
Getting Started with RFmx User Manual

2025-07-23



Contents

RFmx User Manual	4
RFmx Overview	5
RFmx System Requirements.....	6
Creating an Application with RFmx.....	7
Using RFmx in LabVIEW	7
Using RFmx in LabWindows/CVI.....	8
Creating an Application with Microsoft Visual C/C++ and .NET	9
RFmx Examples	12
Troubleshooting	13
Installation and Configuration	13
Programming	13
Calibration.....	13
Add-On Software for Generation	15
Concepts	16
LabVIEW	16
RFmx Overview.....	16
Selector Strings	17
Array Configuration and Fetch VIs	22
Signal Configurations	23
Spectral Measurements Concepts	26
Overlapped Measurements and Named Results.....	27
Using FPGA Extensions with RFmx.....	31
RFmx Analysis-Only Mode	33
External Attenuation Table.....	37
Standard Error Parameter Behavior.....	41
LabWindows/CVI	43
RFmx Overview CVI	43
Selector String (C/LabWindows™/CVI™)	44
Array Configuration and Fetch Functions.....	50
Signal Configurations	52
Spectral Measurements Concepts	55
Overlapped Measurements and Named Results.....	57

RFmx Analysis-Only Mode	61
.NET	65
RFmx Overview	66
Selector Strings	67
Array Configuration and Fetch Functions	74
Signal Configurations	77
Composite Measurements	81
Overlapped Measurements and Named Results	84
Creating a Session	89
Closing a Session	89
RFmx Analysis-Only Mode	91

RFmx User Manual

The RFmx User Manual provides detailed descriptions of the product functionality and the step-by-step processes for use.

Looking For Something Else?

For information not found in the User Manual for your product, such as specifications and API reference, browse ***Related Information***.

Related information:

- [Download RFmx](#)
- [License Setup and Activation](#)
- [RFmx Release Notes](#)



RFmx Overview

RFmx is a family of instrument drivers consisting of multiple libraries of VIs and functions you can call from your application software, such as LabVIEW, LabWindows™/CVI™, or Microsoft Visual Studio, to program your NI vector signal analyzer devices.

RFmx instrument driver APIs are completely interoperable. For example, an RFmx LTE API can be used with all other RFmx APIs, such as RFmx SpecAn, GSM, and so on.

This is accomplished through sharing a single session handle across multiple drivers. You need to open only one RFmx session per signal analyzer and use it to configure and perform measurements with the different RFmx drivers. In addition, the RFmx API provides simple access to the most advanced optimization techniques such as multi-measurement parallelism and multi-DUT measurements. The result is extremely fast and high-quality measurements with minimal software development effort.

RFmx System Requirements

RFmx has the following requirements:

- Processor—1 GHz 64-bit (x64) processor
- 4 GB RAM *
- A screen resolution of 1,024 x 768
- Any supported OS, with all available critical updates and service packs

* Depending on the amount of data acquired and/or processed, a larger amount of memory may be required.

Creating an Application with RFmx

RFmx is the instrument driver that controls the configuration and operation of the NI vector signal analyzer devices. RFmx reduces program development time, facilitates parallel testing of multiple RF components, and simplifies device control.

RFmx is a family of instrument drivers that share a single session handle across multiple drivers. You need to open only one RFmx session per signal analyzer and use it to configure and perform measurements with the different RFmx drivers, such as SpecAn, Demod, LTE, and so on.

RFmx supports the following application development environments (ADEs):

- NI LabVIEW
- NI LabWindows/CVI
- Microsoft Visual C/C++/C# and VB .NET

Application Manifest File

NI recommends that you embed an application manifest in the `.exe` file you create with RFmx to avoid undesirable visual artifacts on very high resolution displays running Windows 10. For WPF-based applications, the `dpiAware` element in the manifest should be set to `true/pm`. For other applications it should be set to `false`. Refer to ***Application Manifests*** on the Microsoft TechNet site for more information about application manifests. Refer to your ***ADE documentation*** for information about how to embed an application manifest in your `.exe` file.

Using RFmx in LabVIEW

To develop an RFmx application in LabVIEW, follow these steps:

1. Open an existing or new LabVIEW VI.
2. From the Functions palette, locate the RFmx VIs at **Measurement I/O»RFmx**.
3. Select the VIs that you want to use and drop them on the block diagram to build your application using the appropriate programming flow steps as described in

RFmx Overview topic.



Note Refer to the *LabVIEW Help* for information about using the LabVIEW ADE.

Related concepts:

- [RFmx Overview](#)

Using RFmx in LabWindows/CVI

LabWindows/CVI support is available for the current version of the RFmx SpecAn and the RFmx Demod drivers.

To develop an RFmx SpecAn application in LabWindows/CVI, follow these steps:

1. Open an existing or new project file.
2. Load the RFmx function panel from the location specified in the RFmx SpecAn Readme.
3. Use the function panel to navigate the function hierarchy and generate function calls with the proper syntax and variable values.
4. Build your application using the appropriate programming flow steps as described in RFmx Overview topic.



Note Refer to the LabWindows/CVI Help for information about using the LabWindows/CVI ADE.

Related concepts:

- [RFmx Overview CVI](#)

Creating an Application with Microsoft Visual C/C++ and .NET

Creating a C/C++ Application with Microsoft Visual Studio 2008

The ADE that you can use to create applications using Microsoft Visual C/C++ is Microsoft Visual Studio 2008 or later.

To develop an RFmx SpecAn application in Microsoft Visual C/C++, follow these general steps:

1. Open an existing or new Visual C/C++ project to manage your application code.
2. Create source code files of type .c (C) or .cpp (C++).
3. Add the source code files to the project.
4. Add following in the source code files: `#include "niRFmxSpecAn.h"`.
5. Add the RFmx include and library files to the project.



Note In the following steps, <NIDir> is an alias to specific National Instruments file folder locations. Refer to the RFmx SpecAn Readme for more information about installed file locations.



Note You can modify an RFmx SpecAn C example to create an application, and all required include and library files are added to the project for you

- a. Select **Project»Properties»Configuration Properties»C/C++»General»Additional Include Directories**, and add the path :
`<NIDir>\Shared\ExternalCompilerSupport\C\include`
- b. Select **Linker»Input»Additional Dependencies**, and add `niRFmxInstr.lib` and `niRFmxSpecAn.lib` separated by a space.



Note For Visual Studio 2010 and above, add `niRFmxInstr.lib` and `niRFmxSpecAn.lib` separated by a semicolon .

- c. Select **Linker»General»Additional Dependencies** and add the path:
`<NIDir>\Shared\ExternalCompilerSupport\C\lib64\msvc;`

6. Build your application using the appropriate programming flow steps as described in the RFmx Overview topic.

Creating a .NET Application with Microsoft Visual Studio 2010

The ADE that you can use to create applications using Microsoft Visual C# or VB .NET is Microsoft Visual Studio 2010 or later.

To develop an RFmx SpecAn application in Microsoft Visual C# or VB .NET, follow these general steps:

1. Open an existing or new Microsoft Visual C# or VB .NET project to manage your application code.
2. Create source code files of type .cs (C#) or .vb (VB .NET).
3. Add the source code files to the project.
4. Use the following RFmx namespaces in the source code files:
 - using NationalInstruments.Common;
 - using NationalInstruments.RFmx.InstrMX;
 - using NationalInstruments.RFmx.SpecAnMX;
5. Set the target .NET Framework of your .NET project to 4.0 or higher.
 - In **Solution Explorer**, right click on your project name and click **Properties**.
 - Click on the **Application** tab and choose .NET Framework 4.0 or higher from the **Target Framework** list.



Note Client Profile is not supported.

6. Add the following .NET Class Libraries to the project.



Note You can modify an existing RFmx C# or VB .NET example to create an application, and all required .NET class libraries added to the project for you.

Select **Project»Add References**, navigate to the .NET tab and add the following files to your project references:

- NationalInstruments.Common.dll
- NationalInstruments.RFmx.InstrMX.Fx40.dll
- NationalInstruments.RFmx.SpecAnMX.Fx40.dll



Note For Visual Studio 2012 and above, navigate to the Assemblies»Extensions tab.

7. Build your application using the appropriate programming flow steps as described in the RFmx Overview topic.

RFmx Examples

RFmx includes several example applications that demonstrate the functionality of your software and can serve as interactive tools, programming models, and building blocks for your own applications.

You can access all the installed RFmx examples from the **Start** menu.

For instance, select `Start»National Instruments»RFmx SpecAn Examples` to access RFmx SpecAn examples.

NI Example Finder

The NI Example Finder is a utility that organizes examples into categories and allows you to browse and search installed examples. For example, search for "SpecAn" to locate all SpecAn examples. You can see descriptions and compatible hardware models for each example or see all the examples compatible with one particular hardware model.

To locate examples using the NI Example Finder within LabVIEW or LabWindows/CVI, select **Help»Find Examples**. The NI Example Finder offers two ways to access all installed LabVIEW example VIs and their descriptions:

- Click the **Browse** tab to locate examples by task at `Hardware Input and Output»Modular Instruments»RFmx`.
- Click the **Search** tab to search all installed examples by keyword. For example, search for SpecAn to locate all RFmx SpecAn examples.



Note NI Example Finder support is available only for the RFmx SpecAn and the RFmx Demod drivers.

Installation and Configuration

RFmx drivers requires that all the installed driver versions remain the same for proper functionality. This requirement is because RFmx drivers share components. If driver incompatibilities exist, they are reported through a dialog box after installation. You can also use the Version Compatibility Checker utility to manually check for version incompatibilities.

Use the following resources if you have problems installing your NI-RFSA hardware and/or software:

- Refer to ni.com/kb for documents on troubleshooting common installation and programming problems and for answering frequently asked questions about NI products.
- If you think you have damaged your device and need to return your NI hardware for repair or calibration, refer to ***How Do I Send a National Instruments board in for repair or calibration?*** topic at ni.com/kb.

Programming

To help you get started programming, you can:

- Use shipping examples for your ADE
- Visit NI's extensive library of technical support resources at ni.com/support
- Use NI I/O Trace to analyze the functions you have called in the API with NI application software. With NI I/O Trace, you can watch the order of execution of the application and locate errors as they happen.

Related concepts:

- [Creating an Application with RFmx](#)

Calibration

For information about externally calibrating your device, including step-by-step calibration procedures, refer to ni.com/calibration.

For the differences between self-calibration and external calibration, refer to ***Devices*** section in the NI RF Vector Signal Analyzers Help.

Add-On Software for Generation

Add-On Software for Generation

NI provides the following add-on software toolkits, which extend the capability of the RF vector signal generator.

RFmx Waveform Creator

The RFmx Waveform Creator provides an easy-to-use interface to create and generate generic RF and cellular signals quickly and easily. Signal configurations can be saved in the rfws format and the waveforms can be saved in the TDMS file format for future use.

NI-RFSG Playback Library

The NI-RFSG Playback Library extends the functionality of the NI-RFSG API by adding the ability to easily load and playback waveforms from TDMS files created by the NI software. Use the NI-RFSG Playback library API in conjunction with waveforms generated using the RFmx Waveform Creator to easily generate RF waveforms in the ADE environment of your choice (LabVIEW/LabWindows/CVI/Visual Studio).

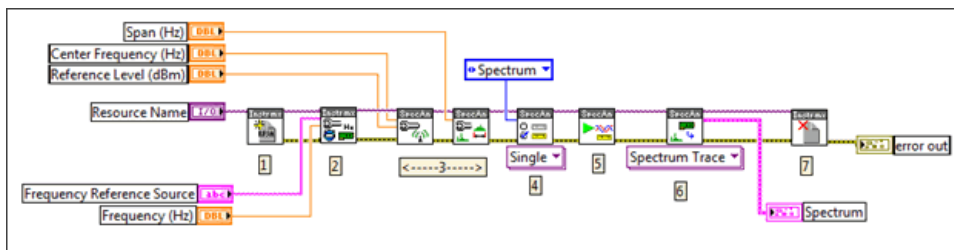
RFmx Overview

RFmx is a family of instrument drivers that share a single session handle across multiple drivers. You need to open only one RFmx session per signal analyzer, and you use that single session handle to configure and perform measurements with the different RFmx drivers, such as SpecAn, Demod, LTE, and so on.

The RFmx driver APIs have a shared set of API VIs for opening and closing the RFmx session as well as configuring hardware settings, such as the Reference Clock configuration, that are shared across drivers. These shared VIs are called the RFmxInstr driver. RFmxInstr does not contain any measurements, and you always use RFmxInstr along with one or more other RFmx drivers that expose the measurements you need to perform.

Because you must use RFmxInstr VIs to use the RFmx drivers that contain the measurements, for convenience, commonly used VIs, such as RFmxInstr Initialize NIRFSA VI, RFmxInstr Close VI, and RFmxInstr Wait for Acquisition Complete VI, are replicated in the LabVIEW palettes for the other RFmx drivers.

The following is an example of how you use the RFmx API drivers in LabVIEW to capture a spectrum using the RFmx SpecAn driver. The same sequence applies to other ADEs as well.



1. Open an RFmx session using RFmxInstr.
2. Configure the frequency reference using the VIs or the properties in RFmxInstr.
3. Configure the reference level, frequency, and span using RFmxSpecAn.
4. Select the Spectrum measurement using RFmxSpecAn Select Measurement VI.
5. Initiate the measurement using RFmxSpecAn.
6. Fetch the spectrum using RFmxSpecAn.
7. Close the RFmx session using RFmxInstr or RFmxSpecAn.

When you use RFmxInstr VIs to configure hardware settings in an RFmx session, those settings apply to all the RFmx drivers that are present in the same RFmx session. However, the following settings are RFmx driver-specific and are not shared across different RFmx drivers within the same RFmx session:

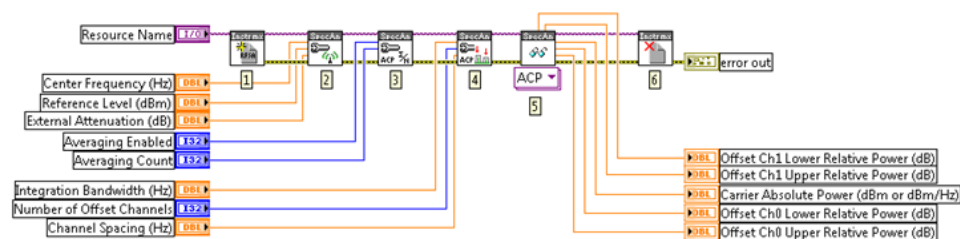
- Center frequency
- Reference level
- External attenuation
- Trigger settings

For example, if you are making both RFmx LTE and RFmx WCDMA measurements within one RFmx session, setting the frequency in RFmx LTE does not affect the frequency used in RFmx WCDMA.

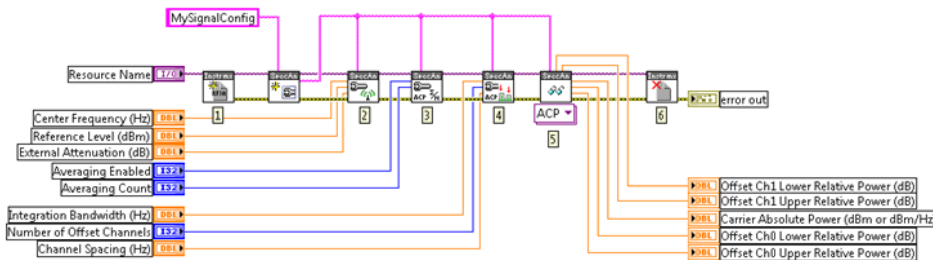
Selector Strings

RFmx is a powerful API that supports basic and advanced features and provides multiple ways to configure measurements and fetch measurement results. The same configuration and fetch VIs/functions can be used in basic and advanced use cases, such as named signal configurations, without having to rewrite code. The following image shows a basic ACP measurement that does not use named signal

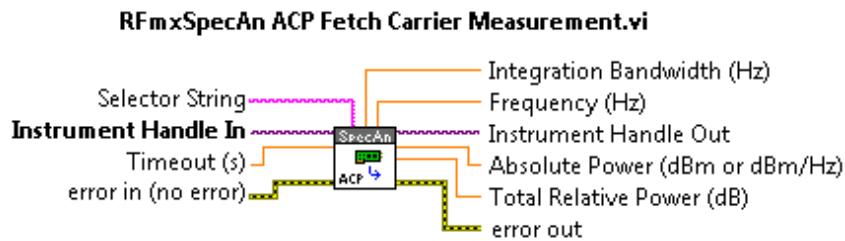
configurations.



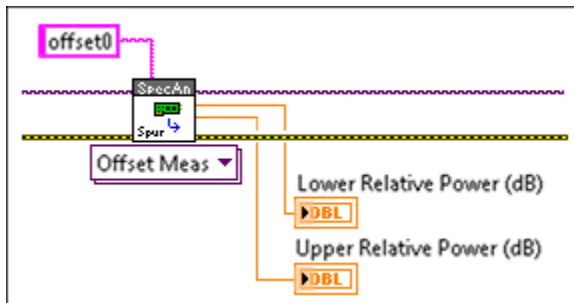
You can enhance the code as shown in the following figure to use named signal configurations by adding a Create Signal VI and wiring the signal name into the **Selector String** input of the VIs/functions the basic ACP code was already using.



The RFmx **Selector String** input allows you to specify the full context of what you are configuring or fetching. The following figure shows an example of the **Selector String** input on one of the RFmx fetch measurements VIs.



For example, an ACP measurement can have multiple offset channels, and you can configure the frequency and measurement bandwidth properties for each offset channel. When you configure the measurement bandwidth for a specific offset channel, you must specify the offset channel for which you are configuring the Measurement Bandwidth property. When you configure the first offset channel, you must specify the context in the **Selector String** input as “offset0”, as shown in the following figure.



The contexts you can specify depend on what you are configuring or fetching. For example, if you are configuring a carrier channel, you specify which carrier channel, such as “carrier3”, you are configuring. If you are configuring an offset channel, you specify which offset channel, such as “offset2”, you are configuring.

Some contexts, such as a named signal configuration, can be specified in all configuration and fetch VIs. Some contexts, such as a named result, can be specified in only the initiate and fetch VIs.

Multiple Contexts

When using multiple features in the API, you can specify multiple contexts in the same

Selector String input separated with a '/'. For example, to configure carrier 0 in the named signal configuration called “MySignal”, you must specify “signal::Mysignal/carrier0” as the selector string.

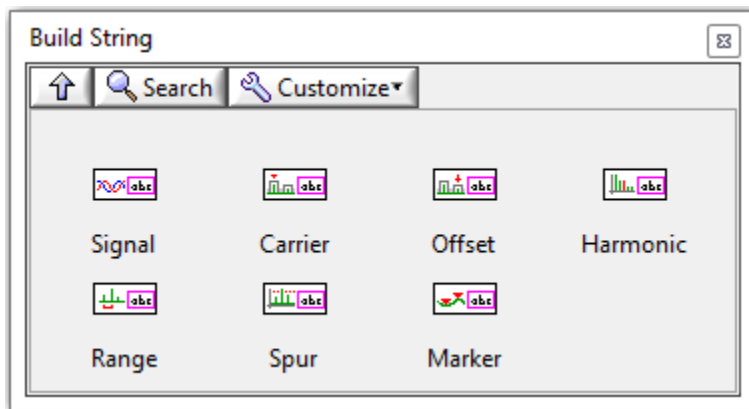
Another example of multiple contexts is performing LTE noncontiguous carrier aggregation measurements. In this case, the signal is composed of multiple subblocks, and each subblock can contain multiple carriers. When you configure a property of a carrier in LTE noncontiguous carrier aggregation, you must specify the index of the carrier within the subblock and the index of the subblock. For example, you would specify that you are configuring the first carrier in the second subblock by using “subblock1/carrier0” as the selector string.

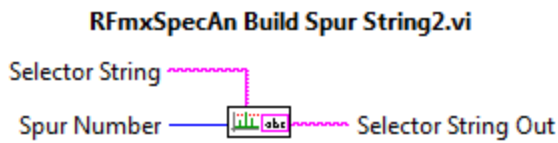
String Prefixes in Selector Strings

RFmx requires a prefix, such as “carrier” or “signal::”, before the number or name in the selector string so that it is clear what object is being configured. The “::” is required for “signal::” and “result::” because those items use arbitrary user-defined names and the API must identify where the user-specified name starts in the selector string. You can specify the selector strings as constants, such as “carrier0” and “signal::mysignal” or use the build string utility VIs/functions provided by RFmx to create correctly formed strings.

Build String Utility VIs/Functions

RFmx provides a set of utility VIs/functions/methods, as shown in the following figure, to build the context string for various measurement configurations.





For example, the Build Signal String VI takes in a named signal configuration name and/or a result name and constructs a string containing the contexts you specified. If you call the Build Signal String VI with “MySignal” passed to the **Signal Name** input, the output string is “signal::MySignal”. Each RFmx driver provides build string utility VIs/functions/methods for specific cases, such as Build Spur String2, which takes in a spur number in addition to optionally taking in a previously constructed selector string. If you call the Build Spur String2 VI with a spur set to 2, and the **Selector String** input set to the output of the previously called Build Range String2 VI, say range1, the output string is “signal::MySignal/range1/spur2”.

Selector String in Property Node

When setting properties or fetching results from an RFmx property node in LabVIEW, you specify the context using the Selector String property. You must set the Selector String property above the other properties that you want the context in the selector string to apply to.

Selector Strings when using Array VIs

Notice that RFmx provides array versions of configure and fetch VIs that allow you to configure multiple instances of a property or fetch multiple instances of a result. For example, to configure the frequencies of all the ACP measurement offset channels, you can use the ACP Configure Offset (Array) VI, passing in an array that has the frequency for each offset channel. When using these array VIs/functions/methods, you do not need to specify in the **Selector String** input the context for the elements in the array. For example, you pass “” as the selector string in the Configure ACP Offset Channel (Array) VI. However, if you are using named signal configurations or named results, you pass those contexts in the selector string even when you use the array VIs/functions/methods.

Selector String Syntax

The following table lists the various selector string formats that you can use. For integral channel specifiers, valid values for *i* are 0 to (number of channels - 1). For

example, if you set the value of ACP Num Offsets property to 5, you can configure ACP offset properties for offset0, offset1, offset2, offset3, and offset4.

In the following table, the term channel or ch is used to refer to numeric contexts such as carrier, offset, or range.

Selector String	String Function	Example
ch<i>	Configures one channel	carrier0
ch<i>-<j>	Configures a range of channels between i and j	carrier0-5
ch<i>:<j>	Configures a range of channels between i and j	carrier0:5
ch<i>, ch<j>, ch<k>	Configures channels i, j, and k	carrier0, carrier2, carrier4
ch<i>, ch<j>-<k>	Configures channel i and a range of channels between j and k	carrier0, carrier1-5
ch<i>, ch<j>:<k>	Configures channel i and a range of channels between j and k	carrier0, carrier1:5
ch::all	Configures all channels from 0 to (Number of X - 1). Number of X is the current value of the Number of ... property, that corresponds to the context used. For example, in an ACP measurement, you set the Number of Offsets property before setting any offset properties and when you use “all”, RFmx uses the current value of Number of Offsets to determine which offset channels to configure.	offset::all

The following table lists the various selector string formats that you can use to specify the named signal configurations (using the “signal::” prefix) and named results (using the “result::” prefix). Notice that the “result::” prefix is valid only when fetching results and currently cannot be used to configure measurements.

Syntax	String Function	Example
signal::<signalname>	Specifies signal <signalname>	signal::sig1
signal::<signalname>/ch<i>	Specifies channel i on signal <signalname>	signal::sig1/ carrier0
signal::<signalname>/result::<resultname>/ch<i>	Specifies channel i for result	signal::sig1/

Syntax	String Function	Example
	<resultname> on signal <signalname>	result::r1/ carrier0
signal::<signalname>/result::<resultname>	Specifies named result <resultname> on signal <signalname>	signal::sig1/ result::r1
result::<resultname>	Specifies named result <resultname> on the default signal	result::r1

The following characters are not allowed in signal names and result names: ()*+,-./{} !"#\$%&'";<=>?@[]\^`|~ and any non-printable character.

Related concepts:

- [Signal Configurations](#)
- [Overlapped Measurements and Named Results](#)
- [Array Configuration and Fetch VIs](#)

Array Configuration and Fetch VIs

Array Configuration VIs

RFmx provides array-based configuration VIs to enable you to set different values to multiple instances of a property in a single call to the VI. The first N instances of each property are configured as the values specified in the corresponding elements of the array. If you pass an empty array to the input of a VI, none of the instances of the property corresponding to that input are modified. You can pass up to M values in the array, where M is the current value of the <Number of X> property (for example, Number of Carriers, Number of Harmonics, Number of Offsets, and so on). If the array size, N, is smaller than M, only the first N instances of the property are modified. For example, if you set the ACP Number of Offsets property to three, and if the Offset Sideband parameter of the RFmxSpecAn ACP Configure Offset (Array) VI has two elements in the array, the Offset Sideband property of the first two offset channels are set to the values in the array, and the Offset Sideband property of the third offset channel is not modified.

To set the values to a subset of property instances, you can use the non-array-based VIs. In the non-array-based VIs, you can configure one or more instances of a property to the same value in each call to the VI.

Array Fetch VIs

RFmx provides array-based Fetch VIs to enable you to fetch multiple instances of a result in a single call to the VI. Array Fetch VIs return arrays containing the values for all instances of the results during a specific call to a measurement. For example, if you have configured an ACP measurement to measure three offset channels, the array contains the results for these three offset channels.

Signal Configurations

RFmx allows you to create multiple measurement configurations, called signal configurations, within the same personality. Each signal configuration instance holds a unique combination of signal and measurement property values. Using multiple signal configurations allows you to switch between different measurements' configurations without configuring the measurements each time. Signal configurations also allow you to execute multiple configurations of the same measurement at the same time.

RFmx creates one default signal configuration per personality. This default configuration is unnamed. You can create additional named signal configurations.

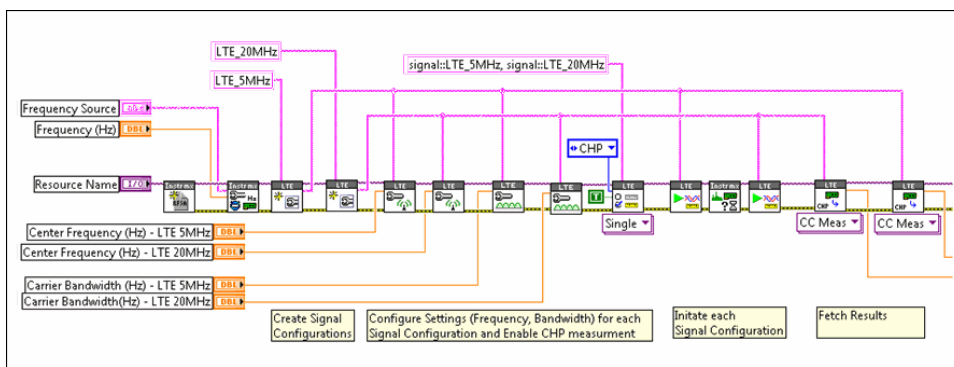
Select the signal configuration to use by passing the signal configuration name in the RFmxSpecAn Initiate VI **Selector String** input or the initiate VI of any other RFmx driver. RFmx performs the measurements that are enabled in that signal configuration. If you pass multiple signal configuration names, for example, signal::sig1, signal::sig2, and so on, to the RFmx Initiate VI, RFmx performs the measurements that are enabled in each of the signal configurations. The following signal properties must be configured for each signal configuration, otherwise their default values are used:

- Center Frequency
- Reference Level
- External Attenuation
- Trigger Settings

The following figure shows the creation of two signals, named LTE-5MHZ and LTE_20MHz in RFmx LTE. The two signal configurations use different center frequencies and bandwidths. The LTE_5MHz signal configuration is initiated before the LTE_20MHz signal configuration.

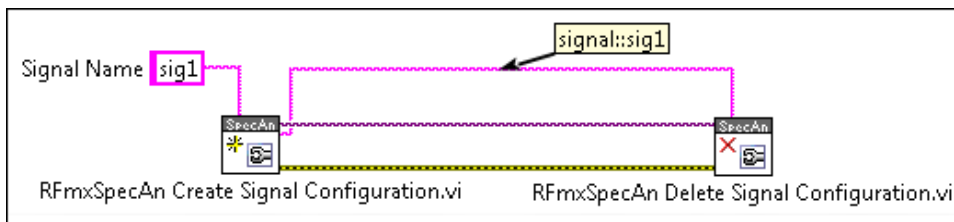


Note NI does not recommend using named signal configurations when only a few attributes are different between the configurations. For example, when performing the same measurement across different frequencies, NI recommends you use a single signal configuration and change the center frequency before each measurement.



The following characters are not allowed in signal or result names: ()*+,-./{} !"#\$%&'::<=>?@[\\^`|~ and any nonprintable character.

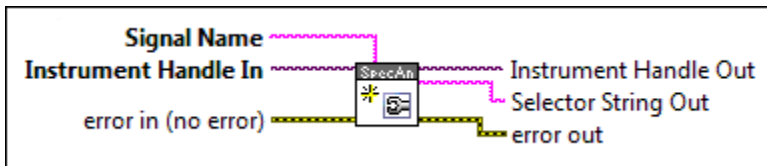
The following figure shows an example of creating and deleting a signal configuration.



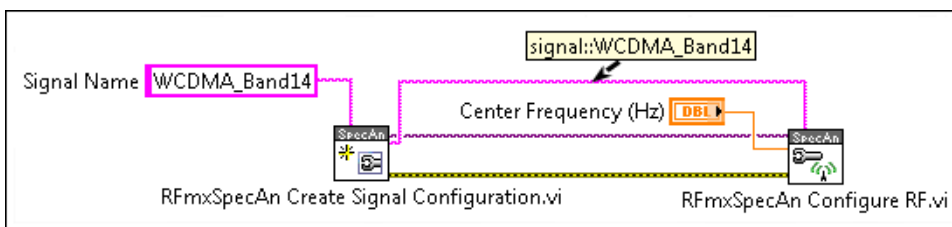
When you use signal configurations, you must provide a signal name that is correctly formatted to the RFmx Configure, Initiate, and Fetch VIs. Use the following syntax for signal names.

```
signal::
Example: signal::s1
```

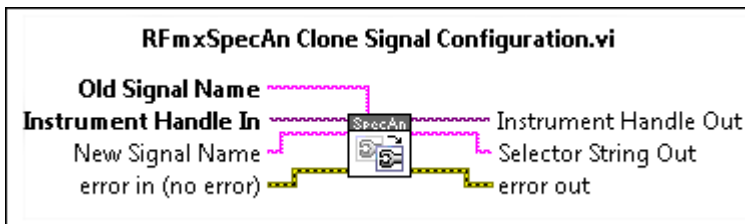
When you create a new signal configuration using the RFmxSpecAn Create Signal Configuration VI, the VI automatically creates a correctly formatted signal name that can be used in the RFmx Configure, Initiate and Fetch VIs.



The following figure shows an example of creating a signal named WCDMA_Band14 and calling a configure VI with that signal name.

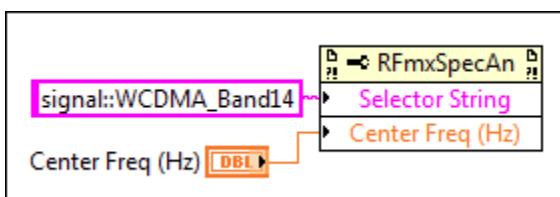


RFmx signal configurations can be cloned using a driver-specific RFmx Clone Signal Configuration VI.



When you clone a configuration, a new signal configuration is created that inherits all the configuration settings of the signal configuration being cloned. In the new signal configuration, you can overwrite the original configuration settings by calling the appropriate signal configurations VIs with the new signal configuration name.

The following figure shows an example of the property node using the signal name.

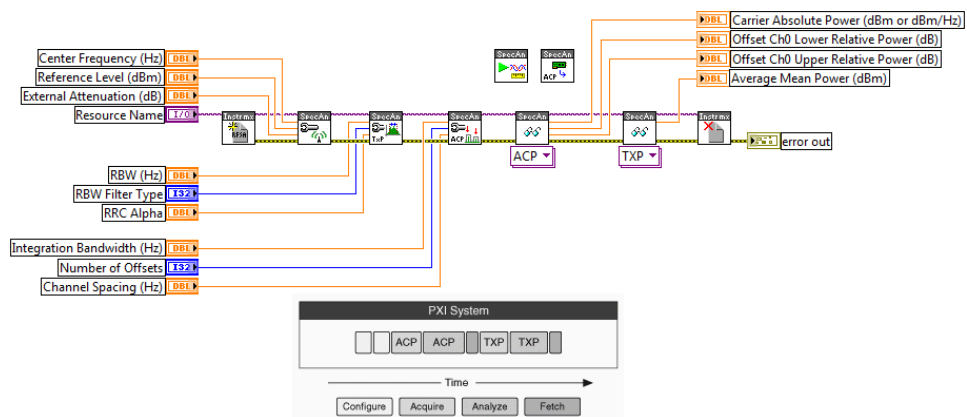


Spectral Measurements Concepts

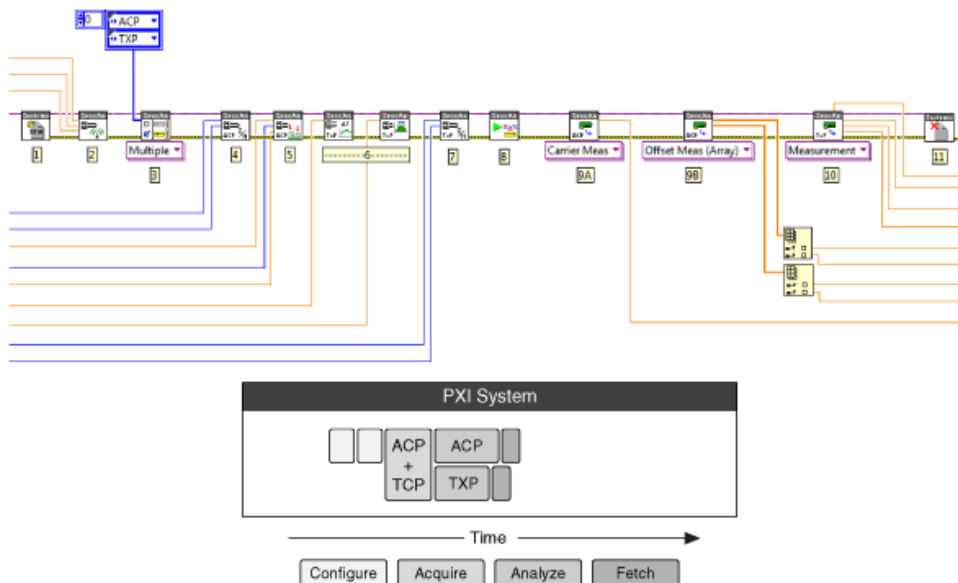
Composite measurement is a technique that allows multiple measurements to be performed in parallel on the same acquired data. This technique leads to faster measurement times because all the measurements in a composite set are based on the same acquired data, and they are processed in parallel threads on the CPU. Although the same data set is used for each measurement in a composite set, RFmx allows different configuration settings for each measurement, enabling multifaceted signal analysis.

The following figures show the difference between serial measurements and composite measurements.

Serial Measurements



Composite Measurements



When the composite measurements are performed, only a single acquisition is made and each measurement in the composite set is performed on the same acquired data. Internally, RFmx merges the acquisition settings from all the measurements in a composite set to provide the acquisition settings that satisfy the needs of all the measurements being performed. However, not all measurements can be performed as composite measurements. For example, a harmonics measurement cannot be performed in composite with an ACP measurement, because a harmonics measurement requires multisegment acquisitions for different harmonics.

To perform multiple measurements as composite measurements, use the Multiple personality of the Select Measurement VI, and wire in an array containing the measurement types you want to perform together.

Besides faster test execution times, composite measurements also allow you to see different measurements on the same DUT signal, enabling you to debug difficult issues. For example, you can see if a bad ACP result is correlated to a bad EVM result.

Overlapped Measurements and Named Results

Many automated test applications require multiple independent measurements on a single instrument. Such applications include amplifier frequency/power sweeps or multi-DUT wireless device tests. Engineers often implement these applications by serially sequencing measurements.

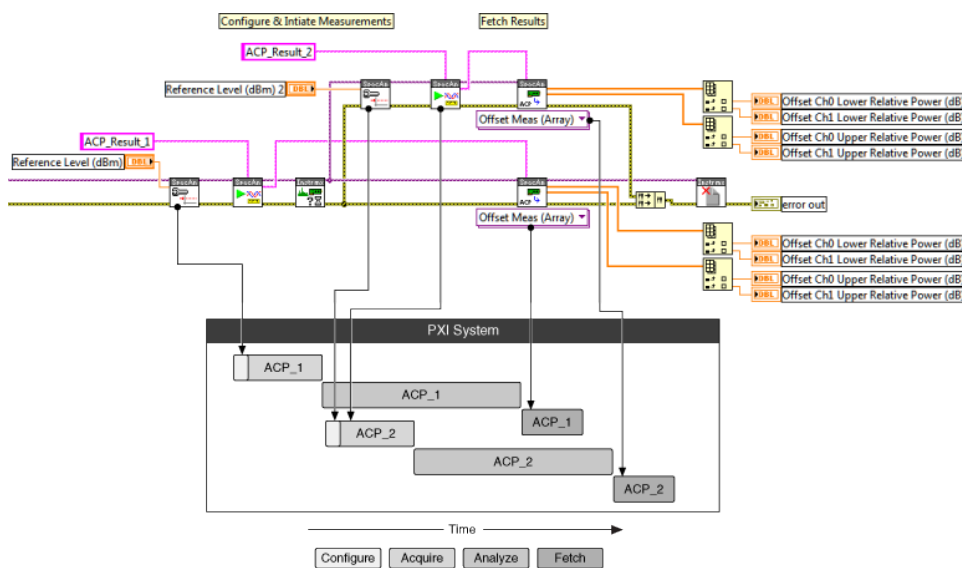
However, with RFmx, you can perform multiple measurements overlapped in time with a feature called overlapped measurements. You can use this feature when measurements in a sequence are independent of previous results. The measurements must be performed on separate acquisitions because they require different hardware configurations, such as a different frequency or reference level.

Overlapped measurements maximize use of the RF hardware and enable parallel processing of the measurements, thus vastly reducing the overall test time for a sequence of multiple measurements. In the following conceptual example, two overlapped ACP measurements are performed at two different reference levels. This simple program completes the following operations:

- Configures the reference level for the first ACP measurement.
- Initiates the first ACP acquisition.

- Waits for the ACP acquisition to be complete and the PXI RF module to be released. RFmx spawns a thread for ACP analysis in the background without delaying the next measurement.
- Initiates the second ACP acquisition before the first ACP measurement analysis is complete.
- Waits for the second ACP acquisition to be complete before the first ACP measurement analysis is complete.
- Fetches both the measurement results when their respective analysis threads are complete.

The following figure shows the RFmx overlapped measurement execution flow:



The key feature that enables overlapped measurements in RFmx is a feature called **Named Results**. In the previous example, both the first and second ACP measurements are initiated and tagged with a name. The RFmxSpecAn Initiate VI is called first with the ACP measurement enabled and the result name as ACP_Results_1, and then it is called again with the second ACP measurement enabled and the result name as ACP_Results_2. The results of these two measurements are stored independently, and can be fetched in any sequence after the measurements are complete using the same result names passed to the initiate VI.

Measurement results are referenced by the result name, measurement type, and the signal instance. If you do not specify any result name, the default result name ("") is

used.

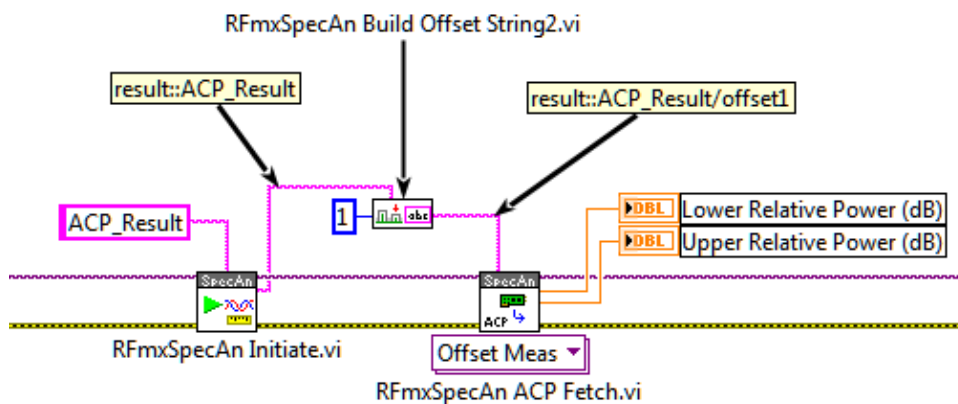
The following characters are not allowed in signal name and result name: `()*+,-./{}!\"#$%&';<=>?@[\\]^_`|~` and any nonprintable character.

When using overlapped measurements, you must specify the result name in Initiate and then use that name to fetch the results. If you reuse the same result name, the previous named results with that name are overwritten. This behavior applies to the default result name (“”) as well.

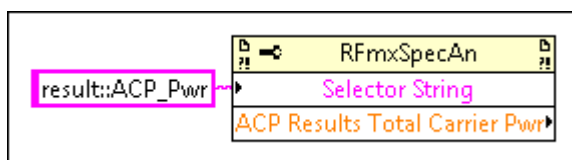
When you initiate a measurement, RFmx clears out all previous results with the same result name passed to Initiate; this clearing process keeps you from accidentally fetching a result from a previous Initiate. Notice that only the results with the same name in the default or named signal configuration passed to Initiate are cleared.

For example, if you call Initiate with Selector String set to “signal:MySignal” and Result Name set to “MyResult”, Initiate clears all results named “MyResult” that were previously initiated with the signal “MySignal”.

RFmx provides a set of Build String Utility VIs to create named result strings as shown in the following figure.



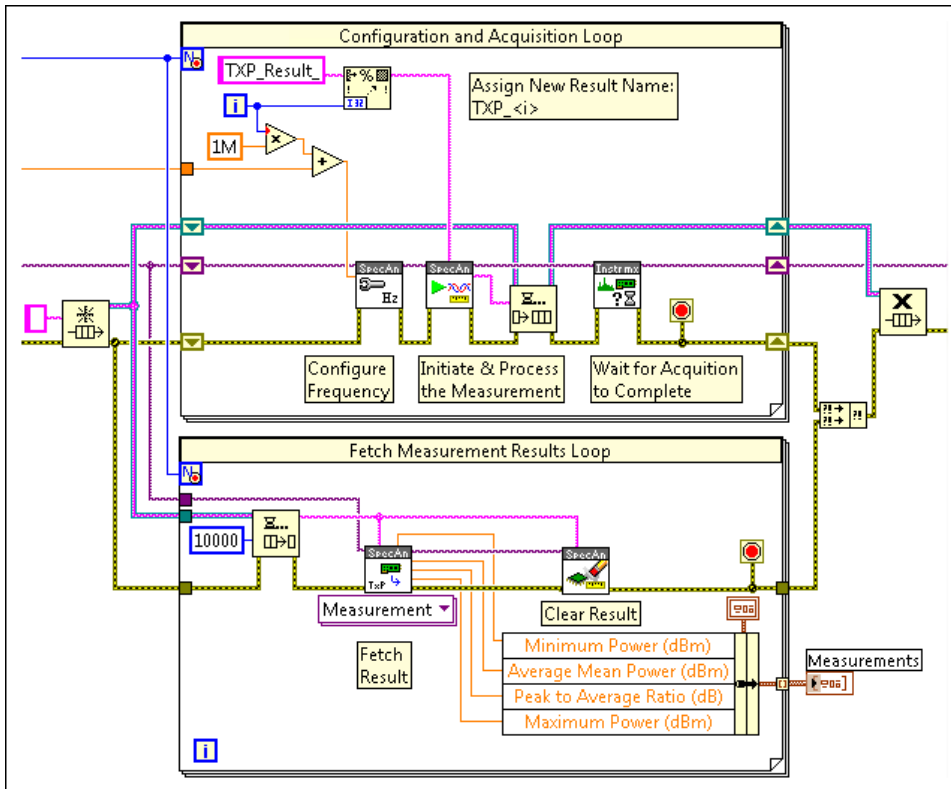
The following figure shows how to use a property node to fetch a result using an arbitrary result name.



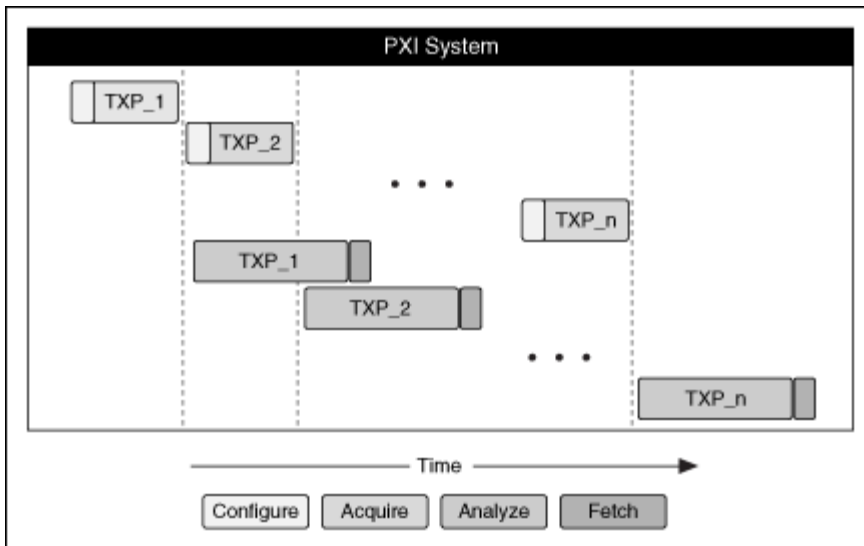
The benefit of the overlapped measurements technique increases with more measurements in your sequence. In fact, there is no limit on the number of measurements you can overlap. However, as the number of named results increases, consider deleting the measurement results that have been fetched to conserve process memory space. The named results can be deleted using the RFmxSpecAn Clear Named Result VI.

After a named result is cleared, all the measurement results for that Named Result are erased from the process memory of your application. If all named results must be cleared from the memory, call the RFmxSpecAn Clear All Named Results VI.

The following example program conducts multiple TxP measurements at different frequencies and clears the named result for TxP measurement after it has been fetched.



The following figure demonstrates the sequence of events for the previous TxP measurement example.



The actual time it takes to acquire the samples and analyze the TxP measurements may vary based on your system performance and measurement settings; however, the sequence of events, such as, configuring, acquiring data, and so on will remain the same. RFmx automatically starts the analysis or processing of the measurement immediately after the samples for the measurement have been acquired. Once the processing of the measurement is complete, you can fetch measurement results.

Depending on the computational complexity of the measurement, the overlapped measurements technique can have a huge impact on overall test time. Measurement parallelism is a known technique to reduce test time; RFmx makes this easier by managing threads and results, so you can develop and realize the benefits of PXI faster.

Using FPGA Extensions with RFmx

To use NI-RFSA instrument driver FPGA extensions with RFmx, you must install NI-RFSA along with RFmx and the latest version of LabVIEW. When you install NI-RFSA, a model-specific folder for your device is created in the following location:

- Users\Public\Documents\National Instruments\FPGA Extensions Bitfiles

For RFmx to recognize your precompiled bitfile, ensure that the precompiled bitfile is located in the model-specific folder that corresponds to your device.



Note The PXIe-5624 is supported by RFmx only as part of the PXIe-5668 VSA. Refer to the ***NI IF Digitizers Help***, located at ni.com/manuals, for more information about using the PXIe-5624 as a stand-alone device.

PXIe-5668 Pre-installed FPGA Bitfiles

NI-RFSA includes two FPGA bitfiles for use with the PXIe-5668 in a separate directory from the directories listed in the previous section. The PXIe-5668 default 400 MHz bitfile and the optional 800 MHz bitfile are located in the following directory:

- `ProgramData\National Instruments\FPGA Extensions
Bitfiles\NI PXIe-5624`

Loading an FPGA Bitfile

To use RFmx with NI-RFSA instrument driver FPGA extensions, you must specify the precompiled bitfile you want to use when you initialize the RFmx session. If you want to access additional FPGA functionality provided by the precompiled bitfile, you must also obtain an FPGA reference to use with the LabVIEW FPGA host interface.



Note You can load only one bitfile on a device at a time. If you are using both RFmx and another device driver such as NI-RFSA or NI-RFSG to simultaneously access the same instrument, both sessions must initialize with the same bitfile.

Complete the following steps to open an instrument driver session that uses a precompiled bitfile:

1. Call the RFmxInstr Initialize NIRFSA VI.
2. Wire the RFmx Setup string to the **Option String** input.
3. Include the Bitfile tag in the RFmx Setup string, and set the value of the Bitfile tag to the name of the bitfile you want to load, as shown in the following example string:

```
RFmxSetup=Bitfile:filename.lvbitx
```

where filename.lvbitx is the filename of the bitfile you want to load.

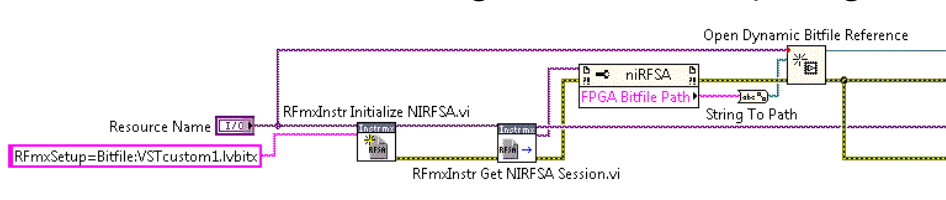
Obtaining an FPGA Reference

Complete the following steps to obtain an FPGA reference to use with the LabVIEW FPGA host interface:

1. Obtain the **NI RFSA Session** using RFmxInstr Get NIRFSA Session VI.
2. Read the **FPGA Bitfile Path** property using the NI-RFSA Session. The FPGA Bitfile Path property returns the absolute path to the bitfile as a string.
3. Wire the output of the FPGA Bitfile Path property to the **String** input of the String to Path function.
4. Wire the **Path** output of the String to Path function to the **Bitfile Path** input of the Open Dynamic Bitfile Reference function.

The following figure illustrates how the code might look after completing the

preceding steps.



RFmx Analysis-Only Mode

The RFmx Analysis-Only Mode further extends the functionality and flexibility of the RFmx API to handle the following use cases:

- Perform RF measurements with hardware that RFmx does not natively support
- Perform RF measurements on simulated waveforms
- Perform RF measurements on waveform data that was acquired previously and saved to disk

In the RFmx Analysis-Only Mode, time domain IQ data or spectrum data is passed to the RFmx Analyze VI that performs the RF measurements that you specify on the raw waveform data. In LabVIEW, the RFmx Analyze VI is located under the Advanced palette for each RFmx driver.

Time domain measurements such as ModAcc/EVM or CCDF only accept complex I/Q data. Spectral measurements such as ACP, SEM, or CHP accept either complex I/Q data

or spectrum data which must be in units of dBm. The f_0 in the spectrum data passed to the RFmx Analyze VI is calculated using the following formula:

$$f0_rfmx = f0_driver - fc$$

where,

$f0_rfmx$ is start frequency of the spectrum passed to the RFmx Analyze Spectrum VI

fc is the center frequency of spectral acquisition

$f0_driver$ is the start frequency of the spectrum

In addition, if hardware is used to acquire the data, the input data provided to the Analyze VI must meet or exceed the minimum requirements of the RFmx recommended acquisition settings.

For example, the sample I/Q rate can be higher than recommended sample rate, acquisition length can be longer than the recommended acquisition length, spectrum span can be wider than the recommended acquisition span, and so on.

The recommended acquisition settings are calculated based on the user-defined measurement configuration, and can be accessed through the RFmxInstr property node in LabVIEW or by getting the values of the equivalent attributes in C/.NET.

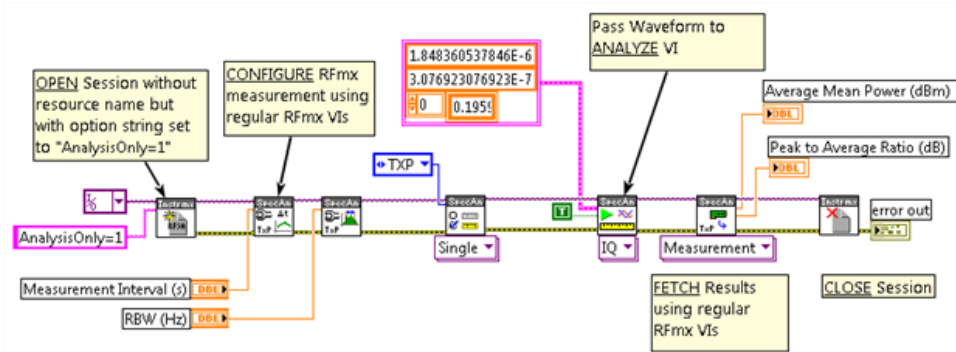
When using the RFmx Analysis-Only Mode, the RFmx driver ignores RFmx hardware settings such as reference level and attenuation. The only RF hardware setting that is not ignored is the center frequency, since it is needed for spectral measurement traces as well as some of the wireless standard-specific measurements, such as ACP and SEM, which define different configurations or limits based on the center frequency and the value which you set for the Band property.

The following image shows the typical programming flow for performing the time-domain transmit power (TxP) measurement on a simulated QAM waveform using the following operations in the Analysis-Only Mode:

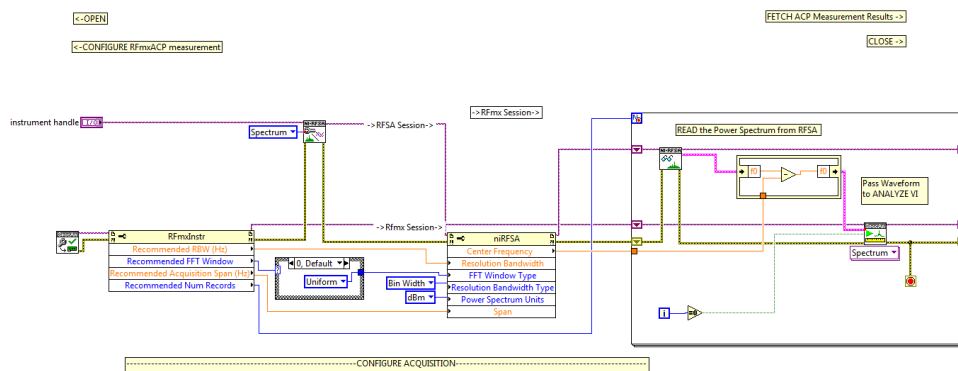
- **OPEN**—Opens the RFmx session with an empty hardware resource name and with “AnalysisOnly=1” string wired to option string input of the RFmxInstr

Initialize.vi.

- **CONFIGURE**—Configures the TxP measurement as if RFmx was used with hardware.
- **ANALYZE**—Passes a simulated waveform to the Analyze VI which is called instead of Initiate VI that is used when using RFmx with the built-in hardware support.
- **FETCH**—Fetches the measurement results using the same RFmx VIs used to fetch results while using RFmx with the built-in hardware support.
- **CLOSE**—Closes the RFmx session.



Another way to use the RFmx Analysis-Only Mode is in combination with another driver, such as NI-RFSA, that acquires the waveform which is then passed to RFmx for analysis. In this case, the waveform acquisition, performed by the driver, is configured based on the recommended acquisition settings calculated by RFmx. The following example demonstrates this approach.



The programming flow for using the RFmx Analysis-Only Mode in combination with NI-RFSA closely resembles the flow used when analyzing a simulated waveform using RFmx. However, in addition to opening and configuring RFmx session, you can also open and configure an NI-RFSA session.

NI-RFSA performs the actual acquisition of the waveform based on the recommended acquisition settings that are passed on from RFmx session to RFSA session using property nodes (LabVIEW) or setting the appropriate attribute values (C/.NET). You must call the RFmx Commit VI to calculate the recommended acquisition settings based on the user-defined RFmx measurement configuration before recommended acquisition settings are passed on to the RFSA.

After NI-RFSA acquires the waveform, RFmx Analyze VI is called to perform user-defined measurements on the raw waveform data. Once the analysis of the waveform is complete, you can fetch measurement results and close both RFmx and RFSA sessions.

Refer to the following examples for more information about using the RFmx Analysis-Only Mode to perform RF measurements:

- `<LabVIEW directory>\examples\RFmx\SpecAn\RFmxSpecAn ACP (Analysis Only).vi`
- `<Users>\Public\Documents\National Instruments\RFmx\SpecAn\Examples\C\RFmxSpecAnAcpAnalysisOnly`
- `<Users>\Public\Documents\National Instruments\RFmx\SpecAn\Examples\DotNET\VS2010\RFmxSpecAnAcpAnalysisOnly`



Note

- Measurements such as harmonics and spurious emissions that require multiple acquisitions with different configurations are not supported in the Analysis-Only Mode. The Dynamic Range ACP measurement method is also not currently supported.
- For the Analysis-Only mode, you need to adjust the t_0 value of the I/Q waveforms according to pre-trigger samples present in the waveform before passing the waveform to the RFmx Analyze (IQ) VI.



Note Use the following recommended settings while analyzing the LTE waveforms in the Analysis-Only mode:

- If the waveform data has been acquired with local oscillator (LO) at the center of RF signal bandwidth, set the the value of Acq BW Optimization Enabled property to **True**; otherwise, set the value of this property to **False**.
- If you set the ModAcc Common Clock Source Enabled property to **True**, the value of the Center Freq property configured in RFmxLTE session should be same as the center frequency used to acquire the waveform from the hardware.

External Attenuation Table

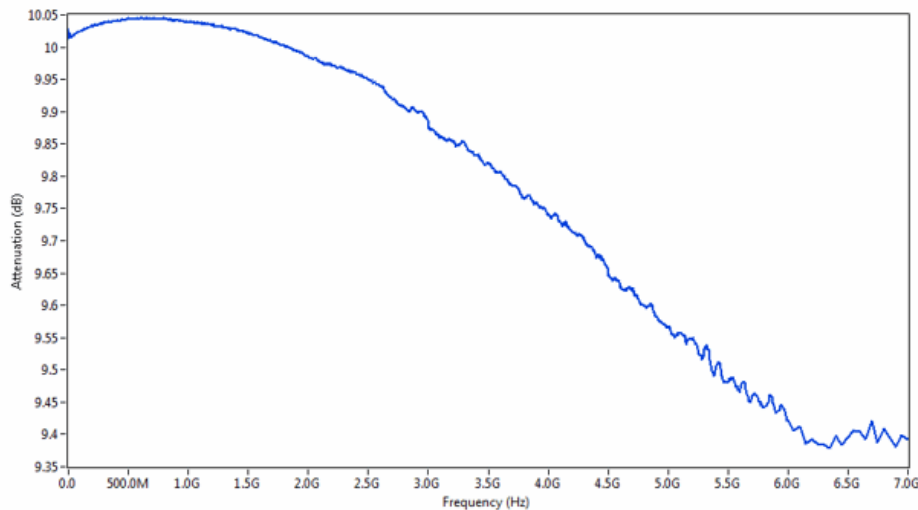
External Attenuation Table

The spectrum measurement allows you to view a signal in the frequency domain within the frequency span that you have configured.

Accuracy of the spectral measurements performed on a radio frequency (RF) signal is limited by several random and systematic sources of error [1]. Examples of random sources of error are, Additive White Gaussian Noise (AWGN) [2], quantization noise [3] etc. Examples of systematic sources of error are; incorrect or outdated signal or spectrum analyzer calibration, impedance mismatches of RF ports, and uncompensated frequency response of external RF components like cables, power splitters, switches, attenuators etc.

Impact of random sources of error can generally be reduced by measurement averaging, but systematic errors must be explicitly compensated.

The following illustration shows the frequency response of a 10 dB RF attenuator. Note that the attenuation is not exactly 10 dB for all frequencies, introducing systematic errors in measurements.



Frequency Response of 10dB RF Attenuator

External Attenuation Table in RFmx

In RFmx, to compensate for systematic errors due to frequency response of RF components in spectral measurements, **External Attenuation (dB)** and the corresponding **Frequency (Hz)** parameters should be specified as a table.

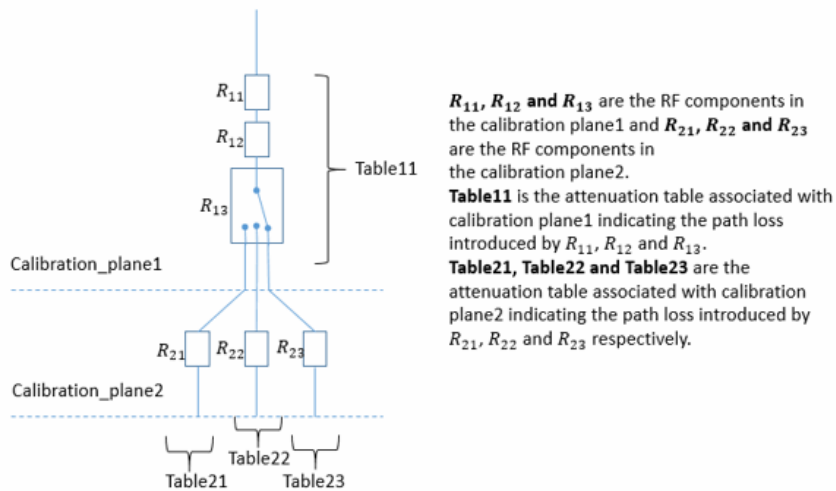
If the RF signal is attenuated by the RF components in the signal path, you need to enter a positive value of the **Attenuation (dB)** in the table. Similarly, if the RF signal is amplified by the RF components in the signal path, you need to enter a negative value.

Specifying Calibration Planes and External Attenuation Tables

Every external attenuation table should be associated with a unique string, the **Table Name**.

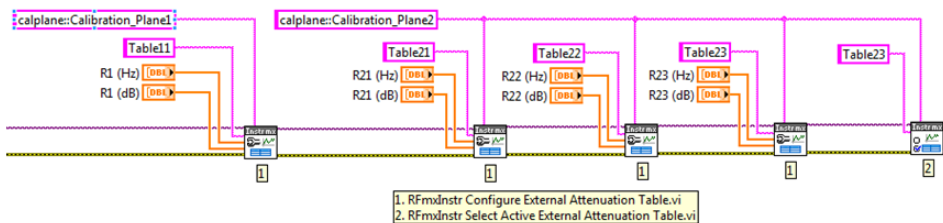
The signal path can consist of one or more calibration planes, where each calibration plane represents one or a group of RF components. In a calibration plane that is indicated by the selector string, you can configure multiple external attenuation tables, but you can select only one **Active Table** per calibration plane.

If you do not mention a **Table Name** and a **Selector String** to be used for the calibration plane, a default is used.



Illustrative example of concept of calibration plane

The configuration of the external attenuation table and the selection of the Active Table in the calibration plane can be done as follows for the preceding example.



Configuration of Tables and selection of an active table in calibration plane

If the Active Table does not contain an attenuation value for a particular frequency, it is calculated by linearly interpolating the available entries in the table. For frequency values lesser than the minimum frequency specified in the table, attenuation value corresponding to the minimum frequency in the table is used. Similarly, for frequency values greater than the maximum frequency specified in the table, attenuation value corresponding to the maximum frequency in the table is used.

A single table configured on the calibration plane makes the table active by default.

Amplitude Correction Type

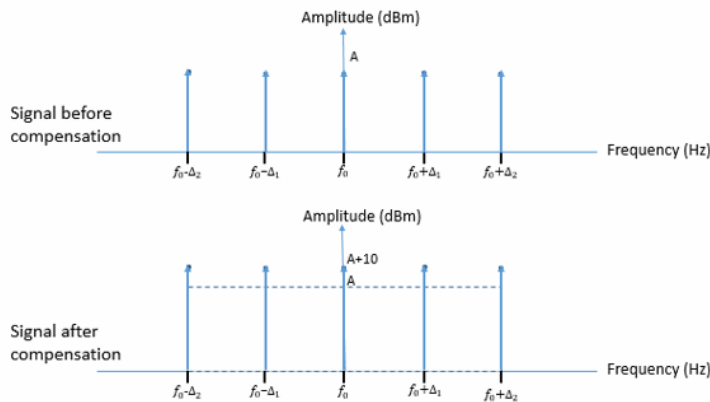
There are two ways of compensating the acquired spectrum in all spectral measurement; RF Center Frequency, and the Spectrum Frequency Bin.

RF Center Frequency

Compensates the entire acquired spectrum with a constant attenuation value that is obtained by interpolating all the active tables for the RF acquisition frequency. This is the default amplitude correction type.

Active External Attenuation Table

Frequency (Hz)	$f_0 - \Delta_2$	$f_0 - \Delta_1$	f_0	$f_0 + \Delta_1$	$f_0 + \Delta_2$
Attenuation (dB)	11	10.5	10	10.3	10.2



Compensation of the acquired spectrum when Amplitude Correction Type is set to RF Center Frequency and f_0 is the acquisition frequency

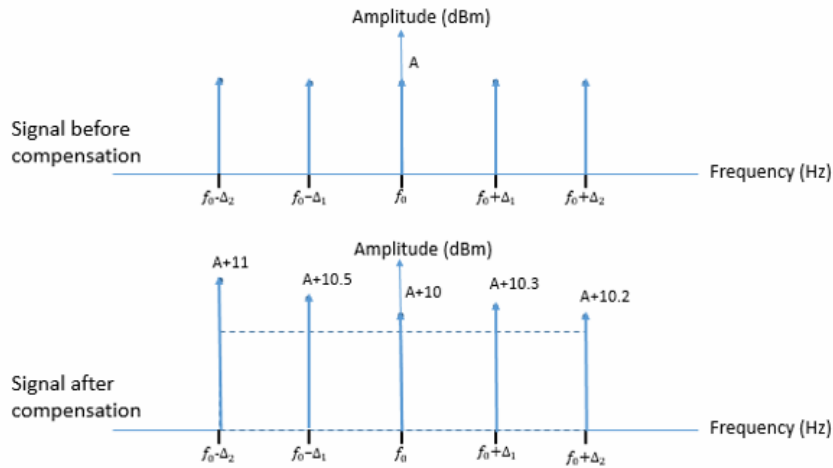
In case of measurements which are required to acquire at multiple frequencies like IM, Spur, Harmonics and ACP with the Measurement Method property set to **Dynamic Range**, the external attenuation corresponding to that acquisition frequency is used in every segment.

Spectrum Frequency Bin

Compensates all frequency points in the acquired spectrum with the respective attenuation values obtained by interpolating all active tables.

Active External Attenuation Table

Frequency (Hz)	$f_0 - \Delta_2$	$f_0 - \Delta_1$	f_0	$f_0 + \Delta_1$	$f_0 + \Delta_2$
Attenuation (dB)	11	10.5	10	10.3	10.2



Compensation of the acquired spectrum when Amplitude Correction Type is set to Spectrum Frequency Bin

References

- [1] Taylor, John. Introduction to error analysis, the study of uncertainties in physical measurements. 1997
- [2] Johnson, John Bertrand. "Thermal agitation of electricity in conductors." Physical review 32.1 (1928): 97
- [3] Gray, Robert M., and David L. Neuhoff. "Quantization." IEEE transactions on information theory 44.6 (1998): 2325-2383

Standard Error Parameter Behavior

Many LabVIEW nodes contain `error in` or `error out` parameters that you can use to manage errors. These parameters typically provide the same, standard functionality. When a node exhibits different parameter functionality, the exceptions are documented in the reference material for that node.

Standard behavior for `error in` and `error out` is described in the following table.



Note Some nodes, such as error handling VIs, contain an `error in`

parameter that does not provide standard error in functionality, but that contains an `error in` cluster that is standard.

	<code>error in</code> describes error conditions that occur before this node runs. The default is no error. If an error occurs before this node runs, the node passes the <code>error in</code> value to <code>error out</code>. This node runs normally only if no error occurs before this node runs. If an error occurs while this node runs, it runs normally and sets its own error status in <code>error out</code>. Use <code>error in</code> and <code>error out</code> to check errors and to specify execution order by wiring <code>error out</code> from one node to <code>error in</code> of the next node.
	<code>error out</code> contains error information. If <code>error in</code> indicates that an error occurs before this node runs, <code>error out</code> contains the same error information. Otherwise, it describes the error status that this node produces. Right-click the <code>error out</code> front panel indicator and select Explain Error from the shortcut menu for more information about the error.

`error in` cluster contains the following cluster elements:

	<code>status</code> is TRUE if an error occurs before this node runs or FALSE to indicate a warning or that an error did not occur before the node runs. The default is FALSE.
	<code>code</code> is the error or warning code. The default is 0. If <code>status</code> is TRUE, <code>code</code> is an error code. If <code>status</code> is FALSE, <code>code</code> is 0 or a warning code.
	<code>source</code> specifies the origin of the error or warning, and in most cases is the name of the node that produces the error or warning. The default is an empty string.

`error_out` contains the following cluster elements:

	<code>status</code> is TRUE if an error occurs before or while this node runs, or FALSE to indicate a warning or that no error occurred before or while this node runs.
	<code>code</code> is the error or warning code. If <code>status</code> is TRUE, <code>code</code> is an error code. If <code>status</code> is FALSE, <code>code</code> is 0 or a warning code.
	<code>source</code> specifies the origin of the error or warning, and in most cases is the name of the node that produces the error or warning.

RFmx Overview CVI

RFmx is a family of instrument drivers that share a single session handle across multiple drivers. You must open one RFmx session per signal analyzer and use that single session handle to configure and perform measurements with the different drivers, such as SpecAn, Demod, LTE, and so on.

The RFmx driver has a shared set of API functions for opening and closing the RFmx session as well as configuring hardware settings, such as the Reference Clock configuration, that are shared across drivers. These shared functions are called the RFmxInstr driver. RFmxInstr does not contain any measurements, and you always use RFmxInstr along with one or more other RFmx drivers that expose the measurements you need to perform.

Some of the commonly used RFmxInstr-related APIs are replicated in other RFmx drivers. For example, RFmxSpecAn has an Initialize function, which in turn calls the Initialize function of RFmxInstr, so you can use the Initialize function of RFmxSpecAn to create the RFmx session. Similarly, some of the other commonly used APIs in RFmxInstr are replicated in the other drivers.

The following is an example of how you can use the RFmx API drivers in C to capture a spectrum using the RFmx SpecAn driver. The same sequence applies to other ADEs as well.

```

RFmxInstr_Initialize(resourceName, "", &instrumentHandle, NULL); //Step 1
RFmxInstr_CfgFrequencyReference(instrumentHandle, "", frequencyReferenceSource,
frequency); //Step 2
RFmxSpecAn_CfgRF(instrumentHandle, "", centerFrequency, referenceLevel,
externalAttenuation); //Step 3
RFmxSpecAn_SpectrumCfgSpan(instrumentHandle, "", span);
RFmxSpecAn_SelectMeasurements(instrumentHandle, "", RFMXSPECAN_VAL_SPECTRUM,
RFMXSPECAN_VAL_TRUE); //Step 4
RFmxSpecAn_Initiate(instrumentHandle, "", ""); //Step 5
RFmxSpecAn_SpectrumFetchSpectrum(instrumentHandle, "", timeout, &x0, &dx, spectrum,
arraySize, NULL); //Step 6
RFmxInstr_Close(instrumentHandle, RFMXSPECAN_VAL_FALSE); //Step 7

```

1. Open an RFmx session using RFmxInstr.
2. Configure the frequency reference using the configure functions or properties in RFmxInstr.
3. Configure the reference level, frequency, and span using RFmxSpecAn.
4. Select the spectrum measurement using RFmxSpecAn.
5. Initiate the measurement using RFmxSpecAn.
6. Fetch the spectrum using RFmxSpecAn.
7. Close the RFmx session using RFmxInstr or RFmxSpecAn.

When you use RFmxInstr functions to configure hardware settings in an RFmx session, those settings apply to all the RFmx drivers that are present in the same RFmx session. However, the following settings are RFmx driver-specific and are not shared across different RFmx drivers within the same RFmx session:

- Center frequency
- Reference level
- External attenuation
- Trigger settings

For example, if you are making both RFmx LTE and RFmx WCDMA measurements within one RFmx session, setting the frequency in RFmx LTE does not affect the frequency used in RFmx WCDMA.

Selector String (C/LabWindows™/CVI™)

RFmx is a powerful API that supports basic and advanced features and provides

multiple ways to configure measurements and fetch measurement results. You can use the same configuration and fetch functions in basic and advanced use cases, such as named signal configurations, without having to rewrite code. The following code shows a basic ACP measurement that does not use named signal configurations.

```
/* Initialize a session */
RFmxSpecAn_Initialize(resourceName, "", &instrumentHandle, NULL);
/* Configure ACP measurement */
RFmxSpecAn_CfgRF(instrumentHandle, "", centerFrequency, referenceLevel,
externalAttenuation);
RFmxSpecAn_ACPCfgAveraging(instrumentHandle, "", averagingEnabled, averagingCount
averagingType);
RFmxSpecAn_ACPCfgCarrierAndOffsets(instrumentHandle, "", integrationBandwidth,
numberOfOffsetChannels, channelSpacing);
/* Retrieve results */
RFmxSpecAn_ACPRead(instrumentHandle, "", timeout, &carrierAbsPower,
&offCh0LowerRelativePower, offCh0UpperRelativePower, &offCh1LowerRelativePower,
&offCh1UpperRelativePower);
/* Close session */
RFmxSpecAn_Close(instrumentHandle, RFMXSPECAN_VAL_FALSE);
```

You can enhance the code as shown in the following code to use named signal configurations by adding an `RFmxSpecAn_CreateSignalConfiguration` function to create the signal and the `RFmxSpecAn_BuildSignalString` function to build a signal string in a required form and passing that signal string into the **selectorString** input of the functions that the basic ACP code is using.

```
/* Initialize a session */
RFmxSpecAn_Initialize(resourceName, "", &instrumentHandle, NULL);
/* Create a signal and build a signalString */
RFmxSpecAn_CreateSignalConfiguration(instrumentHandle, "MySignalConfig");
RFmxSpecAn_BuildSignalString("MySignalConfig", "", MAX_SELECTOR_STRING,
selectorString);
/* Configure ACP measurement */
RFmxSpecAn_CfgRF(instrumentHandle, selectorString, centerFrequency, referenceLevel,
externalAttenuation);
RFmxSpecAn_ACPCfgAveraging(instrumentHandle, selectorString, averagingEnabled,
averagingCount averagingType);
RFmxSpecAn_ACPCfgCarrierAndOffsets(instrumentHandle, selectorString,
integrationBandwidth, numberOfOffsetChannels, channelSpacing);
/* Retrieve results */
RFmxSpecAn_ACPRead(instrumentHandle, selectorString, timeout,
```

```
&carrierAbsPower, &offCh0LowerRelativePower, offCh0UpperRelativePower,
&offCh1LowerRelativePower, &offCh1UpperRelativePower);
/* Close session */
RFmxSpecAn_Close(instrumentHandle, RFMXSPECAN_VAL_FALSE);
```

The **selectorString** parameter in RFmx functions allows you to specify the full context of what you are configuring or fetching. The following code shows an example of the **selectorString** parameter on one of the RFmx fetch measurement functions.

```
int32 __stdcall RFmxSpecAn_ACPFetchCarrierMeasurement
(niRFmxInstrHandle instrumentHandle, char selectorString[],
float64 timeout, float64* absolutePower, float64*
totalRelativePower, float64* carrierOffset, float64*
integrationBandwidth);
```

For example, an ACP measurement can have multiple offset channels, and you can configure the frequency and measurement bandwidth properties for each offset channel. When you configure the measurement bandwidth for a specific offset channel, you must specify the offset channel for which you are configuring the Measurement Bandwidth attribute. When you configure the first offset channel, you must specify the context in the **selectorString** input as “offset0”, as shown in the following code.

```
/* Fetch ACP offset measurements */

RFmxSpecAn_ACPFetchOffsetMeasurement (instrumentHandle,
"offset0", timeout, &lowerRelativePower,
&upperRelativePower, &lowerAbsolutePower,
&upperAbsolutePower);
```

The contexts you can specify depend on what you are configuring or fetching. For example, if you are configuring a carrier channel, you specify which carrier channel, such as “carrier3”, you are configuring. If you are configuring an offset channel, you specify which offset channel, such as “offset2”, you are configuring.

Some contexts, such as a named signal configuration, can be specified in all configuration and fetch functions. Some contexts, such as a named result, can be specified in only the initiate and fetch functions.

Multiple Contexts

When using multiple features in the API, you can specify multiple contexts in the same **selectorString** input separated with a '/'. For example, to configure carrier 0 in the named signal configuration called "MySignal", you must specify "signal::Mysignal/carrier0" as the selector string.

Another example of multiple contexts is when performing LTE non-contiguous carrier aggregation measurements. In this case the signal is composed of multiple subblocks and each subblock can contain multiple carriers. When you configure a property of a carrier in LTE non-contiguous carrier aggregation, you must specify the index of the carrier within the subblock and the index of the subblock. For example, you would specify that you are configuring the first carrier in the second subblock by using "subblock1/carrier0" as the selector string.

String Prefixes in Selector Strings

RFmx requires a prefix, such as "carrier" or "signal::" before the number or name in the selector string so that it is clear what is being configured. The "::" is required for "signal::" and "result::" because those items use arbitrary user-defined names and the API must identify where the user specified name starts in the selector string. You can specify the selector strings as constants, such as "carrier0" and "signal::mysignal", or use the build string utility functions provided by RFmx to create correctly formed strings.

Build String Utility Functions

RFmx provides a set of utility functions as shown below to build context string for various measurement configurations.

- RFmxSpecAn_BuildSignalString
- RFmxSpecAn_BuildCarrierString2
- RFmxSpecAn_BuildHarmonicString2
- RFmxSpecAn_BuildOffsetString2
- RFmxSpecAn_BuildSpurString2
- RFmxSpecAn_BuildRangeString2
- RFmxSpecAn_BuildMarkerString2

For example, the `BuildSignalString` function takes in a named signal configuration name and/or a result name and produces a string containing the contexts you specified. If you call the `BuildSignalString` function with “MySignal” passed to the **signalName** input, the output string is “signal::MySignal”. Each RFmx driver provides build string utility functions for specific cases, such as `BuildRangeSpurString`, which takes in a range number and spur number, in addition to optionally taking in a previously constructed selector string. If you call the `BuildSpurString2` function with a spur set to 2, and the **selectorString** input set to the output of the previously called `BuildRangeString2` function, say `range1`, the output string is “signal::MySignal/range1/spur2”.

```
int32 __stdcall RFmxSpecAn_BuildSpurString2 (char
selectorString[], int32 spurNumber, int32
selectorStringOutLength, char selectorStringOut[]);
```

Selector String in Set or Get Property Functions

When setting properties from an RFmx property set or get function in C, you specify the context using the `selectorString` parameter.

Selector Strings when using Array Functions

Notice that RFmx provides array versions of configure and fetch functions for many cases that allow you to configure multiple instances of a property or fetch multiple instances of a result. For example, to configure the frequencies of all the ACP measurement offset channels, you can use `ACPCfgOffsetArray` passing in an array that has the frequency for each of the offset channels. When using these array functions, you do not need to specify in the **selectorString** input the context for the elements in the array. For example, you pass "" as the **selectorString** in `ACPCfgOffsetArray`. However, if you are using named signal configurations or named results, you pass those contexts in the selector string even when you use the array functions.

Selector String Syntax

The following table lists the various selector string formats that you can use. For integral channel specifiers, valid values for *i* are 0 to (number of channels - 1). For example, if you set the value of the

`RFMXSPECAN_ATTR_ACP_NUMBER_OF_OFFSETS` attribute to 5, you can configure

ACP offset attributes for offset0, offset1, offset2, offset3, and offset4.

Selector String	String Function	Example
ch<i>	Configures one channel	carrier0
ch<i>-<j>	Configures a range of channels between i and j	carrier0-5
ch<i>:<j>	Configures a range of channels between i and j	carrier0:5
ch<i>, ch<j>, ch<k>	Configures channels i, j, and k	carrier0, carrier2, carrier4
ch<i>, ch<j>-<k>	Configures channel i and a range of channels between j and k	carrier0, carrier1-5
ch<i>, ch<j>:<k>	Configures channel i and a range of channels between j and k	carrier0, carrier1:5
ch::all	Configures all channels from 0 to (Number of X - 1). Number of X is the current value of the Number of ... property that corresponds to the context being used. For example, in an ACP measurement, you set the Number of Offsets property before setting any offset properties and when you use “all”, RFmx uses the current value of Number of Offsets to determine which offset channels to configure.	offset::all

The following table lists the various selector string formats that you can use to specify the named signal configurations, using the “signal::” prefix, and named results, using the “result::” prefix. Notice that the “result::” prefix is valid only when fetching results and currently cannot be used to configure measurements.

Syntax	String Function	Example
signal::<signalname>	Specifies signal <signalname>	signal::sig1
signal::<signalname>/ch<i>	Specifies channel i on signal <signalname>	signal::sig1/ carrier0
signal::<signalname>/result::<resultname>/ch<i>	Specifies channel i for result <resultname> on signal <signalname>	signal::sig1/ result::r1/ carrier0
signal::<signalname>/result::<resultname>	Specifies named result <resultname> on signal	signal::sig1/ result::r1

Syntax	String Function	Example
	<signalname>	
result::<resultname>	Specifies named result <resultname> on the default signal	result::r1

The following characters are not allowed in signal names and result names: ()*+,-./{} !"#\$%&'::<=>?@[]\^`|~ and any non-printable character.

Related concepts:

- [Signal Configurations](#)
- [Overlapped Measurements and Named Results](#)
- [Array Configuration and Fetch Functions](#)

Array Configuration and Fetch Functions

Array Configuration Functions

RFmx provides array-based configuration functions to enable you to set different values to multiple instances of an attribute in a single call to the function. The first N instances of each attribute are configured to the values specified in the corresponding elements of the array. If you pass an empty array to the input of a function, none of the instances of the attribute corresponding to that input are modified. You can pass up to M values in the array, where M is the current value of the <Number of X> attribute (for example, Number of Carriers, Number of Harmonics, Number of Offsets, and so on). If the array size, N, is smaller than M, only the first N instances of the attribute are modified.

For example, if you set the `RFMXSPECAN_ATTR_ACP_NUMBER_OF_OFFSETS` attribute to 3, and if the **offsetSideband** parameter of the `RFmxSpecAn_ACPCfgOffsetArray` function has two elements in the array, the `RFMXSPECAN_ATTR_ACP_OFFSET_SIDE BAND` attribute for the first two offset channels are set to the values in the array, and the `RFMXSPECAN_ATTR_ACP_OFFSET_SIDE BAND` attribute for the third offset channel is not modified.

In the following code, **offsetReferenceCarrier** and **offsetReferenceSpecific** should be of same size. In this case, the array size is equal to the value of the **numberOfOffsets** parameter.

```
RFmxSpecAn_ACPCfgNumberOfOffsets (instrumentHandle, "", numberOfOffsets);
RFmxSpecAn_ACPCfgOffsetPowerReferenceArray (instrumentHandle, "",
offsetReferenceCarrier, offsetReferenceSpecific, numberOfOffsets);
```



Note For array configuration functions that have multiple array inputs, you must pass identically sized arrays to each input.

To set the values to a subset of attribute instances, you can use the non-array based functions. In the non-array based functions, you can configure one or more instances of a property to the same value in each call to the function.

Array Fetch Functions

RFmx provides array-based fetch functions to enable you to fetch multiple instances of a result in a single call to the function. For array-based Fetch functions, you should pass array buffers that have a size greater than or equal to the value of the **arraySize** parameter. You can use array fetch functions in the below shown way to query the size of the arrays.

```
RFmxSpecAn_ACPFetchOffsetMeasurementArray
(instrumentHandle, "", timeout, NULL, NULL, NULL, NULL, 0,
&actualArraySize);
```

In the above function call, the variable **actualArraySize** is the size of the arrays returned by the fetch function.

Array fetch functions return arrays containing the values for all instances of the results during a specific call to a measurement. For example, if you have configured an ACP measurement to measure three offset channels, the array contains the results for these three offset channels. In the following code, the size of the **lowerRelativePower**, **upperRelativePower**, **lowerAbsolutePower**, and **upperAbsolutePower** parameters are equal to the value of the **actualArraySize** parameter.

```
lowerRelativePower = (float64 *)malloc(sizeof(float64)*actualArraySize);
upperRelativePower = (float64 *)malloc(sizeof(float64)*actualArraySize);
lowerAbsolutePower = (float64 *)malloc(sizeof(float64)*actualArraySize);
```

```
upperAbsolutePower = (float64 *)malloc(sizeof(float64)*actualArraySize);
RFmxSpecAn_ACPFetchOffsetMeasurementArray (instrumentHandle, "", timeout,
lowerRelativePower, upperRelativePower, lowerAbsolutePower, upperAbsolutePower,
actualArraySize, NULL);
```

Signal Configurations

RFmx allows you to create measurement configurations called signal configurations, within the same personality. Each signal configuration instance holds a unique instance of signal and measurement property values. By using multiple signal configurations, you can switch between different measurement configurations without the overhead of configuring the measurements each time. Signal configurations also allow you to execute multiple configurations of the same measurement at the same time.

RFmx creates one default signal configuration for each personality. This default configuration is unnamed. You can create additional named signal configurations.

Select the signal configuration to use by passing the signal configuration name in the `RFmxSpecAn_Initiate` function **selectorString** parameter or the initiate function of any other RFmx driver. RFmx performs the measurements that are enabled in that signal configuration. If you pass multiple signal configuration names, such as, `signal::sig1`, `signal::sig2`, and so on, to the RFmx Initiate function, RFmx performs the measurements that are enabled in each of the signal configurations. The following signal properties must be configured for each signal configuration, otherwise their default values are used:

- Center frequency
- Reference level
- External attenuation
- Trigger settings

The following code shows the creation of two signals, named `LTE-5MHZ` and `LTE_20MHz`, in RFmx LTE. The two signal configurations use different center frequencies and bandwidths. The `LTE_5MHz` signal configuration is initiated before the `LTE_20MHz` signal configuration.

```
// Create a new RFmx Session
RFmxLTE_Initialize(resourceName, "", &instrumentHandle, &isNewSession);
//Configure reference frequency and source
RFmxLTE_CfgFrequencyReference(instrumentHandle, "", frequencySource, frequency);
//Create Signal Configurations
RFmxLTE_CreateSignalConfiguration(instrumentHandle, "LTE_5MHz");
RFmxLTE_BuildSignalString("LTE_5MHz", "", SELECTOR_STRING_LEN,
LTE_5MHzSignalString);
//LTE_5MHzSignalString =
"signal::LTE_5MHz" RFmxLTE_CreateSignalConfiguration(instrumentHandle,
"LTE_20MHz");
RFmxLTE_BuildSignalString("LTE_20MHz", "", SELECTOR_STRING_LEN,
LTE_20MHzSignalString);
//LTE_20MHzSignalString = "signal::LTE_20MHz"
//Configure Settings (Frequency, Bandwidth) for each Signal Configurations and
Enable CHP measurement
RFmxLTE_CfgFrequency(instrumentHandle, LTE_5MHzSignalString,
centerFrequency_LTE_5MHz);
RFmxLTE_CfgFrequency(instrumentHandle, LTE_20MHzSignalString,
centerFrequency_LTE_20MHz);
RFmxLTE_CfgComponentCarrier(instrumentHandle, LTE_5MHzSignalString,
carrierBandwidth_LTE_5MHz,
componentCarrierFrequency, cellID);
RFmxLTE_CfgComponentCarrier(instrumentHandle, LTE_5MHzSignalString,
carrierBandwidth_LTE_20MHz,
componentCarrierFrequency, cellID);
RFmxLTE_SelectMeasurements(instrumentHandle, "signal::LTE_5MHz,signal::LTE_20MHz",
RFMXLTE_VAL_CHP, RFXMLTE_VAL_TRUE);
//Initiate each Signal Configuration
RFmxLTE_Initiate(instrumentHandle, LTE_5MHzSignalString, "");
RFmxLTE_WaitForAcquisitionComplete(instrumentHandle, timeout);
RFmxLTE_Initiate(instrumentHandle, LTE_20MHzSignalString, "");
//Fetch Results
RFmxLTE_CHPFetchComponentCarrierMeasurement(instrumentHandle,
LTE_5MHzSignalString, timeout, &absolutePower_LTE_5MHz,
&relativePower_LTE_5MHz);
RFmxLTE_CHPFetchComponentCarrierMeasurement(instrumentHandle,
LTE_20MHzSignalString,
timeout, &absolutePower_LTE_20MHz,&relativePower_LTE_20MHz);
```



Note NI does not recommend using named signal configurations when only a few attributes are different between the configurations. For example, when performing the same measurement across different frequencies, NI recommends you use a single-signal configuration and change the center

frequency before each measurement.

The following characters are not allowed in signal name and result name: ()*+,-./{} !"#\$\$%&'";<=>?@[]\^`|~ and any non-printable character.

The following code shows an example of creating and deleting a signal name, “sig1”.

```
/*Creating a signal*/
RFmxSpecAn_CreateSignalConfiguration(instrumentHandle, "sig1");
/*Deleting the signal*/
RFmxSpecAn_DeleteSignalConfiguration(instrumentHandle, "sig1");
```

When you use signal configurations, you must provide a signal name that is correctly formatted to the RFmx Configure, Initiate, and Fetch functions. Use the following syntax for signal names.

signal::<signalname>

Example: signal::WCDMA

RFmx also provides a Build String Utility function, `RFmxSpecAn_BuildSignalString`, to create the formatted signal string. Refer to Build String functions for more information about all Build String Utility functions. Signal names are required for configure, initiate, and fetch functions.

The following code shows an example of creating a signal named “WCDMA_Band14” and calling a configure function with that signal name.

```
/* Configure ACP parameters for WCDMA signal analysis */
RFmxSpecAn_BuildSignalString("WCDMA_Band14", "", SIGNAL_STRING_LEN,
WCDMASignalString); // WCDMASignalString = "signal:: WCDMA_Band14"
RFmxSpecAn_CreateSignalConfiguration(instrumentHandle, WCDMASignalString);
RFmxSpecAn_CfgRF (instrumentHandle, WCDMASignalString, wcdmaCenterFrequency,
referenceLevel, externalAttenuation);
```

RFmx signal configurations can be cloned using the driver-specific RFmx Clone Signal Configuration API, such as the `RFmxSpecAn_CloneSignalConfiguration` function in RFmx SpecAn.

When you clone a configuration, a new signal configuration is created that inherits all the configuration settings of the signal configuration being cloned. In the new signal configuration, the original configuration settings can be overwritten by calling the appropriate signal configurations VIs with the new signal configuration name.

The following code shows an example of setting a property node using the signal name.

```
/* Configure ACP parameters for WCDMA signal analysis */
RFmxSpecAn_BuildSignalString("WCDMA_Band14", "", SIGNAL_STRING_LEN,
WCDMASignalString); RFmxSpecAn_CreateSignalConfiguration(instrumentHandle,
WCDMASignalString); RFmxSpecAn_SetCenterFrequency (instrumentHandle,
WCDMASignalString, wcdmaCenterFrequency);
```

Spectral Measurements Concepts

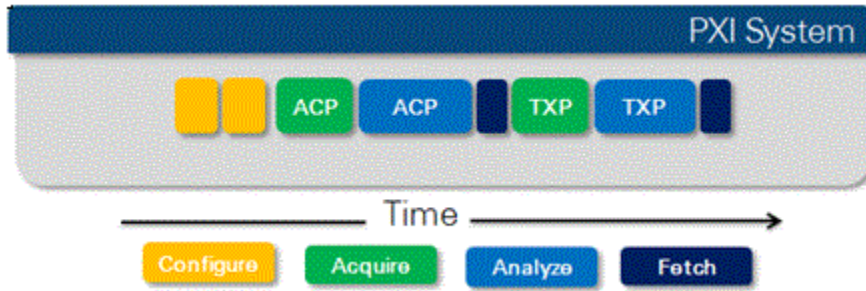
Composite measurement is a technique that allows multiple measurements to be performed in parallel on the same acquired data. This technique leads to faster measurement times since all the measurements in a composite set are based on the same acquired data, and they are processed in parallel threads on the CPU. Although the same data set is used for each measurement in a composite set, RFmx allows different configuration settings for each measurement, enabling multifaceted signal analysis.

The following code snippets show the difference between serial measurements and composite measurements.

Serial Measurement

```
RFmxSpecAn_Initialize (resourceName, "", &instrumentHandle, NULL);
RFmxSpecAn_CfgRF (instrumentHandle, "", centerFrequency, referenceLevel,
externalAttenuation);
RFmxSpecAn_TXPCfgRBWFilter(instrumentHandle, "", RBW , RBWFilterType, RRCAlpha);
RFmxSpecAn_ACPCfgCarrierAndOffsets(instrumentHandle, "", integrationBandwidth,
numberOfOffsetChannels, channelSpacing);
RFmxSpecAn_ACPRead(instrumentHandle, "", timeout, &carrierAbsolutePower,
&offCh0LowerRelativePower,
&offCh0UpperRelativePower, &offCh1LowerRelativePower, &offCh1UpperRelativePower);
RFmxSpecAn_TXPRead(instrumentHandle, "", timeout, &averageMeanPower,
&peakToAverageRatio, &maximumPower, &minimumPower);
```

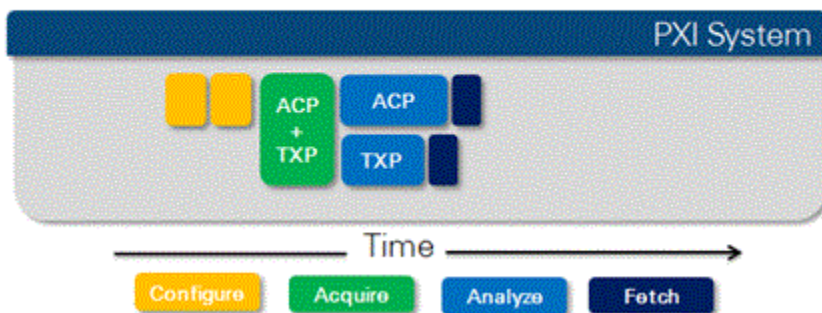
```
RFmxSpecAn_Close(instrumentHandle, RFMXSPECAN_VAL_FALSE);
```



Composite Measurement

```
RFmxSpecAn_Initialize (resourceName, "", &instrumentHandle, NULL);
RFmxSpecAn_CfgRF(instrumentHandle, "", centerFrequency, referenceLevel,
externalAttenuation);
RFmxSpecAn_TXPCfgRBWFilter(instrumentHandle, "", RBW, RBWFilterType, RRCAlpha);
RFmxSpecAn_ACPCfgCarrierAndOffsets(instrumentHandle, "", integrationBandwidth,
numberOfOffsetChannels, channelSpacing);
RFmxSpecAn_SelectMeasurements(instrumentHandle, "", (RFMXSPECAN_VAL_ACP |
RFMXSPECAN_VAL_TXP), RFMXSPECAN_VAL_TRUE); RFmxSpecAn_Initiate(instrumentHandle,
"", "");
RFmxSpecAn_ACPFetchOffsetMeasurementArray(instrumentHandle, "", timeout,
lowerRelativePowerArr,
upperRelativePowerArr, lowerAbsolutePowerArr, upperAbsolutePowerArr,
numberOfOffsetChannels, NULL);
RFmxSpecAn_TXPFetchMeasurement(instrumentHandle, "", timeout, &averageMeanPower,
&peakToAverageRatio, &maximumPower, &minimumPower);
RFmxSpecAn_Close(instrumentHandle, RFMXSPECAN_VAL_FALSE);
```

Multiple measurements on the same IQ data



When the composite measurements are performed, only a single acquisition is made, and each measurement in the composite set is performed on the same acquired data.

Internally, RFmx merges the acquisition settings from all the measurements in a composite set to provide the acquisition settings that satisfy the needs of all the measurements being performed. However, not all measurements can be performed as composite measurements. For example, a harmonics measurement cannot be performed in composite with an ACP measurement, since a harmonics measurement requires multisegment acquisitions for different harmonics.

To perform multiple measurements as composite measurements, use the `RFmxSpecAn_SelectMeasurements` function and pass the measurement types you want to perform together to the **measurements** parameter using a bit-wise OR operator.

Besides faster test execution times, composite measurements also allow you to see different measurements on the same DUT signal enabling you to debug difficult issues. For example, you can see if a bad ACP result is correlated to a bad EVM result.

Overlapped Measurements and Named Results

Many automated test applications require multiple independent measurements on a single instrument. Such applications include amplifier frequency/power sweeps or multi-DUT wireless device tests. Often engineers implement these applications by serially sequencing measurements. However, with RFmx, you can perform multiple measurements overlapped in time with a feature called overlapped measurements.

This feature can be used when measurements in a sequence are independent of previous results. The measurements must be performed on separate acquisitions because they require different hardware configurations, such as a different frequency or reference level.

Overlapped measurements maximize utilization of the RF hardware and enable parallel processing of the measurements, thus vastly reducing the overall test time for a sequence of multiple measurements. In the following conceptual example, two overlapped ACP measurements are performed at two different reference levels. This simple program completes the following operations:

- Configures the reference level for the first ACP measurement.
- Initiates the first ACP acquisition.
- Waits for the ACP acquisition to be complete and the PXI RF module to be released.

RFmx spawns a thread for ACP analysis in the background without delaying the next measurement.

- Initiates the second ACP acquisition before the first ACP measurement analysis is complete.
- Waits for the second ACP acquisition to be complete before the first ACP measurement analysis is complete.
- Fetches both the measurement results when their respective analysis threads are complete.

```
RFmxSpecAn_CfgReferenceLevel(instrumentHandle, "", referenceLevel1);
RFmxSpecAn_Initiate(instrumentHandle, "", "ACP_Results_1");
RFmxSpecAn_WaitForAcquisitionComplete(instrumentHandle, timeout);
RFmxSpecAn_CfgReferenceLevel(instrumentHandle, "", referenceLevel2s);
RFmxSpecAn_Initiate(instrumentHandle, "", "ACP_Results_2");
RFmxSpecAn_WaitForAcquisitionComplete(instrumentHandle, timeout);
RFmxSpecAn_BuildSignalString("", "ACP_Results_1", SELECTOR_STRING_LEN,
resultString1); //
resultString1 = "result::ACP_Results_1"
RFmxSpecAn_ACPFetchOffsetMeasurementArray(instrumentHandle, resultString1, timeout,
lowerRelativePowerArr1, upperRelativePowerArr1, lowerAbsolutePowerArr1,
upperAbsolutePowerArr1,
arraySize1, NULL);
RFmxSpecAn_BuildSignalString("", "ACP_Results_2", SELECTOR_STRING_LEN,
resultString2); //
resultString2 = "result::ACP_Results_2"
RFmxSpecAn_ACPFetchOffsetMeasurementArray(instrumentHandle, resultString2, timeout,
lowerRelativePowerArr2,
upperRelativePowerArr2, lowerAbsolutePowerArr2, upperAbsolutePowerArr2, arraySize2,
NULL);
```

The key feature that enables overlapped measurements in RFmx is a feature called Named Results. In the previous example, both the first and second ACP measurements are initiated and tagged with a name.

The RFmxSpecAn_Initiate function is called first with the ACP measurement enabled and the result name as ACP_Results_1, and then it is called again with the second ACP measurement enabled and the result name as ACP_Results_2. The results of these two measurements are stored independently, and can be fetched in any sequence after the measurements are complete using the same result names passed to the initiate function.

You can create the result name passed to the fetch function using a set of Build String Utility functions provided by RFmx. In the above example, RFmxSpecAn_BuildSignalString function is used to create a result string “result::ACP_Results_1” and passed to the Fetch function.

Measurement results are referenced by the result name, measurement type, and the signal instance. If no result name is specified, the default result name (“”) is used.

The following characters are not allowed in signal name and result name: ()*+,-./{} !"#\$%&'";<=>?@[]\^`|~ and any non-printable character.

When using overlapped measurements, you must specify the result name in Initiate and then use that name to fetch the results. If you reuse the same result name, the previous named results with that name are overwritten. This behavior applies to the default result name (“”) as well.

When you initiate a measurement, RFmx clears out all previous results with the same result name passed to the initiate VI. This clearing process keeps you from accidentally fetching a result from a previous initiate. Notice that only the results with the same name in the default or named signal configuration passed to Initiate are cleared. For example, if you call initiate with **selectorString** set to “signal:MySignal” and **resultName** set to “MyResult”, Initiate clears all results named “MyResult” that were previously initiated with the signal “MySignal”.

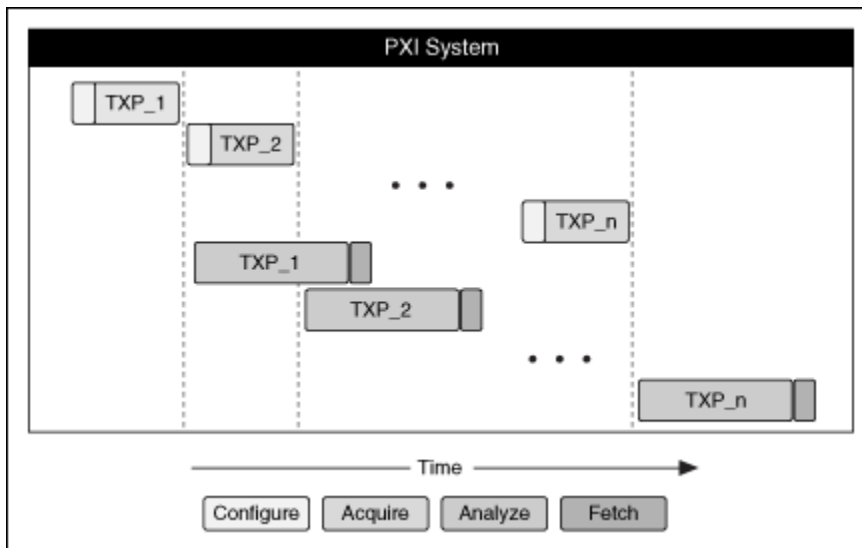
As mentioned, RFmx provides a set of Build String Utility functions to create named result strings. The following code snippet shows RFmxSpecAn_BuildOffsetString2 function to create a result string and fetch result for offset 1.

```
RFmxSpecAn_BuildSignalString("", "ACP_Result", SELECTOR_STRING_LEN,
selectorString);
// selectorString = "result::ACP_Result" RFmxSpecAn_Initiate(instrumentHandle,
selectorString, ""); RFmxSpecAn_BuildOffsetString2("", selectorString, 1,
SELECTOR_STRING_LEN, offsetResultString); //
offsetResultString = "result::ACP_Result/offset1"
RFmxSpecAn_ACPFetchOffsetMeasurement(instrumentHandle, offsetResultString, timeout,
&offCh1LowerRelativePower,
&offCh1UpperRelativePower, &offCh1LowerAbsolutePower, &offCh1UpperAbsolutePower);
```

The benefit of the overlapped measurements technique improves with more measurements in your sequence. In fact, there is no limit on the number of measurements you can overlap. However, as the number of named results increases, consider deleting the measurement results that have been fetched to conserve process memory space. The named results can be deleted using the `RFmxSpecAn_ClearNamedResult` function. After a named result is cleared all the measurement results for that named result are erased from the process memory of your application. If all named results need to be cleared from the memory `RFmxSpecAn_ClearAllNamedResults` function can be called.

Refer to the `RFmxSpecAnTxpMultipleResultNames` example application installed by RFmx SpecAn, which conducts multiple TxP measurements at different frequencies and clears the named result for TxP measurement once it has been fetched. The example can be found in the following location: `Users\Public\Documents\National Instruments\RFmx\SpecAn\Examples\C\Advanced\RFmxSpecAnTxpMultipleResultNames`

The following figure demonstrates the sequence of events for the mentioned TxP measurement example.



The actual time it takes to acquire the samples and analyze the TxP measurements may vary based on your system performance and measurement settings; however, the sequence of events such as configure, acquire, and so on, will remain the same. RFmx automatically starts the analysis or processing of the measurement immediately after the samples for the measurement are acquired. After the processing of the

measurement is complete, you can fetch the measurement results.

Depending on the computational complexity of the measurement, overlapped measurements technique can have a huge impact on overall test time. Measurement parallelism is a known technique to reduce test time. RFmx makes reducing test time easier by managing threads and results so that you can develop and realize the benefits of PXI faster.

RFmx Analysis-Only Mode

The RFmx Analysis-Only Mode further extends the functionality and flexibility of the RFmx API to handle the following use cases:

- Perform RF measurements with hardware that RFmx does not natively support
- Perform RF measurements on simulated waveforms
- Perform RF measurements on waveform data that was acquired previously and saved to disk

In the RFmx Analysis-Only Mode, time domain I/Q data or spectrum data is passed to the `RFmxSpecAn_AnalyzeIQ` function that performs the RF measurements that you specify on the raw waveform data.

Time domain measurements such as ModAcc/EVM or CCDF only accept complex I/Q data. Spectral measurements such as ACP, SEM, or CHP accept either complex I/Q data or spectrum data which must be in units of dBm. The `f0` in the spectrum data passed to the `RFmxSpecAn_AnalyzeIQ` function is calculated using the following formula:

$$f0_rfmx = f0_driver - fc$$

where,

`f0_rfmx` is start frequency of the spectrum passed to the `RFmxSpecAn_AnalyzeSpectrum` function

`fc` is the center frequency of spectral acquisition

`f0_driver` is the start frequency of the spectrum

In addition, if hardware is used to acquire the data, the input data provided to the `RFmxSpecAn_AnalyzeIQ` function must meet or exceed the minimum requirements of the RFmx recommended acquisition settings. For example, the sample I/Q rate can be higher than recommended sample rate, acquisition length can be longer than the recommended acquisition length, spectrum span can be wider than the recommended acquisition span, and so on.

The recommended acquisition settings are calculated based on the user-defined measurement configuration, and can be accessed through the `RFmxInstr` Get/Set methods in C/.NET, or by getting the values of the equivalent attributes in LabVIEW.

When using the RFmx Analysis-Only Mode, the RFmx driver ignores RFmx hardware settings such as reference level and attenuation. The only RF hardware setting that is not ignored is the center frequency, since it is needed for spectral measurement traces as well as some of the wireless standard-specific measurements, such as ACP and SEM, which define different configurations or limits based on the center frequency and the value which you set for the `BandClass` attribute.

In the following example, time-domain transmit power (TxP) measurement is performed on a simulated QAM waveform. This example completes the following operations in the Analysis-Only Mode:

- **OPEN**—Opens the RFmx session with an empty hardware resource name and with “`AnalysisOnly=1`” string wired to option string input of the `RFmxInstr` constructor.
- **CONFIGURE**—Configures the TxP measurement as if RFmx was used with hardware.
- **ANALYZE**—Passes a simulated waveform to the `RFmxSpecAn_AnalyzeIQ` function which is called instead of the `RFmxInstr_Initialize` function that is used when using RFmx with the built-in hardware support.
- **FETCH**—Fetches the measurement results using the same RFmx methods used to fetch results while using RFmx with the built-in hardware support.
- **CLOSE**—Closes the RFmx session.

```
// OPEN Session without resource name but with option string set to
"AnalysisOnly=1"
RFmxSpecAn_Initialize("", "AnalysisOnly=1", &instrumentHandle, NULL);
// CONFIGURE RFmx measurement using regular RFmx functions
```

```

RFmxSpecAn_TXPCfgMeasurementInterval(instrumentHandle, "", measInterval);
RFmxSpecAn_TXPCfgRBWFilter(instrumentHandle, "", RBW, RBWFilterType, RRCAlpha);
RFmxSpecAn_SelectMeasurements(instrumentHandle, "", RFMXSPECAN_VAL_TXP,
RFMXSPECAN_VAL_TRUE);
// Pass Waveform to AnalyzeIQ function RFmxSpecAn_AnalyzeIQ
(instrumentHandle, "", "", x0, dx, iq, arraySize, reset, out averagingDone);
// FETCH Results using regular RFmx functions
RFmxSpecAn_TXPFetchMeasurement(instrumentHandle, "", timeout, &averageMeanPower,
&peakToAveragePower, &maximumPower, &minimumPower);
// CLOSE Session
RFmxSpecAn_Close(instrumentHandle, RFMXSPECAN_VAL_FALSE);

```

Another way to use the RFmx Analysis-Only Mode is in combination with another driver, such as NI-RFSA, that acquires the waveform which is then passed to RFmx for analysis. In this case, the waveform acquisition, performed by the driver, is configured based on the recommended acquisition settings calculated by RFmx. The following example demonstrates this approach.

```

//Open
//Configure RFmx ACP measurement
//RFmx COMMIT RFmxSpecAn_Commit(instrumentHandle, "");
// Configure Acquisition
niRFSA_GetAttributeViReal64(rfsaSession, "",
NIRFSA_ATTR_MAX_DEVICE_INSTANTANEOUS_BANDWIDTH, &maxDeviceInstantaneousBandwidth)
RFmxInstr_GetRecommendedNumberOfRecords(instrumentHandle, "", &numberOfRecords);
RFmxInstr_GetRecommendedSpectralAcquisitionSpan(instrumentHandle, "",
&spectralAcquisitionSpan); if (maxDeviceInstantaneousBandwidth ≥
spectralAcquisitionSpan) {
// IQ Acquisition
} Else {
// Spectral Acquisition
RFmxInstr_GetRecommendedSpectralFFTWindow(instrumentHandle, "", &
recommendedFFTWindow);
RFmxInstr_GetRecommendedSpectralResolutionBandwidth(instrumentHandle, "",
&spectralRBW);
niRFSA_ConfigureAcquisitionType(rfsaSession, NIRFSA_VAL_SPECTRUM); switch
(recommendedFFTWindow) {
case
RFMXSPECAN_VAL_SPECTRUM_FFT_WINDOW_FLAT_TOP : FFTWindowType = NIRFSA_VAL_FLAT_TOP;
break; //other cases
case RFMXSPECAN_VAL_SPECTRUM_FFT_WINDOW_NONE : FFTWindowType = NIRFSA_VAL_UNIFORM;
break; }
niRFSA_GetAttributeViReal64(rfsaSession, "", NIRFSA_ATTR_CENTER_FREQUENCY,

```

```

&updatedCenterFrequency);    niRFSA_SetAttributeViReal64(rfsaSession, "",
NIRFSA_ATTR_RESOLUTION_BANDWIDTH, spectralRBW);
niRFSA_SetAttributeViInt32(rfsaSession, "", NIRFSA_ATTR_FFT_WINDOW_TYPE,
FFTWindowType);
niRFSA_SetAttributeViInt32(rfsaSession, "", NIRFSA_ATTR_RESOLUTION_BANDWIDTH_TYPE,
NIRFSA_VAL_RBW_BIN_WIDTH);
niRFSA_SetAttributeViInt32(rfsaSession, "", NIRFSA_ATTR_POWER_SPECTRUM_UNITS,
NIRFSA_VAL_DBM);
niRFSA_SetAttributeViReal64(rfsaSession, "", NIRFSA_ATTR_SPECTRUM_SPAN,
spectralAcquisitionSpan);
for(i= 0; i < numberOfRecords; i++)    {            if(i == 0)            reset = 1;
else reset = 0;
//READ the Power Spectrum from RFSA
niRFSA_ReadPowerSpectrumF64(rfsaSession, "", timeout, powerSpectrumData64,
numberOfSpectralLines, &spectrumInfo);
spectrumInfo.initialFrequency = spectrumInfo.initialFrequency -
updatedCenterFrequency;
for(j=0;j<numberOfSpectralLines;j++)    {            powerSpectrumData32[j] =
(ViReal32)powerSpectrumData64[j];            }
// Pass Waveform to AnalyzeIQ function
RFmxSpecAn_AnalyzeSpectrum(instrumentHandle, "", "", spectrumInfo.initialFrequency,
spectrumInfo.frequencyIncrement, powerSpectrumData32,
spectrumInfo.numberOfSpectralLines, reset, NULL)    } }
//Fetch ACP Measurement Results
// Close

```

The programming flow for using the RFmx Analysis-Only Mode in combination with NI-RFSA closely resembles the flow used when analyzing a simulated waveform using RFmx. However, in addition to opening and configuring RFmx session, you can also open and configure an NI-RFSA session. NI-RFSA performs the actual acquisition of the waveform based on the recommended acquisition settings that are passed on from RFmx session to RFSA session using get/sets in C/ .NET or setting the property nodes (LabVIEW). You must call the RFmx AnalyzeIQ function to calculate the recommended acquisition settings based on the user-defined RFmx measurement configuration before recommended acquisition settings are passed on to the RFSA. After NI-RFSA acquires the waveform, RFmxSpecAn_AnalyzeIQ function is called to perform user-defined measurements on the raw waveform data. Once the analysis of the waveform is complete, you can fetch measurement results and close both RFmx and RFSA sessions. Refer to the following examples for more information about using the RFmx Analysis-Only Mode to perform RF measurements:

- <LabVIEW directory>\examples\RFmx\SpecAn\RFmxSpecAn ACP

(Analysis Only).vi

- <Users>\Public\Documents\National Instruments\RFmx\SpecAn\Examples\C\RFmxSpecAnAcpAnalysisOnly
- <Users>\Public\Documents\National Instruments\RFmx\SpecAn\Examples\DotNET\VS2010\RFmxSpecAnAcpAnalysisOnly



Note

- Measurements such as harmonics and spurious emissions that require multiple acquisitions with different configurations are not supported in the Analysis-Only Mode. The Dynamic Range ACP measurement function is also not currently supported.
- For the Analysis-Only mode, you need to adjust the t_0 value of the I/Q waveforms according to pre-trigger samples present in the waveform before passing the waveform to the RFmx Analyze (IQ) function.



Note Use the following recommended settings while analyzing the LTE waveforms in the Analysis-Only mode:

- If the waveform data has been acquired with local oscillator (LO) at the center of RF signal bandwidth, set the value of RFMXLTE_ATTR_ACQUISITION_BANDWIDTH_OPTIMIZATION_ENABLED function to RFMXLTE_VAL_ACQUISITION_BANDWIDTH_OPTIMIZATION_ENABLED_TRUE; otherwise, set the value of this function to RFMXLTE_VAL_ACQUISITION_BANDWIDTH_OPTIMIZATION_ENABLED_FALSE.
- If you set the RFMXLTE_ATTR_MODACC_COMMON_CLOCK_SOURCE_ENABLED function to RFMXLTE_VAL_MODACC_COMMON_CLOCK_SOURCE_ENABLED_TRUE, the value of the RFMXLTE_ATTR_CENTER_FREQUENCY function configured in RFmxLTE session should be same as the center frequency used to acquire the waveform from the hardware.

RFmx Overview

RFmx is a family of instrument drivers that share a single session handle across multiple drivers. You open one RFmx session per signal analyzer and use that single session handle to configure and perform measurements with the different drivers, such as SpecAn, Demod, LTE, and so on.

The RFmx driver has a shared set of APIs for opening and closing the RFmx session as well as configuring hardware settings, such as the Reference Clock configuration, that are shared across drivers. These shared functions are called the RFmxInstr driver. RFmxInstr does not contain any measurements, and you always use RFmxInstr along with one or more other RFmx drivers that expose the measurements you need to perform.

The following is an example of how you use the RFmx API drivers in .NET to capture a spectrum using the RFmx SpecAn driver.

C#

```
instrSession = new RFmxInstrMX(resourceName, ""); //Step 1
specAn = instrSession.GetSpecAnSignalConfiguration(); //Step 2
instrSession.ConfigureFrequencyReference("", frequencySource, frequency); //Step 3
specAn.ConfigureRF("", centerFrequency, referenceLevel, externalAttenuation);
//Step 4
specAn.Spectrum.Configuration.ConfigureSpan("", span); //Step 4
specAn.SelectMeasurements("", RFmxSpecAnMXMeasurementTypes.Spectrum, true); //Step
5
specAn.Initiate("", ""); //Step 6
specAn.Spectrum.Results.FetchSpectrum("", timeout, ref spectrum); //Step 7
instrSession.Close(); //Step 8
```

VB

```
instrSession = New RFmxInstrMX(resourceName, "") 'Step 1
specAn = instrSession.GetSpecAnSignalConfiguration() 'Step 2
instrSession.ConfigureFrequencyReference("", frequencySource, frequency) 'Step 3
specAn.ConfigureRF("", centerFrequency, referenceLevel, externalAttenuation) 'Step
4
specAn.Spectrum.Configuration.ConfigureSpan("", span) 'Step 4
specAn.SelectMeasurements("", RFmxSpecAnMXMeasurementTypes.Spectrum, True) 'Step 5
```

```
specAn.Initiate("", "") 'Step 6
specAn.Spectrum.Results.FetchSpectrum("", timeout, spectrum) 'Step 7
instrSession.Close() 'Step 8
```

1. Open an RFmx session using RFmxInstr constructor.
2. Get a SpecAn Signal using Signal Configuration methods from the RFmx session.
Note: In .NET library, the Signal configuration methods are defined as extension methods on RFmxInstr in the RFmxSpecAn library. You can call these methods using RFmxInstr Session.
3. Configure the Frequency Reference using Configure APIs or Set/Get APIs in RFmxInstr.
4. Configure the Reference Level, Frequency, and Span using RFmxSpecAn.
5. Select Spectrum measurement using RFmxSpecAn.
6. Initiate the measurement using RFmxSpecAn.
7. Fetch the Spectrum using RFmxSpecAn.
8. Close the RFmx session using RFmxInstr.

When you use RFmxInstr APIs to configure hardware settings in an RFmx session, those settings apply to all the RFmx drivers that are present in the same RFmx session. However, the following settings are RFmx driver-specific and are not shared across different RFmx drivers within the same RFmx session:

- Center Frequency
- Reference Level
- External Attenuation
- Trigger Settings

For example, if you are making both RFmx LTE and RFmx WCDMA measurements within one RFmx session, setting the frequency in RFmx LTE does not affect the frequency used in RFmx WCDMA.

Selector Strings

RFmx is a powerful API that supports basic and advanced features and provides multiple ways to configure measurements and fetch measurement results. The same configuration and fetch methods can be used in basic and advanced use cases such as named signal configurations without having to rewrite code. The following code shows

a basic ACP measurement that does not use named signal configurations.

C#

```
/* Create a new RFmx Session */
RFmxInstrMX instrSession = new RFmxInstrMX(resourceName, "");
/* Get SpecAn signal */
RFmxSpecAnMX specAn = instrSession.GetSpecAnSignalConfiguration();
/* Configure ACP measurement */
specAn.ConfigureRF("", centerFrequency, referenceLevel, externalAttenuation);
specAn.Acp.Configuration.ConfigureAveraging("", averagingEnabled, averagingCount,
averagingType);
specAn.Acp.Configuration.ConfigureCarrierAndOffsets("", integrationBandwidth,
numberOfOffsetChannels, channelSpacing);
/* Retrieve results */
specAn.Acp.Results.Read("", timeout, out carrierAbsolutePower, out
offCh0LowerRelativePower, out offCh0UpperRelativePower,
out offCh1LowerRelativePower, out offCh1UpperRelativePower);
instrSession.Close();
```

VB

```
' Create a new RFmx Session
Dim instrSession As New RFmxInstrMX(resourceName, "")
' Get SpecAn signal
Dim specAn As RFmxSpecAnMX = instrSession.GetSpecAnSignalConfiguration()
'Configure ACP measurement
specAn.ConfigureRF("", centerFrequency, referenceLevel, externalAttenuation)
specAn.Acp.Configuration.ConfigureAveraging("", averagingEnabled, averagingCount,
averagingType)
specAn.Acp.Configuration.ConfigureCarrierAndOffsets("",
integrationBandwidth, numberOfOffsetChannels, channelSpacing)
' Retrieve results
specAn.Acp.Results.Read("", timeout, out carrierAbsolutePower, out
offCh0LowerRelativePower,
out offCh0UpperRelativePower, out offCh1LowerRelativePower, out
offCh1UpperRelativePower);
instrSession.Close()
```

You can enhance the code as shown in the following code to use named signal configurations by passing a signalName to the GetSpecAnSignalConfiguration method. In .NET API, you provide signal name to Get Signal Configuration methods only.

C#

```

/* Create a new RFmx Session */
RFmxInstrMX instrSession = new RFmxInstrMX(resourceName, "");
/* Get SpecAn signal */
RFmxSpecAnMX specAn = instrSession.GetSpecAnSignalConfiguration("MySignalConfig");
/* Configure measurement */
specAn.ConfigureRF("", centerFrequency, referenceLevel, externalAttenuation);
specAn.Acp.Configuration.ConfigureAveraging("", averagingEnabled, averagingCount,
averagingType);
specAn.Acp.Configuration.ConfigureCarrierAndOffsets("", integrationBandwidth,
numberOfOffsetChannels, channelSpacing);
/* Retrieve results */
specAn.Acp.Results.Read("", timeout, out carrierAbsolutePower, out
offCh0LowerRelativePower,
out offCh0UpperRelativePower, out offCh1LowerRelativePower, out
offCh1UpperRelativePower);
instrSession.Close();

```

VB

```

' Create a new RFmx Session
Dim instrSession As New RFmxInstrMX(resourceName, "")
' Get SpecAn signal
Dim specAn As RFmxSpecAnMX =
instrSession.GetSpecAnSignalConfiguration(MySignalConfig)
' Configure measurement
specAn.ConfigureRF("", centerFrequency, referenceLevel, externalAttenuation)
specAn.Acp.Configuration.ConfigureAveraging("", averagingEnabled, averagingCount,
averagingType) specAn.Acp.Configuration.ConfigureCarrierAndOffsets("",
integrationBandwidth numberOfOffsetChannels, channelSpacing)
' Retrieve results
specAn.Acp.Results.Read("", timeout, carrierAbsolutePower,
offCh0LowerRelativePower, offCh0UpperRelativePower,
offCh1LowerRelativePower, offCh1UpperRelativePower)
instrSession.Close()

```

The **selectorString** parameter in RFmx methods allows you to specify the full context of what you are configuring or fetching. In .NET API, you will not need to provide signal name context in selectorString parameter of Configure and Fetch methods. The following code shows an example of the **selectorString** input on one of the RFmx fetch measurement methods.

```
public int FetchCarrierMeasurement(string selectorString, double timeout, out
double
absolutePower, out double totalRelativePower, out double carrierFrequency, out
double
integrationBandwidth);
```

For example, an ACP measurement can have multiple offset channels and you can configure the frequency and measurement bandwidth properties for each offset channel. When you configure the measurement bandwidth for a specific offset channel, you must specify the offset channel for which you are configuring the Measurement Bandwidth property. When you configure the first offset channel you must specify the context in the **selectorString** input as “offset0” as shown in the following code.

C#

```
/* Fetch ACP offset measurements */
specAn.Acp.Configuration.ConfigureOffsetRrcFilter("offset0", rrcFilterEnabled,
rrcFilterAlpha);
```

VB

```
' Fetch ACP offset measurements
specAn.Acp.Configuration.ConfigureOffsetRrcFilter("offset0", rrcFilterEnabled,
rrcFilterAlpha)
```

The contexts you can specify depend on what you are configuring or fetching. For example, if you are configuring a carrier channel you specify which carrier channel, such as “carrier3”, that you are configuring. If you are configuring an offset channel, you specify which offset channel, such as “offset2”, that you are configuring. Named Results can be specified in only the Initiate and fetch methods.

Multiple Contexts

When using multiple features in the API, you can specify multiple contexts in the same **selectorString** input separated with a ‘/’. For example, to fetch a result for carrier 0 in the named result context called “MyResult”, you must specify “result::MyResult/carrier0” to the selectorString parameter in the Fetch method. Another example of

multiple contexts is when performing LTE non-contiguous carrier aggregation measurements. In this case the signal is composed of multiple subblocks and each subblock can contain multiple carriers. When you configure a property of a carrier in LTE non-contiguous carrier aggregation, you must specify the index of the carrier within the subblock and the index of the subblock. For example, you would specify that you are configuring the first carrier in the second subblock by using “subblock1/carrier0” as the selector string.

String Prefixes in Selector Strings

RFmx requires a prefix, such as “carrier” or “result::”, before the number or name in the selector string so that it is clear what is being configured. The “::” is required for “signal::” and “result::” because those items use arbitrary user defined names and the API needs to be able to know where the user specified name starts in the selector string. You can specify the selector strings as constants, such as “carrier0” and “result::myResult” or use the build string utility methods provided by RFmx to create correctly formed strings.

Build String Utility Methods

RFmx provides a set of utility methods as shown below to build context string for various measurement configurations.

- BuildResultString
- BuildCarrierString2
- BuildHarmonicString2
- BuildMarkerString2
- BuildOffsetString2
- BuildSpurString2
- BuildRangeString2

For example, the BuildResultString method takes in a result name and outputs a string containing the contexts you specified. If you call the BuildResultString method with “MyResult” passed to the **resultName** input, the output string will be “result::MyResult”. Each RFmx driver provides build string utility methods for specific cases such as BuildSpurString2 which takes in a spur number, in addition to optionally taking in a previously constructed selector string.

C#

```
static public string BuildSpurString2(string
selectorString, int spurNumber);
```

VB

```
Public Shared Function BuildSpurString2(selectorString As
String, spurNumber As Integer) As String
```

Selector String in Set or Get Methods

When setting properties from an RFmx set or get method in .NET, you specify the context using the **selectorString** parameter.

Selector Strings when using Array Methods

Note that RFmx provides array versions of configure and fetch methods for many cases that allow you to configure multiple instances of a property or fetch multiple instances of a result. For example, to configure the frequencies of all the ACP measurement offset channels, you can use `Acp.Configuration.ConfigureOffsetArray` passing in an array that has the frequency for each of the offset channels. When using these array methods, you do not need to specify, in the **selectorString** input, the context that the elements in the array correspond to. For example, you pass "" as the selector string in `Acp.Configuration.ConfigureOffsetArray`. However, in case of Fetch array methods, if you are using named results, you pass the result context in the selector string.

Selector String Syntax

The following table lists the various active channel string formats that you can use. For integral channel specifiers, valid values for *i* are 0 to (number of channels - 1). For example, if you set the value of `NumberOfOffsets` to 5, you can configure ACP offset properties for `offset0`, `offset1`, `offset2`, `offset3`, and `offset4`. In the following table, the term channel or *ch* is used to refer to numeric contexts such as carrier, offset, or range.

Selector	String Method	Example

String		
ch<i>	Configures one channel	carrier0
ch<i>-<j>	Configures a range of channels between i and j	carrier0-5
ch<i>:<j>	Configures a range of channels between i and j	carrier0:5
ch<i>, ch<j>, ch<k>	Configures channels i, j, and k	carrier0, carrier2, carrier4
ch<i>, ch<j>-<k>	Configures channel i and a range of channels between j and k	carrier0, carrier1-5
ch<i>, ch<j>:<k>	Configures channel i and a range of channels between j and k	carrier0, carrier1:5
ch::all	Configures all channels from 0 to (Number of X - 1). Number of X is the current value of the Number of .. property that corresponds to the context being used. For example, in an ACP measurement, you set the Number of Offsets property before setting any offset properties and when you use “all”, RFmx uses the current value of Number of Offsets to determine which offset channels to configure.	offset::all

The following table lists the various selector string formats that you can use to specify the named results (using the “result::” prefix). Note that the “result::” prefix is valid only when fetching results and currently cannot be used to configure measurements.

Syntax	String Function	Example
result:: <resultname>/ch<i><i></i></resultname>	Specifies channel <i>i</i> for result <i><resultname></i> on signal corresponding to the signal configuration object being used	result::r1/ carrier0
result:: <resultname></resultname>	Specifies named result <i><resultname></i> on signal corresponding to the signal configuration object being used	result::r1

The following characters are not allowed in signal names and result names: ()*+,-./{} !"#\$%&'<=>?@[]\^`|~ and any non-printable character.

Array Configuration and Fetch Functions

Array Configuration Methods

RFmx provides array-based configuration methods to enable you to set different values to multiple instances of an attribute in a single call to the method. The first N instances of each attribute are configured to the values specified in the corresponding elements of the array. If you pass an empty array to the input of a method, none of the instances of the attribute corresponding to that input are modified. You can pass up to M values in the array, where M is the current value of the <Number of X> attribute (for example, Number of Carriers, Number of Harmonics, Number of Offsets, and so on). If the array size, N, is smaller than M, only the first N instances of the attribute are modified.

For example, if you set SetNumberOfOffsets to 3, and if the **offsetSideband** parameter of the ConfigureOffsetArray method has two elements in the array, the ACP Offset Sideband attribute for the first two offset channels are set to the values in the array, and the ACP Offset Sideband attribute for the third offset channel is not modified.

In the following code, offsetPowerReferenceCarrier and offsetPowerReferenceSpecific should be of same size. In this case, the array size is equal to the value of the

numberOfOffsets parameter.

C#

```
string selectorString = string.Empty; int numberOfOffsets = 3;
RFmxInstrMX instrSession = new RFmxInstrMX("Rfsa", "");
RFmxSpecAnMX specAn = instrSession.GetSpecAnSignalConfiguration();
specAn.SelectMeasurements(selectorString, RFmxSpecAnMXMeasurementTypes.Acp,
true);
RFmxSpecAnMXAcpOffsetPowerReferenceCarrier[] offsetPowerReferenceCarrier = {
RFmxSpecAnMXAcpOffsetPowerReferenceCarrier.Closest,
RFmxSpecAnMXAcpOffsetPowerReferenceCarrier.Composite,
RFmxSpecAnMXAcpOffsetPowerReferenceCarrier.Closest };
int[] offsetPowerReferenceSpecific = {0, 1, 2 };
specAn.Acp.Configuration.ConfigureNumberOfOffsets(selectorString, numberOfOffsets);
specAn.Acp.Configuration.ConfigureOffsetPowerReferenceArray(selectorString,
offsetPowerReferenceCarrier, offsetPowerReferenceSpecific);
```

VB

```
Dim selectorString As String = String.Empty Dim numberOfOffsets As Integer = 3
Dim instrSession As RFmxInstrMX = New RFmxInstrMX("Rfsa", "")
Dim specAn As RFmxSpecAnMX = instrSession.GetSpecAnSignalConfiguration()
specAn.SelectMeasurements (selectorString, RFmxSpecAnMXMeasurementTypes.Acp, True)
Dim offsetPowerReferenceCarrier As RFmxSpecAnMXAcpOffsetPowerReferenceCarrier() =
{RFmxSpecAnMXAcpOffsetPowerReferenceCarrier.Closest,
RFmxSpecAnMXAcpOffsetPowerReferenceCarrier.Composite,
RFmxSpecAnMXAcpOffsetPowerReferenceCarrier.Closest}
Dim offsetPowerReferenceSpecific As Integer() = {0, 1, 2}
specAn.Acp.Configuration.ConfigureNumberOfOffsets (selectorString, numberOfOffsets)
specAn.Acp.Configuration.ConfigureOffsetPowerReferenceArray(selectorString,
offsetPowerReferenceCarrier, offsetPowerReferenceSpecific)
```



Note For array configuration methods that have multiple array inputs, you must pass identically sized arrays to each input.

To set the values to a subset of attribute instances, you can use the non-array based methods. In the non-array based methods, you can configure one or more instances of an attribute to the same value in each call to the method.

Array Fetch Methods

RFmx provides array-based fetch methods to enable you to fetch multiple instances of a result in a single call to the method. Array fetch methods return arrays containing the values for all instances of the results during a specific call to a measurement.

For example, if you have configured an ACP measurement to measure three offset channels, the array contains the results for these three offset channels.

For array-based Fetch methods, the ref arguments can be null or initialized to known sample count before fetching. API will ensure ref arguments are re-initialized to expected array size if it is null or size doesn't match.

C#

```
string selectorString = string.Empty; double timeout = 10F; int numberOfOffsets =
6;
double[] lowerRelativePower = new double[numberOfOffsets];
double[] upperRelativePower = new double[numberOfOffsets];
double[] upperAbsolutePower = new double[numberOfOffsets];
double[] lowerAbsolutePower = new double[numberOfOffsets];
RFmxInstrMX instrSession = new RFmxInstrMX("Rfsa", "");
RFmxSpecAnMX specAn = instrSession.GetSpecAnSignalConfiguration();
specAn.SelectMeasurements("", RFmxSpecAnMXMeasurementTypes.Acp, true);
specAn.Acp.Configuration.ConfigureNumberOfOffsets(selectorString, numberOfOffsets);
specAn.Initiate(selectorString, "");
specAn.Acp.Results.FetchOffsetMeasurementArray(selectorString, timeout,
lowerRelativePower,                                ref
upperRelativePower,
ref lowerAbsolutePower, ref upperAbsolutePower);
```

VB

```
Dim selectorString As String = String.Empty Dim timeout As Double = 10.0
Dim numberOfOffsets As Integer = 6
Dim lowerRelativePower As Double() = New Double(numberOfOffsets) {}
Dim upperRelativePower As Double() = New Double(numberOfOffsets) {}
Dim upperAbsolutePower As Double() = New Double(numberOfOffsets) {}
Dim lowerAbsolutePower As Double() = New Double(numberOfOffsets) {}
Dim instrSession As RFmxInstrMX = New RFmxInstrMX("Rfsa", "")
Dim specAn As RFmxSpecAnMX = instrSession.GetSpecAnSignalConfiguration()
```

```
specAn.SelectMeasurements(selectorString, RFmxSpecAnMXMeasurementTypes.Acp, True)
specAn.Acp.Configuration.ConfigureNumberOfOffsets(selectorString, numberOfOffsets)
specAn.Initiate(selectorString, "")
specAn.Acp.Results.FetchOffsetMeasurementArray(selectorString, timeout,
lowerRelativePower
upperRelativePower, lowerAbsolutePower, upperAbsolutePower)
```

Signal Configurations

RFmx allows you to create measurement configurations called signal configurations, within the same personality. Each signal configuration instance holds a unique instance of signal and measurement property values. By using multiple signal configurations, you can switch between different measurement configurations without the overhead of configuring the measurements each time. Signal configurations also allow you to execute multiple configurations of the same measurement at the same time.

RFmx creates one default signal configuration for each personality. This default configuration is unnamed. You can create additional named signal configurations. These signal configurations can be created using extension methods on RFmxInstr. Each specific driver defines its own extension methods on RFmxInstr. For instance, GetSpecAnSignalConfiguration() in RFmx SpecAn and GetDemodSignalConfiguration() in RFmx Demod and so on.

Default Signal Configuration

You can create the default signal configuration using the following method:

```
public static RFmxSpecAnMX
GetSpecAnSignalConfiguration(this RFmxInstrMX instrSession)
```

C#

```
RFmxInstrMX instrSession = new RFmxInstrMX("Rfsa", "");
RFmxSpecAnMX specAn =
instrSession.GetSpecAnSignalConfiguration();
```

VB

```
Dim instrSession As RFmxInstrMX = New RFmxInstrMX("Rfsa",
"") Dim specAn As RFmxSpecAnMX =
instrSession.GetSpecAnSignalConfiguration()
```

Named Signal Configuration

You can create named signal configurations using the following method:

```
public static RFmxSpecAnMX
GetSpecAnSignalConfiguration(this RFmxInstrMX instrSession,
string signalName)
```

C#

```
RFmxInstrMX instrSession = new RFmxInstrMX("Rfsa", "");
RFmxSpecAnMX specAn =
instrSession.GetSpecAnSignalConfiguration("s1");
```

You can also create named signal using "signal::" prefix in the signal name

```
RFmxInstrMX instrSession = new RFmxInstrMX("Rfsa", "");
RFmxSpecAnMX specAn =
instrSession.GetSpecAnSignalConfiguration("signal::s1");
```

VB

```
Dim instrSession As RFmxInstrMX = New RFmxInstrMX("Rfsa",
"") Dim specAn As RFmxSpecAnMX =
instrSession.GetSpecAnSignalConfiguration("s1")
```

You can also create named signal using "signal::" prefix in the signal name

```
Dim instrSession As RFmxInstrMX = New RFmxInstrMX("Rfsa",
"") Dim specAn As RFmxSpecAnMX =
instrSession.GetSpecAnSignalConfiguration("signal::s1")
```

When you invoke Initiate method on a signal configuration object, RFmx performs the measurements that are enabled in that signal configuration.

The following code shows how to create two signals, named LTE-5MHZ and LTE_20MHz in RFmx LTE. The two signal configurations use different center frequencies and bandwidths. The LTE_5MHz signal configuration is initiated before the LTE_20MHz signal configuration.

C#

```
// Create a new RFmx Session instrSession = new RFmxInstrMX(resourceName, "");
//Configure reference frequency and source
instrSession.ConfigureFrequencyReference("", frequencyReferenceSource,
frequencyReferenceFrequency);
//Create Signal Configurations
lte5MHzSignal = instrSession.GetLteSignalConfiguration("LTE_5MHz");
lte20MHzSignal = instrSession.GetLteSignalConfiguration("LTE_20MHz");
//Configure Settings (Frequency, Bandwidth) for each Signal Configurations and
Enable CHP measurement
lte5MHzSignal.ConfigureFrequency("", centerFrequency_LTE_5MHz);
lte20MHzSignal.ConfigureFrequency("", centerFrequency_LTE_20MHz);
lte5MHzSignal.ComponentCarrier.Configure("", carrierBandwidth_LTE_5MHz,
componentCarrierFrequency, cellID);
lte20MHzSignal.ComponentCarrier.Configure("", carrierBandwidth_LTE_20MHz,
componentCarrierFrequency, cellID);
lte5MHzSignal.SelectMeasurements("", RFmxLteMXMeasurementTypes.Chp, true);
lte20MHzSignal.SelectMeasurements("", RFmxLteMXMeasurementTypes.Chp, true);
//Initiate each Signal Configuration
lte5MHzSignal.Initiate("", "");
instrSession.WaitForAcquisitionComplete(timeout);
lte20MHzSignal.Initiate("", "");
//Fetch Results
lte5MHzSignal.Chp.Results.ComponentCarrier.FetchMeasurement("", timeout, out
absolutePower_LTE_5MHz, out relativePower_LTE_5MHz);
lte20MHzSignal.Chp.Results.ComponentCarrier.FetchMeasurement("", timeout, out
absolutePower_LTE_20MHz, relativePower_LTE_20MHz);
```

VB

```
' Create a new RFmx Session instrSession = New RFmxInstrMX(resourceName, "")
'Configure reference frequency and source
instrSession.ConfigureFrequencyReference("", frequencyReferenceSource,
frequencyReferenceFrequency)
'Create Signal Configurations
lte5MHzSignal = instrSession.GetLteSignalConfiguration("LTE_5MHz")
lte20MHzSignal = instrSession.GetLteSignalConfiguration("LTE_20MHz")
```

```

'Configure Settings (Frequency, Bandwidth) for each Signal Configurations and
Enable CHP measurement
lte5MHzSignal.ConfigureFrequency("", centerFrequency_LTE_5MHz)
lte20MHzSignal.ConfigureFrequency("", centerFrequency_LTE_20MHz)
lte5MHzSignal.ComponentCarrier.Configure("", carrierBandwidth_LTE_5MHz,
componentCarrierFrequency, cellID)
lte20MHzSignal.ComponentCarrier.Configure("", carrierBandwidth_LTE_20MHz,
componentCarrierFrequency, cellID)
lte5MHzSignal.SelectMeasurements("", RFmxLteMXMeasurementTypes.Chp, True)
lte20MHzSignal.SelectMeasurements("", RFmxLteMXMeasurementTypes.Chp, True)
'Initiate each Signal Configuration
lte5MHzSignal.Initiate("", "")
instrSession.WaitForAcquisitionComplete(timeout)
lte20MHzSignal.Initiate("", "")
'Fetch Results
lte5MHzSignal.Chp.Results.ComponentCarrier.FetchMeasurement("", timeout,
absolutePower_LTE_5MHz, relativePower_LTE_5MHz)
lte20MHzSignal.Chp.Results.ComponentCarrier.FetchMeasurement("", timeout,
absolutePower_LTE_20MHz, relativePower_LTE_20MHz)

```



Note NI does not recommend using named signal configurations when only a few attributes are different between the configurations. For example, when performing the same measurement across different frequencies, NI recommends you use a single signal configuration and change the center frequency before each measurement.

The following characters are not allowed in signal or result names: `()*+,-./{}!\"#$%&'&:;<=>?@[\\]^_`|~` and any non-printable character.

The following code shows an example of creating and deleting a signal configuration named “sig1”.

C#

```

RFmxSpecAnMX specAn =
instrSession.GetSpecAnSignalConfiguration("sig1");
specAn.DeleteSignalConfiguration();

```

VB

```
Dim specAn As RFmxSpecAnMX =
instrSession.GetSpecAnSignalConfiguration("sig1")
specAn.DeleteSignalConfiguration()
```

The following code shows an example of creating a signal named WCDMA_Band14 and calling a configure VI with that signal name.

C#

```
RFmxSpecAnMX specAn =
instrSession.GetSpecAnSignalConfiguration("WCDMA_Band14");
specAn.ConfigureFrequency("", centerFrequency);
```

VB

```
Dim specAn As RFmxSpecAnMX =
instrSession.GetSpecAnSignalConfiguration("WCDMA_Band14")
specAn.ConfigureFrequency("", centerFrequency)
```

RFmx signal configurations can be cloned using driver-specific RFmx Clone Signal Configuration method. For instance,

RFmxSpecAnMX.CloneSignalConfiguration function in RFmx SpecAn.

```
public int CloneSignalConfiguration(string newSignalName,
out RFmxSpecAnMX signalConfiguration)
```

When you clone a configuration, a new signal configuration is created that inherits all the configuration settings of the signal configuration being cloned. In the new signal configuration, the original configuration settings can be overwritten by calling the appropriate signal configurations VIs with the new signal configuration name.

Composite Measurements

Composite measurement is a technique which allows multiple measurements to be performed in parallel on the same acquired data. This technique leads to faster measurement times since all the measurements in a composite set are based on the same acquired data, and are processed in parallel threads on the CPU. Although the same data set is used for each measurement in a composite set, RFmx allows different

configuration settings for each measurement, enabling multifaceted signal analysis. The following code snippets show the difference between serial measurements and composite measurements:

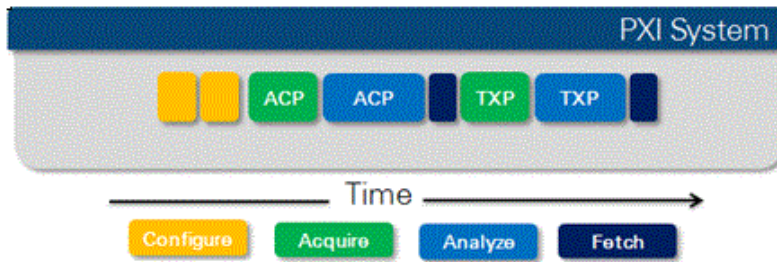
Serial measurement

C#

```
instrSession = new RFmxInstrMX(resourceName, ""); specAn =
    instrSession.GetSpecAnSignalConfiguration() specAn.ConfigureRF("",
centerFrequency,
    referenceLevel, externalAttenuation);
specAn.Txp.Configuration.ConfigureRbwFilter("", rbw,
    rbwFilterType, rrcAlpha);
specAn.Acp.Configuration.ConfigureCarrierAndOffsets("",
    integrationBandwidth, NumberOfOffsetChannels, channelSpacing);
specAn.Acp.Results.Read("",
    timeout, out carrierAbsolutePower, out offCh0LowerRelativePower, out
offCh0UpperRelativePower,
                                out offCh1LowerRelativePower, out
offCh1UpperRelativePower);
    specAn.Txp.Results.Read("", timeout, out averageMeanPower, out
peakToAverageRatio, out
    maximumPower, out minimumPower); instrSession.Close();
```

VB

```
instrSession = New RFmxInstrMX(resourceName, "") specAn =
    instrSession.GetSpecAnSignalConfiguration() specAn.ConfigureRF("",
centerFrequency,
    referenceLevel, externalAttenuation)
specAn.Txp.Configuration.ConfigureRbwFilter("", rbw,
    rbwFilterType, rrcAlpha)
specAn.Acp.Configuration.ConfigureCarrierAndOffsets("",
    integrationBandwidth, NumberOfOffsetChannels, channelSpacing)
specAn.Acp.Results.Read("",
    timeout, carrierAbsolutePower, offCh0LowerRelativePower,
offCh0UpperRelativePower,
                                offCh1LowerRelativePower, offCh1UpperRelativePower)
    specAn.Txp.Results.Read("", timeout, averageMeanPower, peakToAverageRatio,
maximumPower,
    minimumPower) instrSession.Close()
```



Composite measurement

C#

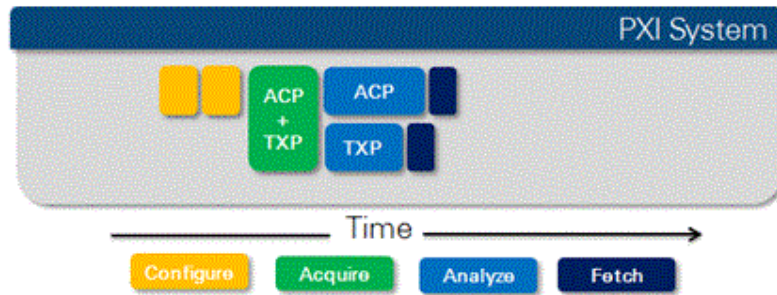
```
instrSession = new RFmxInstrMX(resourceName, "");
specAn = instrSession.GetSpecAnSignalConfiguration()
specAn.ConfigureRF("", centerFrequency, referenceLevel, externalAttenuation);
specAn.Txp.Configuration.ConfigureRbwFilter("", rbw, rbwFilterType, rrcAlpha);
specAn.Acp.Configuration.ConfigureCarrierAndOffsets("",
integrationBandwidth, NumberOfOffsetChannels, channelSpacing);
specAn.SelectMeasurements("", RFmxSpecAnMXMeasurementTypes.Acp |
RFmxSpecAnMXMeasurementTypes.Txp, enableAllTraces);
specAn.Initiate("", ""); specAn.Acp.Results.FetchOffsetMeasurementArray("",
timeout, ref
lowerRelativePowerArr, ref upperRelativePowerArr, ref lowerAbsolutePowerArr, ref
upperAbsolutePowerArr);
specAn.Txp.Results.FetchMeasurement("", timeout, out averageMeanPower, out
peakToAverageRatio, out maximumPower, out minimumPower);
instrSession.Close();
```

VB:

```
instrSession = New RFmxInstrMX(resourceName, "")
specAn = instrSession.GetSpecAnSignalConfiguration()
specAn.ConfigureRF("", centerFrequency, referenceLevel, externalAttenuation)
specAn.Txp.Configuration.ConfigureRbwFilter("", rbw, rbwFilterType, rrcAlpha)
specAn.Acp.Configuration.ConfigureCarrierAndOffsets("", integrationBandwidth,
NumberOfOffsetChannels, channelSpacing)
specAn.SelectMeasurements("", RFmxSpecAnMXMeasurementTypes.Acp Or
RFmxSpecAnMXMeasurementTypes.Txp, enableAllTraces)
specAn.Initiate("", "") specAn.Acp.Results.FetchOffsetMeasurementArray("", timeout,
lowerRelativePowerArr, upperRelativePowerArr,
lowerAbsolutePowerArr, upperAbsolutePowerArr)
specAn.Txp.Results.FetchMeasurement("", timeout, averageMeanPower,
```

```
peakToAverageRatio,maximumPower, minimumPower)
instrSession.Close()
```

Multiple measurements on the same IQ data



When the composite measurements are performed, only a single acquisition is made and each measurement in the composite set is performed on the same acquired data. Internally, RFmx merges the acquisition settings from all the measurements in a composite set to provide the acquisition settings that satisfy the needs of all the measurements being performed.

However, not all measurements can be performed as composite measurements. For example, a harmonics measurement cannot be performed in composite with an ACP measurement, since a harmonics measurement requires multi-segment acquisitions for different harmonics. To perform multiple measurements as composite measurements, use the `RFmxSpecAnMX.SelectMeasurements` method, and pass the measurement types you want to perform together to the parameter measurement using bit-wise OR operator.

Besides faster test execution times, composite measurements also allow you to see different measurements on the same DUT signal enabling you to debug difficult issues. For example, you can see if a bad ACP result is correlated to a bad EVM result.

Overlapped Measurements and Named Results

Many automated test applications require multiple independent measurements on a single instrument. Such applications include amplifier frequency/power sweeps or multi-DUT wireless device test. Often engineers implement these applications by

serially sequencing measurements.

However, with RFmx, you can perform multiple measurements overlapped in time with a feature called overlapped measurements. This feature can be used when measurements in a sequence are independent of previous results. The measurements must be performed on separate acquisitions because they require different hardware configurations such as a different frequency or reference level.

Overlapped measurements maximize utilization of the RF hardware and enable parallel processing of the measurements, thus vastly reducing the overall test time for a sequence of multiple measurements. In the following conceptual example, two overlapped ACP measurements are performed at two different reference levels. This simple program completes the following operations:

- Configure the reference level for the first ACP measurement
- Initiate the first ACP acquisition.
- Wait for the ACP acquisition to be complete and the PXI RF module to be released. RFmx spawns a thread for ACP analysis in the background without holding up the next measurement.
- Initiate the second ACP acquisition (before the first ACP measurement analysis is complete).
- Wait for the second ACP acquisition to complete (before the first ACP measurement analysis is complete).
- Fetch both the measurement results when their respective analysis threads are complete.

C#

```
specAn.ConfigureReferenceLevel("", referenceLevel1); specAn.Initiate("",
"ACP_Results_1"); instrSession.WaitForAcquisitionComplete(timeout);
specAn.ConfigureReferenceLevel("", referenceLevel2); specAn.Initiate("",
"ACP_Results_2"); instrSession.WaitForAcquisitionComplete(timeout);
resultString1 = RFmxSpecAnMX.BuildResultString("ACP_Results_1"); //
resultString1 = "result::ACP_Results_1"
specAn.Acp.Results.FetchOffsetMeasurementArray(resultString1,
timeout, ref lowerRelativePowerArr1, ref upperRelativePowerArr1, ref
lowerAbsolutePowerArr1, ref upperAbsolutePowerArr1);
resultString2 = RFmxSpecAnMX.BuildResultString("ACP_Results_2"); //
resultString2 = "result::ACP_Results_2"
```

```
specAn.Acp.Results.FetchOffsetMeasurementArray(resultString1,
timeout, ref lowerRelativePowerArr1, ref upperRelativePowerArr1, ref
lowerAbsolutePowerArr1, refupperAbsolutePowerArr1);
```

VB

```
specAn.ConfigureReferenceLevel("", referenceLevel1) specAn.Initiate("",
"ACP_Results_1")
instrSession.WaitForAcquisitionComplete(timeout) specAn.ConfigureReferenceLevel("",
referenceLevel2)
specAn.Initiate("", "ACP_Results_2")
instrSession.WaitForAcquisitionComplete(timeout)
resultString1 = RFmxSpecAnMX.BuildResultString("ACP_Results_1")
' resultString1 = "result::ACP_Results_1"
specAn.Acp.Results.FetchOffsetMeasurementArray(resultString1,
timeout, lowerRelativePowerArr1, upperRelativePowerArr1, lowerAbsolutePowerArr1,
upperAbsolutePowerArr1)
resultString2 = RFmxSpecAnMX.BuildResultString("ACP_Results_2")
' resultString2 = "result::ACP_Results_2"
specAn.Acp.Results.FetchOffsetMeasurementArray(resultString2,
timeout, lowerRelativePowerArr2, upperRelativePowerArr2, lowerAbsolutePowerArr2,
upperAbsolutePowerArr2)
```

The key feature that enables overlapped measurements in RFmx is a feature called Named Results. In the previous example, both the first and second ACP measurements are initiated and tagged with a name. The RFmxSpecAnMX.Initiate method is called first with the ACP measurement enabled and the result name as ACP_Results_1, and then called again with the second ACP measurement enabled and the result name as ACP_Results_2. The results of these two measurements are stored independently from each other, and can be fetched in any sequence after the measurements are complete using the same result names passed to Initiate method. The result name passed to the Fetch method can be created using a set of Build String Utility static methods provided by RFmx.

In the above example, RFmxSpecAnMX.BuildResultString method is used to create a result string “result::ACP_Results_1” and passed to the Fetch method. Measurement results are referenced by the result name, measurement type, and the signal instance. If you do not specify any result name, the default result name (“”) is used. The following characters are not allowed in signal name and result name: ()*+,-./{} !"#%&'";<=>?@[\\^`|~ and any non-printable character.

When using overlapped measurements, you must specify the result name in Initiate and then use that name to fetch the results. If you reuse the same result name, the previous named results with that name are overwritten. This behavior applies to the default result name (“”) as well. When you initiate a measurement RFmx clears out all previous results with the same result name passed to Initiate; this keeps you from accidentally fetching a result from a previous Initiate.



Note Only results with the same name in the default or named signal configuration passed to Initiate are cleared.

For example, if you call Initiate with Selector String set to “signal:MySignal” and Result Name set to “MyResult”, Initiate will clear out all results named “MyResult” that were previously initiated with the signal “MySignal”. As mentioned, RFmx provides a set of Build String Utility static methods to create named result strings. Below code snippet shows RFmxSpecAnMX.BuildOffsetString2 method to create a result string and fetch result for offset 1.

C#

```
selectorString = RFmxSpecAnMX.BuildResultString("ACP_Result");
specAn.Initiate(selectorString, ""); offsetResultString =
RFmxSpecAnMX.BuildOffsetString2(selectorString, 1);//
offsetResultString = "result::ACP_Result/offset1"
specAn.Acp.Results.FetchOffsetMeasurement(offsetResultString,
timeout, out lowerRelativePower, out upperRelativePower, out lowerAbsolutePower,
out upperAbsolutePower);
```

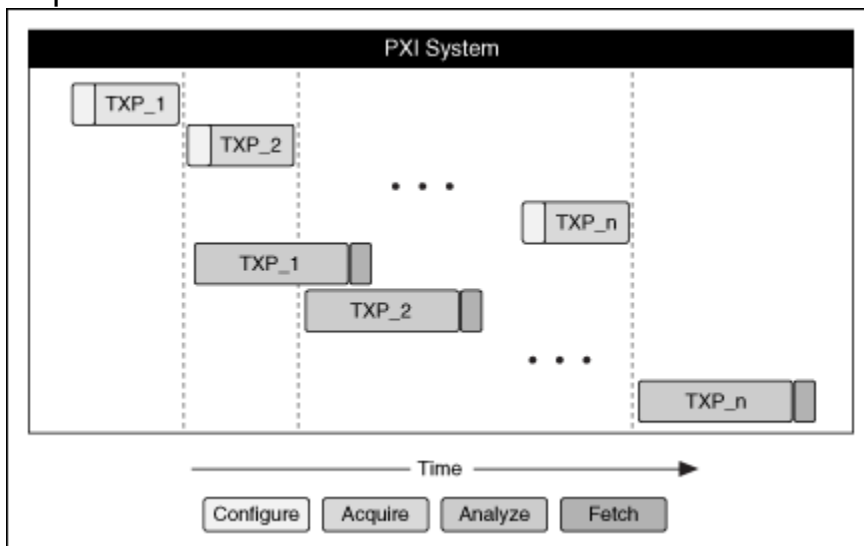
VB

```
selectorString = RFmxSpecAnMX.BuildResultString("ACP_Result")
specAn.Initiate(selectorString, "")
offsetResultString = RFmxSpecAnMX.BuildOffsetString2(selectorString, 1)
' offsetResultString = "result::ACP_Result/offset1"
specAn.Acp.Results.FetchOffsetMeasurement(offsetResultString, timeout, out
lowerRelativePower,
out upperRelativePower, out lowerAbsolutePower, out upperAbsolutePower);
```

The benefit of the overlapped measurements technique improves with more measurements in your sequence. In fact, there is no limit on the number of

measurements you can overlap. However, as the number of named results increases consider deleting the measurement results that have been fetched to conserve process memory space. The named results can be deleted using the `RFmxSpecAnMX.ClearNamedResult` method. Once a Named Result is cleared all the measurement results for that Named Result will be erased from the process memory of your application. If all Named Results need to be cleared from the memory `RFmxSpecAnMX.ClearAllNamedResults` method can be called.

Refer to the `RFmxSpecAnTxpMultipleResultNames` example application installed by RFmx SpecAn which conducts multiple TxP measurements at different frequencies and clears the named result for TxP measurement once it has been fetched. The example can be found at: `Users\Public\Documents\National Instruments\RFmx\SpecAn\Examples\DotNET\VS2010\Advanced\RFmxSpecAnTxpMultipleResultNames`. The following figure demonstrates the sequence of events for the mentioned TxP measurement example.



The actual time it takes to acquire the samples and analyze the TxP measurements may vary based on your system performance and measurement settings, however the sequence of events (configure, acquire, etc.) will remain the same.

RFmx will automatically start the analysis or processing of the measurement immediately after the samples for the measurement have been acquired. Once the processing of the measurement is complete, measurement results can be fetched. Depending on the computational complexity of the measurement, overlapped measurements technique can have a huge impact on overall test time.

Measurement parallelism is a known technique to reduce test time; RFmx makes this easier by managing threads and results so you can develop and realize the benefit of PXI faster.

Creating a Session

RFmx exposes a constructor and two static methods to create a session. Use either the constructor or one of the methods to create a session. You must invoke the constructor only once per resource name, whereas you can call the static method `GetSession` any number of times on the same resource name. The `GetSession` method returns the same session if the session exists for a given resource name.

```
public RFmxInstrMX(string resourceName, string optionString)
public static RFmxInstrMX GetSession(string resourceName, string optionString)
public static RFmxInstrMX GetSession(string resourceName, string optionString, out
bool isNewSession)
```

C#

```
RFmxInstrMX instrSession1 = new RFmxInstrMX("Rfsa", "");
RFmxInstrMX instrSession2 = RFmxInstrMX.GetSession("Rfsa", string.Empty);
bool isNewSession;
RFmxInstrMX instrSession3 = RFmxInstrMX.GetSession("Rfsa", string.Empty, out
isNewSession);
```

VB

```
Dim instrSession1 As RFmxInstrMX = New RFmxInstrMX("Rfsa", "")
Dim instrSession2 As RFmxInstrMX = RFmxInstrMX.GetSession("Rfsa", String.Empty)
Dim isNewSession As Boolean
Dim instrSession3 As RFmxInstrMX = RFmxInstrMX.GetSession("Rfsa", String.Empty,
isNewSession)
```

Closing a Session

RFmx provides the following methods to close a session.

```
public void Dispose() public void Close() public void
```

ForceClose()

The Dispose and Close methods perform a reference counted close.

The following snippets show an example of closing a session using the Close method.

C#

```
//ref count is 0 RFmxInstrMX InstrSession1 = RFmxInstrMX.GetSession("Rfsa","");
RFmxInstrMX InstrSession2 = RFmxInstrMX.GetSession("Rfsa","");
RFmxInstrMX InstrSession3 = RFmxInstrMX.GetSession("Rfsa",""); //ref count is 3
InstrSession1.Close(); //ref count is 2. Session is not disposed yet.
InstrSession2.Close(); InstrSession3.Close(); //Session is closed when ref count
hits 0.
```

VB

```
'ref count is 0 Dim InstrSession1 As RFmxInstrMX = RFmxInstrMX.GetSession("Rfsa",
"")
Dim InstrSession2 As RFmxInstrMX = RFmxInstrMX.GetSession("Rfsa", "")
Dim InstrSession3 As RFmxInstrMX = RFmxInstrMX.GetSession("Rfsa", "") 'ref count is
3
InstrSession1.Close() 'ref count is 2. Session is not disposed yet.
InstrSession2.Close()
InstrSession3.Close() 'Session is closed when ref count hits 0.
```

The ForceClose method closes a session irrespective of the reference count. The following snippets show an example of closing a session using the ForceClose method.

C#

```
//ref count is 0 RFmxInstrMX InstrSession1 = RFmxInstrMX.GetSession("Rfsa","");
RFmxInstrMX InstrSession2 = RFmxInstrMX.GetSession("Rfsa","");
RFmxInstrMX InstrSession3 = RFmxInstrMX.GetSession("Rfsa",""); //ref count is 3
InstrSession1.ForceClose(); //Session is closed irrespective of reference count.
```

VB

```
'ref count is 0 Dim InstrSession1 As RFmxInstrMX = RFmxInstrMX.GetSession("Rfsa",
```

```

""
Dim InstrSession2 As RFmxInstrMX = RFmxInstrMX.GetSession("Rfsa", "")
Dim InstrSession3 As RFmxInstrMX = RFmxInstrMX.GetSession("Rfsa", "") 'ref count is
3
InstrSession1.ForceClose() 'Session is closed irrespective of reference count.

```

RFmx Analysis-Only Mode

The RFmx Analysis-Only Mode further extends the functionality and flexibility of the RFmx API to handle the following use cases:

- Perform RF measurements with hardware that RFmx does not natively support
- Perform RF measurements on simulated waveforms
- Perform RF measurements on waveform data that was acquired previously and saved to disk

In the RFmx Analysis-Only Mode, time domain I/Q data or spectrum data is passed to the RFmxSpecAnMX.AnalyzeIQ method that performs the RF measurements that you specify on the raw waveform data.

Time domain measurements such as ModAcc/EVM or CCDF only accept complex I/Q data. Spectral measurements such as ACP, SEM, or CHP accept either complex I/Q data or spectrum data which must be in units of dBm. The f_0 in the spectrum data passed to the RFmxSpecAnMX.AnalyzeIQ method is calculated using the following formula:

$$f0_rfmx = f0_driver - fc$$

where,

$f0_rfmx$ is start frequency of the spectrum passed to the RFmxSpecAnMX.AnalyzeSpectrum method

fc is the center frequency of spectral acquisition

$f0_driver$ is the start frequency of the spectrum

In addition, if hardware is used to acquire the data, the input data provided to the RFmxSpecAnMX.AnalyzeIQ method must meet or exceed the minimum requirements

of the RFmx recommended acquisition settings. For example, the sample I/Q rate can be higher than recommended sample rate, acquisition length can be longer than the recommended acquisition length, spectrum span can be wider than the recommended acquisition span, and so on.

The recommended acquisition settings are calculated based on the user-defined measurement configuration, and can be accessed through the RFmxInstr Get/Set methods in C/.NET, or by getting the values of the equivalent attributes in LabVIEW.

When using the RFmx Analysis-Only Mode, the RFmx driver ignores RFmx hardware settings such as reference level and attenuation. The only RF hardware setting that is not ignored is the center frequency, since it is needed for spectral measurement traces as well as some of the wireless standard-specific measurements, such as ACP and SEM, which define different configurations or limits based on the center frequency and the value which you set for the BandClass property.

In the following example, time-domain transmit power (TxP) measurement is performed on a simulated QAM waveform. This example completes the following operations in the Analysis-Only Mode:

- **OPEN**—Opens the RFmx session with an empty hardware resource name and with “AnalysisOnly=1” string wired to option string input of the RFmxInstr constructor.
- **CONFIGURE**—Configures the TxP measurement as if RFmx was used with hardware.
- **ANALYZE**—Passes a simulated waveform to the RFmxSpecAnMX.AnalyzeIQ method which is called instead of the RFmxInstrMX constructor that is used when using RFmx with the built-in hardware support.
- **FETCH**—Fetches the measurement results using the same RFmx methods used to fetch results while using RFmx with the built-in hardware support.
- **CLOSE**—Closes the RFmx session.

C#

```
// OPEN Session without resource name but with option string set to
"AnalysisOnly=1"
RFmxInstrMX instrSession = new RFmxInstrMX("", "AnalysisOnly=1");
// Get SpecAn signal
```

```

RFmxSpecAnMX specAn = instrSession.GetSpecAnSignalConfiguration();
// CONFIGURE RFmx measurement using regular RFmx methods
specAn.Txp.Configuration.ConfigureMeasurementInterval("", measurementInterval);
specAn.Txp.Configuration.ConfigureRbwFilter("", rbw, rbwFilterType, rrcAlpha);
specAn.SelectMeasurements("", RFmxSpecAnMXMeasurementTypes.Txp, true);
// Pass Waveform to AnalyzeIQ method
specAn.AnalyzeIQ("", "", iq, reset, out averagingDone);
// Fetch Results using regular RFmx methods
specAn.Txp.Results.FetchMeasurement("", 10.0, out averageMeanPower,
out peakToAverageRatio, out maximumPower, out minimumPower);
// CLOSE Session
specAn.Dispose(); instrSession.Close();

```

VB

```

' OPEN Session without resource name but with option string set to "AnalysisOnly=1"
Dim instrSession As New RFmxInstrMX("", "AnalysisOnly=1")
' Get SpecAn signal
Dim specAn As RFmxSpecAnMX = instrSession.GetSpecAnSignalConfiguration()
' CONFIGURE RFmx measurement using regular RFmx methods
specAn.Txp.Configuration.ConfigureMeasurementInterval("", measurementInterval)
specAn.Txp.Configuration.ConfigureRbwFilter("", rbw, rbwFilterType, rrcAlpha)
specAn.SelectMeasurements("", RFmxSpecAnMXMeasurementTypes.Txp, True)
' Pass Waveform to AnalyzeIQ method
specAn.AnalyzeIQ("", "", iq, reset, averagingDone)
' Fetch Results using regular RFmx methods
specAn.Txp.Results.FetchMeasurement("", 10.0,
averageMeanPower, peakToAverageRatio, maximumPower, minimumPower)
' CLOSE Session
specAn.Dispose() instrSession.Close()

```

Another way to use the RFmx Analysis-Only Mode is in combination with another driver, such as NI-RFSA, that acquires the waveform which is then passed to RFmx for analysis. In this case, the waveform acquisition, performed by the driver, is configured based on the recommended acquisition settings calculated by RFmx. The following example demonstrates this approach.

C#

```

//Open //Configure RFmx ACP measurement //RFmx COMMIT
specAn.Commit("");

```

```

// Configure Acquisition
maxDeviceInstantaneousBandwidth
=rfsaSession.DeviceCharacteristics.MaxInstantaneousBandwidth;
instrSession.GetRecommendedNumberOfRecords("", out numberOfRecords);
instrSession.GetRecommendedSpectralAcquisitionSpan("", out
spectralAcquisitionSpan); if
(maxDeviceInstantaneousBandwidth ≥ spectralAcquisitionSpan) {
// IQ Acquisition
} Else {
// Spectral Acquisition
rfsaSession.Configuration.AcquisitionType = RfsaAcquisitionType.Spectrum;
instrSession.GetRecommendedSpectralFftWindow("", out recommendedFftWindow);
instrSession.GetRecommendedSpectralResolutionBandwidth("", out
resolutionBandwidth);
switch (recommendedFftWindow)      {
case
RfmxInstrMXRecommendedSpectralFftWindow.FlatTop:
fftWindowType = RfsaFftWindowType.FlatTop;
break;
//other cases
default:
fftWindowType = RfsaFftWindowType.Uniform;
break;    }
double updatedCenterFrequency = rfsaSession.Configuration.Spectrum.CenterFrequency;
rfsaSession.Configuration.Spectrum.ResolutionBandwidth = resolutionBandwidth;
rfsaSession.Configuration.Spectrum.FftWindowType = fftWindowType;
rfsaSession.Configuration.Spectrum.ResolutionBandwidthType =
RfsaResolutionBandwidthType.RbwBinWidth;
rfsaSession.Configuration.Spectrum.PowerSpectrumUnits = RfsaPowerSpectrumUnits.dBm;
rfsaSession.Configuration.Spectrum.Span = spectralAcquisitionSpan;    for (int i =
0; i< numberOfRecords; i++)
    {          reset = (i == 0) ? true : false;
RfsaSpectrumInfo spectrumInfo;
double[] spectrumData =
rfsaSession.Acquisition.Spectrum.ReadPowerSpectrum(new PrecisionTimeSpan(timeout),
out spectrumInfo);
//READ the Power Spectrum from RFSA Spectrum
<float> powerSpectrum = CreateSpectrum(spectrumData, spectrumInfo);
powerSpectrum.StartFrequency = powerSpectrum.StartFrequency -
updatedCenterFrequency;
// Pass Waveform to AnalyzeIQ method
specAn.AnalyzeSpectrum("", "", powerSpectrum, reset, out averagingDone);    } }
//Fetch ACP Measurement Results // Close

```

VB

```

`Open `Configure RFmx ACP measurement `RFmx COMMIT specAn.Commit("")
` Configure Acquisition
maxDeviceInstantaneousBandwidth=rfsaSession.DeviceCharacteristics.MaxInstantaneousB
andwidth
instrSession.GetRecommendedNumberOfRecords("", out numberOfRecords)
instrSession.GetRecommendedSpectralAcquisitionSpan("", out spectralAcquisitionSpan)
If
(maxDeviceInstantaneousBandwidth ≥ spectralAcquisitionSpan)
` IQ Acquisition Else
`Spectral Acquisition
rfsaSession.Configuration.AcquisitionType = RfsaAcquisitionType.Spectrum
instrSession.GetRecommendedSpectralFftWindow("", out recommendedFftWindow)
instrSession.GetRecommendedSpectralResolutionBandwidth("", out resolutionBandwidth)
Select Case recommendedFftWindow
Case RFmxInstrMXRecommendedSpectralFftWindow.FlatTop
fftWindowType = RfsaFftWindowType.FlatTop
Exit Select
`Other cases
Case Else fftWindowType = RfsaFftWindowType.Uniform
Exit
End
Select updatedCenterFrequency = rfsaSession.Configuration.Spectrum.CenterFrequency
rfsaSession.Configuration.Spectrum.ResolutionBandwidth = resolutionBandwidth
rfsaSession.Configuration.Spectrum.FftWindowType = fftWindowType
rfsaSession.Configuration.Spectrum.ResolutionBandwidthType =
RfsaResolutionBandwidthType.RbwBinWidth
rfsaSession.Configuration.Spectrum.PowerSpectrumUnits = RfsaPowerSpectrumUnits.dBm
rfsaSession.Configuration.Spectrum.Span = spectralAcquisitionSpan
For i As Integer = 0
To numberOfRecords - 1
reset = If((i = 0), True, False)
Dim spectrumInfo As RfsaSpectrumInfo
Dim spectrumData As Double() =
rfsaSession.Acquisition.Spectrum.ReadPowerSpectrum(New PrecisionTimeSpan(timeout),
spectrumInfo)
`READ the Power Spectrum from RFSA
Dim powerSpectrum As Spectrum(Of Single) = CreateSpectrum(spectrumData,
spectrumInfo)          powerSpectrum.StartFrequency =
powerSpectrum.StartFrequency - updatedCenterFrequency
` Pass Waveform to AnalyzeIQ method
specAn.AnalyzeSpectrum("", "", powerSpectrum, reset, averagingDone)
Next
End If
`Fetch ACP Measurement Results
`Close

```

The programming flow for using the RFmx Analysis-Only Mode in combination with NI-RFSA closely resembles the flow used when analyzing a simulated waveform using RFmx. However, in addition to opening and configuring RFmx session, you can also open and configure an NI-RFSA session.

NI-RFSA performs the actual acquisition of the waveform based on the recommended acquisition settings that are passed on from RFmx session to RFSA session using get/sets in C/ .NET or setting the property nodes (LabVIEW). You must call the `RFmxSpecAnMX.AnalyzeIQ` method to calculate the recommended acquisition settings based on the user-defined RFmx measurement configuration before recommended acquisition settings are passed on to the RFSA.

After NI-RFSA acquires the waveform, `RFmxSpecAnMX.AnalyzeIQ` method is called to perform user-defined measurements on the raw waveform data. Once the analysis of the waveform is complete, you can fetch measurement results and close both RFmx and RFSA sessions. Refer to the following examples for more information about using the RFmx Analysis-Only Mode to perform RF measurements:

- `<LabVIEW directory>\examples\RFmx\SpecAn\RFmxSpecAn ACP (Analysis Only).vi`
- `<Users>\Public\Documents\National Instruments\RFmx\SpecAn\Examples\C\RFmxSpecAnAcpAnalysisOnly`
- `<Users>\Public\Documents\National Instruments\RFmx\SpecAn\Examples\DotNET\VS2010\RFmxSpecAnAcpAnalysisOnly`



Note

- Measurements such as harmonics and spurious emissions that require multiple acquisitions with different configurations are not supported in the Analysis-Only Mode. The Dynamic Range ACP measurement method is also not currently supported.
- For the Analysis-Only mode, you need to adjust the t_0 value of the I/Q waveforms according to pre-trigger samples present in the waveform before passing the waveform to the RFmx Analyze (IQ) method.



Note Use the following recommended settings while analyzing the LTE waveforms in the Analysis-Only mode:

- If the waveform data has been acquired with local oscillator (LO) at the center of RF signal bandwidth, set the value of `RFmxLteMX.SetAcquisitionBandwidthOptimizationEnabled` method to **True**; otherwise, set the value of this method to **False**.
- If you set the `RFmxLteMXModAccConfiguration.SetCommonClockSourceEnabled` method to **True**, the value of the `RFmxLteMX.SetCenterFrequency` method configured in RFmxLTE session should be same as the center frequency used to acquire the waveform from the hardware.