operational parameters:

```
double lowestFrequency = getTestParameterValue(wxT("lowerlimit"), mParamsNode);
double highestFrequency = getTestParameterValue(wxT("higherlimit"), mParamsNode);
```

- The fundamental frequency of the stimulus and all higher order harmonics are removed from the spectrum. Neighbouring components are also suppressed, using a notch filter defined as:

```
mNotchBandwidth = getTestParameterValue(wxT("notchbw"), mParamsNode);
```

- A search for the highest component in the suppressed spectrum is performed:

```
//now find highest peak in suppressed frequency response
FreqPoint inhPnt = findPeakInRange(lowestFrequency, highestFrequency, mFrequencyResponse);
```

- The frequency, linear and log peak amplitude values of the strongest inharmonic component are therefore:

```
mMaxSpISFrequency = inhPnt.frequency;
mMaxSpISLevel_Lin = inhPnt.peakValueLin;
mMaxSpISLevel_Log = inhPnt.peakValueLog;
```

- The pass/fail outcome is obtained by comparing the value of mMaxSpISLevel_Log with the published reference.

6. Building ADCTester from the source.

ADCTester has been developed primarily as a Windows application, therefore a Visual Studio Community 2015 solution is provided with the source tree.
Very few Windows-specific objects have been written, and all external libraries used in the project are cross-platform: it should be possible, therefore, to port the application to other Operating Systems.

6.1 Third Party libraries.

The following libraries have been used to develop ADCTester:

wXWidgets frameworks.

WxWidgets is a cross-platform C++ library that provides an application framework, GUI base classes and threads implementation.
Version 3.1.1 has been used in the development of ADCTester.
The library is released under the wxWindows library license.
https://www.wxwidgets.org/about/licence/

PortAudio library.

PortAudio provides a very simple API for recording and/or playing sound using a simple callback function or a blocking read/write interface.
Licensing information can be found here: http://www.portaudio.com/license.html

LibSndfile library.

LibSndFile is a C library that provides support for reading and writing compressed/uncompressed audio files.
The library is released under L-GPL. http://www.mega-nerd.com/libsndfile/#Licensing

6.2 Third party source code.

The following classes have been used to develop ADCTester:

Kiss FFT.

A lightweight FFT implementation, released under the BSD license.
https://sourceforge.net/projects/kissfft/

wxMathPlot

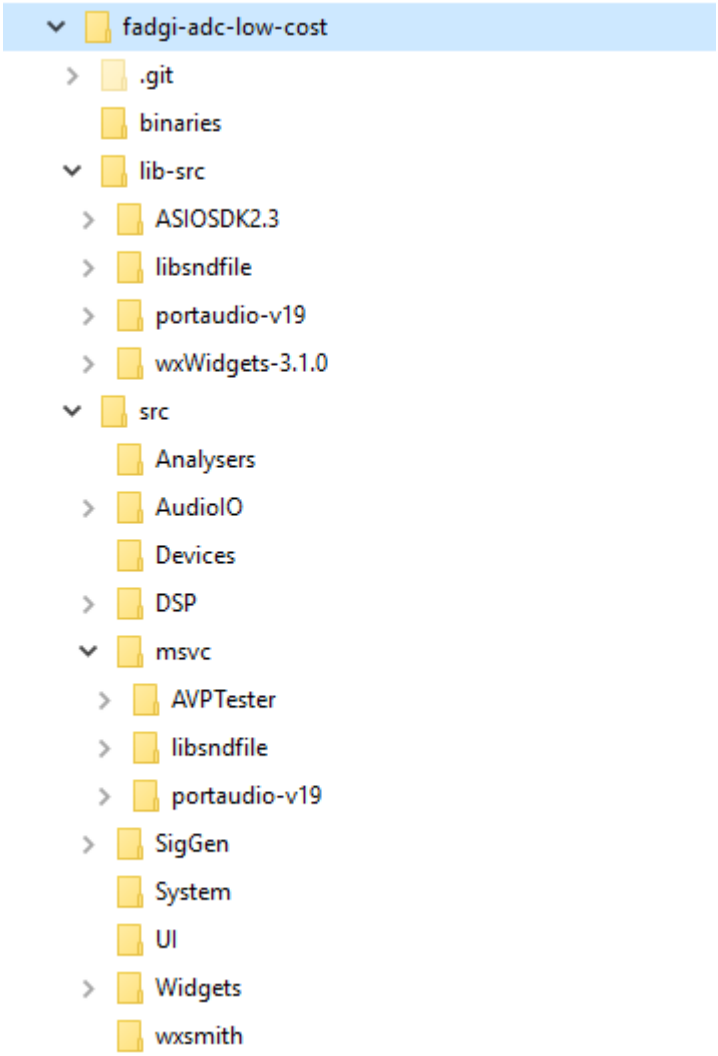
A library to add 2D scientific plot functionality to wxWidgets.

It provides a window for plotting scientific, statistical or mathematical data, with additions like legend or coordinate display in overlay.
wxMathPlot is distributed under the therms of the http://wxmathplot.sourceforge.net/

The original code has been modified to provide desired functionalities to the ADCTester application.

6.3 Folder structure.

The source repository has the following structure:



binaries
Pre-compiled executable and Windows installer.

lib-src
The folder contain source code for all libraries necessary to compile the ADCTester application.
The folder ASIOSDK2.3 provides ASIO support to portAudio.

src
The folder contain all source code used by the ADCTester application.

Src\Analysers
Analysers modules.

Src\AudioIO
AudioEngine and helper classes, and PortAudio interfacing.

Src\Devices
Audio devices discovery class

Src\DSP
Classes providing signal processing functionalities to the analysers.

Src\msvc
Contains MSVC 2015 solutions and projects files for the ADCTester application, libsndfile and portaudio.

Src\SigGen
Test signal generation modules.

Src\System
Application preferences I/O.

Src\UI
Graphics files used by GUI classes.

Src\Widgets
Custom GUI classes.

Src\wxsmith
Design templates used for the configuration of the application's Frames and Dialogs.
An external IDE has been used for this purpose.
In practice we use the wxSmith plugin in code::blocks to quickly generate and modify the main UI classes and MSVC 2015 to compile the application.

6.4 Compiling ADCTester.

First the wxWidgets libraries must be compiled:

Go to: ***lib-src\wxWidgets-3.1.1\build\msw***

load the solution named: ***wx_vc15.sln*** into Microsoft Visual Studio Community 2017.

Build the solution using the following configurations: ***Win32 Debug*** and ***Win32 Release***

This will build both debug and release multi-threaded static libraries for wxWidgets.

Next build the application, go to:

\src\msvc\AVPTester

load the solution named: ***AVPTester.sln*** into Microsoft Visual Studio Community 2015.

Build the solution using the following configurations: ***x86 Debug*** and ***x86 Release***

This will build both debug and release version of the application.

The binaries will be saved in: ***\src\msvc\AVPTester\Debug*** and ***\src\msvc\AVPTester\Release***, respectively.

The distribution must contain the following elements:

AVPTester.exe **–** Application executable.
Default.avt **–** Default test procedures template.
./UI **–** folder containing run-time graphics elements.