Level 2

Capstone Project

Simulated Beamformer

Contents

Project Overview

Background

New emerging wireless technologies are leveraging antenna arrays to augment existing functionality or to extend coverage into new frequency bands. For example, Bluetooth 5.1 adds support for direction-finding features by extending the Bluetooth packet with a constant-tone extension (CTE). This CTE portion of the signal is sampled by the receiver with multiple antenna elements in Angle of Arrival (AoA) mode, or is transmitted by multiple antenna elements in Angle of Departure (AoD) mode. When receiving a signal in an AoA configuration, during signal acquisition the Access Point (AP) will rapidly switch between each antenna while sampling. Each antenna will receive the same signal at a unique phase and amplitude. Similarly, for an AoD configuration, the transmitter will quickly switch between each antenna during transmission of the CTE field. In either configuration, with this information relative position can be determined by analyzing the amplitude and phase differences produced by the physical position differences between each antenna element.

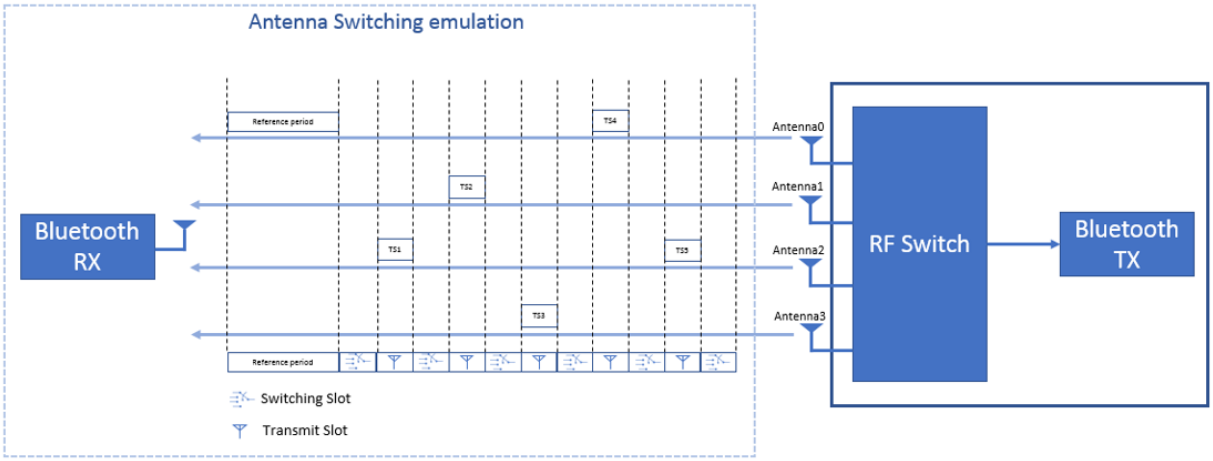

Figure 1: Antenna switching emulation in the NI Bluetooth Toolkit for Bluetooth 5.1 testing

Another emerging technology, 5G New Radio (NR), adds support for new mmWave frequency bands. At mmWave, the free-space path loss becomes an even more significant impediment to signal quality, requiring new ways of delivering appropriate power to the UE. The NR specification addresses this by using phased antenna arrays to create beamforming devices. These devices can “steer” a beam by shifting the phase and amplitude of individual antenna elements, leading to the constructive and destructive interference of signals. This allows a base station to focus energy in the direction of the UE while attenuating the signal in undesired directions.

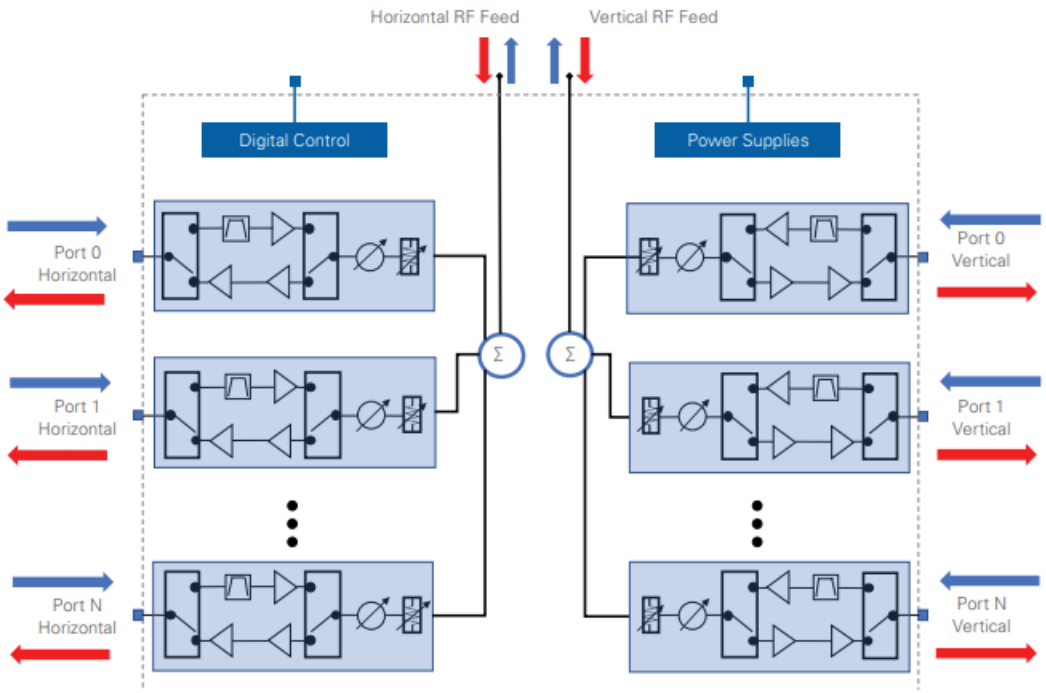

Figure 2: 5G NR RF to RF beamformer device (source)

In both cases, a common test for these devices is to measure the phase and amplitude of an acquired signal versus time. For beamformers and phase shifting devices, it is common to step through various phase and amplitude stages for an antenna element and measure the relative phase and amplitude for the step. In this way, the antenna element and phase shifter can be validated to function correctly when assigned a certain phase and amplitude offset.

Design

In this project, you are tasked with developing measurement code to perform relative phase and amplitude measurements for a customer's beamforming device. However, the customer has not yet completed early samples of the device and does not have any units available for testing. Once you receive the DUT, you will have a short window of time in which to integrate the device into your test program before it must be returned.

In order to ensure that you will be able to meet the customer's aggressive timelines, you will develop a simulated beamformer device using an NI Vector Signal Generator. With this simulated device, you will be able to validate your measurement code prior to receiving the customer's DUT. In order to ensure a rapid integration of the customer's DUT, you will develop your code in such a way that you can substitute the customer's DUT in your application without changing any of the measurement code. Thus, you will create a general model of a beamformer and implement a specific instance using NI RFSG.

To configure and control the measurement, you will build an application similar to the one shown in Figure 3. This application will enable the user to configure a table of relative phase and amplitude offsets which will be communicated to the beamformer. After configuring the measurement and offsets, the application will communicate the requested offsets to the DUT and measure the results.

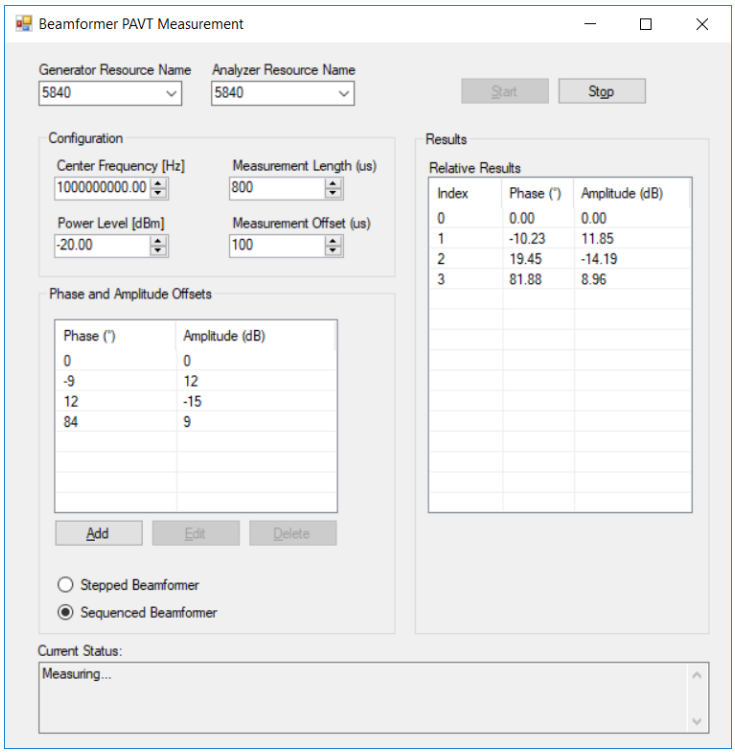

Figure 3: Sample beamformer test application

Measurements

As of RFmx 19.0, a measurement named PAVT (Phase and Amplitude Versus Time) has been added for phase and amplitude measurements of beamforming devices. If you are not already familiar with the measurement, review the documentation link to familiarize yourself with the available measurement configurations.

Getting Started

The project starting code is hosted on GitHub here. Follow the instructions in the README file to fork the public repository to your own GitHub account or an internal Azure DevOps repository. Use your forked repository to collaboratively develop this project with your partner.

Reviewing the Files

A solution file has been provided to you named `L2CapstoneProject.sln` with stub code for you to get started. Take a moment to familiarize yourself with the provided code.

frmBeamformerPavtController.cs

This file contains the main interface for the application. Users will select devices and configure the measurements using this interface. Pressing the *Add* button opens *frmOffset*. You will modify the code so that *frmOffset* returns a phase and amplitude offset that will be logged in the *Phase and Amplitude Offsets* ListView.

frmOffset.cs

This form simply allows for the configuration of a single phase and amplitude offset point. It also has an edit mode which with your modifications can edit a configured offset when double-clicked, the enter key is pressed, or the user selects *Edit*.

Week 1: Beamformer Model

Tasks for the Week

- ☐ Develop a `PhaseAmplitudeOffset` type to be used throughout your application
- ☐ Complete an abstract beamformer model that will be used to create your simulated beamformer next week
- ☐ Integrate the `PhaseAmplitudeOffset` type that you create into the stub application

PhaseAmplitudeOffset Type

Throughout your code, you will need a consistent way of referring to a particular phase and amplitude offset. **Develop a new type named `PhaseAmplitudeOffset` that can store this information.**

The next two sections of this week's development will use this newly created type.

Beamformer Design

For the sake of this project, the focus will be on RF to RF beamformers, which do not include any frequency conversion. In other words, the device will be stimulated with a CW tone at a given frequency, and its response will be measured at the same frequency. Additionally, to simplify the project, you only need to consider a single element of the beamformer. Your model does not need to include code for switching to different antenna elements or incorporating other RF switching. You may assume that you have a direct connection to a single antenna port, and that antenna has already been selected on the device.

Your model will represent the composite control of the beamformer device. In other words, from the perspective of your code, your model should represent everything necessary to control the beamformer, even if multiple instruments are required to control it. For example, a beamformer that is powered using an NI DC Power instrument and commanded with an NI Digital Pattern instrument would still be modeled as a single device. In this case, the underlying code implements the functionality needed to control the two instruments.

Build a model representing the following beamformer types in your C# project.

Beamformer Types

Your model should be able to account for two primary categories of beamformers, defined for the sake of this project as "stepped" and "sequenced" beamformers.

Stepped Beamformers

Most beamformers fall into this category, in which the phase and amplitude will remain constant until a register is written to on the device indicating a new phase and amplitude offset. A trigger is sent either from the device itself or the controlling instrument to notify the signal analyzer that the new offset has been set to trigger the measurement.

Command	Description
Connect	Connect to the DUT (power on, write to register, etc.)
Write Offset	Writes a `PhaseAmplitudeOffset` to registers on the device.
Disconnect	Disconnect from the DUT (write registers, power off, etc.

Table 1: Sample commands for a stepped beamformer

Sequenced Beamformers

This type of beamformer can have a list of phase and amplitude offsets written to it as a sequence. Then, with a command, the device can step through the various requested states using its own internal hardware timing. Because this type of device can receive a list of phase and amplitude offsets at test time, this will be referred to as “dynamic”.

Alternatively, a stepped beamformer can act as a sequenced beamformer if the DUT control routine can be sequenced. For example, when controlling a beamformer using the NI Digital Pattern Instrument, a pattern can be generated prior to the test that executes a series of offset commands in a row. This will sweep various phase and amplitude offsets with hardware timing. This would transform a stepped beamformer into a sequenced beamformer, at least from the perspective of this model. In this configuration, the beamformer will be considered “static”, as it does not receive a list of commands at test time. Rather, a known configuration (such as referencing a specific pattern by name) is applied by the code to write the sequence to the DUT.

Command	Description
Connect	Connect to the DUT (power on, write to register, etc.)
Configure Sequence	Configures a sequence based on a list of `PhaseAmplitudeOffset` values (dynamic), or by loading a pre-configured sequence (static)
Initiate Sequence	Initiates a previously configured sequence
Abort Sequence	Aborts a running sequence
Disconnect	Disconnect from the DUT (write registers, power off, etc.

Table 2: Sample commands for a sequenced beamformer

Beamformer Diagrams

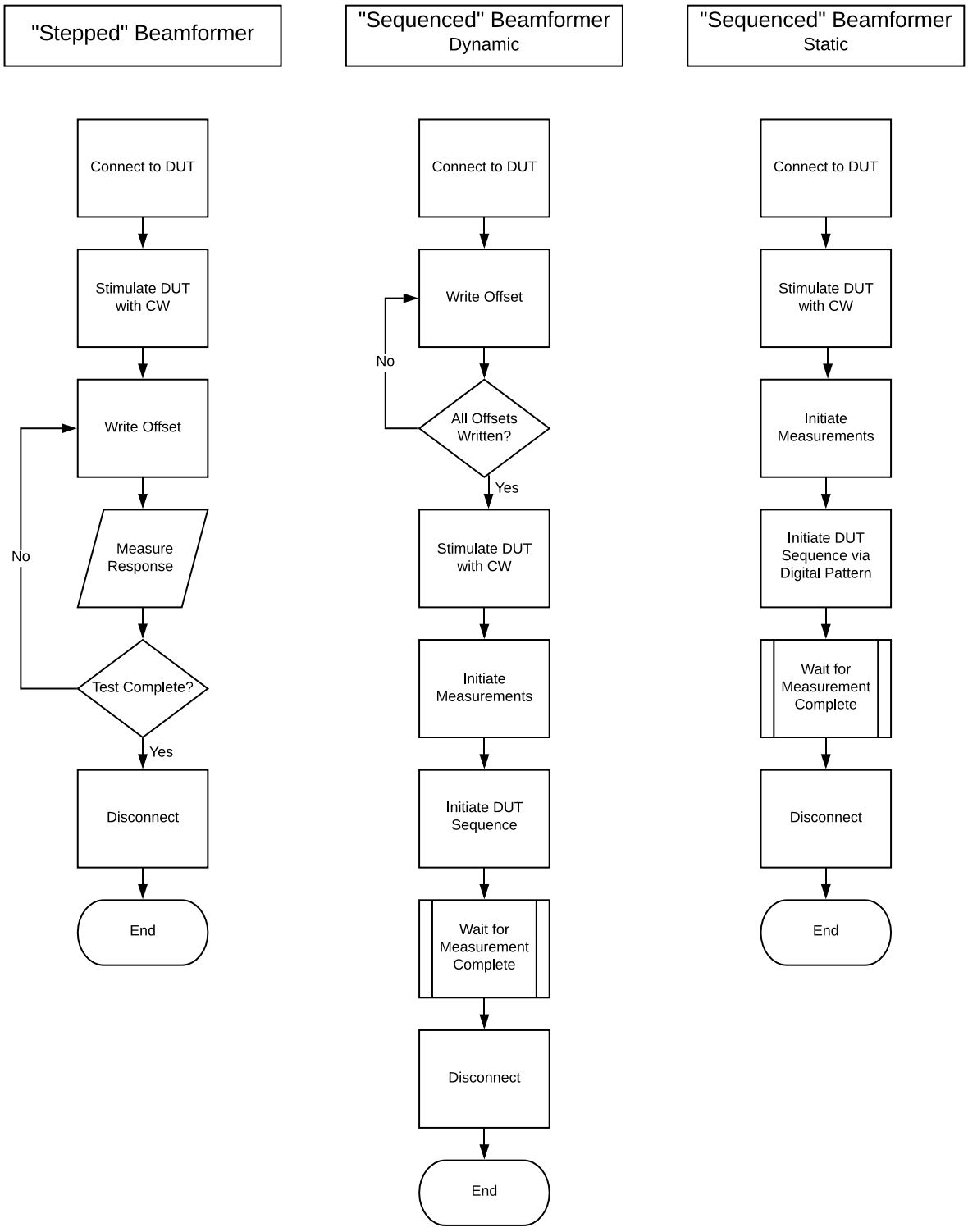

Figure 4: Beamformer Types

Integrating the Type

Using the `PhaseAmplitudeOffset` type developed previously, **complete the implementation of the Phase and Amplitudes Offsets editor in the main form**.

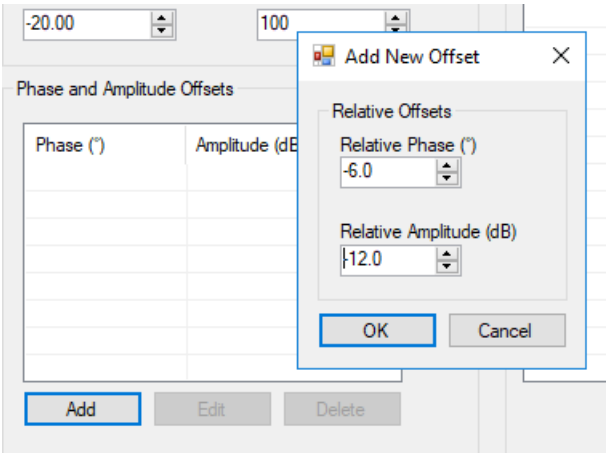

Figure 5: The phase and amplitude offsets table and editor

1) Update the `frmOffset` class included in the starting code to return a `PhaseAmplitudeOffset` object based on the values configured in the form controls when the user selects OK. *(Hint – a form is just a class, after all. Hence, you can add information to a class to be read by another class in exactly the same way as you would with any other class.)*
2) Modify the `AddOffset` function to add the newly created point to the ListView on the form if the user has selected the "OK" button.

   You will later need some way of passing this list of offsets to your measurement code, so be sure to consider how you will maintain the configured offsets to easily pass.
3) **(Optional)** Complete the functionality for the edit and delete buttons.

Week 2: Simulated VSG Beamformer

Tasks for the Week

- ☐ Implement a simulated beamformer using your beamformer model and RFSG
- ☐ Integrate and test your simulated beamformer in the Windows Forms application

Simulating the Beamformer with RFSG

Using the model that you created last week, build a simulated beamformer implemented using NI RFSG. You may choose to implement EITHER a stepped or dynamic sequenced beamformer.

Remember that the final test configuration is an RF to RF beamformer. Hence, your measurement code will already configure an RFSG session to stimulate the DUT with a CW prior to instructing the DUT to shift the phase and amplitude. Your simulated beamformer should be able to account for this.

Optional Challenge | Developing Both Types

Implement both a stepped and sequenced beamformer with NI RFSG.

There are multiple ways of approaching the design of the beamformer in RFSG. If you choose to implement this using custom arbitrary waveforms, see the notes below about working with NI waveforms in C#.

Creating NI Waveforms in C#

In LabVIEW, waveforms for RF applications are typically implemented using a LabVIEW cluster containing the start time of the waveform, the time between each sample, and an array of samples. The samples can be complex doubles or complex singles.

In C#, a few steps are required to create waveforms. First of all, the language has no native implementation of complex numbers. To address this, NI defines a custom value type (struct) for complex doubles, singles, and integers in the `NationalInstruments.Common` assembly. You can create the complex numbers using real and imaginary data components using the constructor:

```
public ComplexDouble(double real, double imaginary);
```

Alternatively, additional construction options are available using the static constructors:

```
public static ComplexDouble FromDouble(double real);

public static ComplexDouble FromPolar(double magnitude, double phase);
```

Next, waveforms are defined in a generic class `ComplexWaveform<TData>`, where `TData` can be `ComplexDouble`, `ComplexSingle`, or `ComplexInt16`.

Creating NI Waveforms in C# (cont.)

The simplest way to create a new waveform is to use the static constructor method:

```
public static ComplexWaveform<TData> FromArray1D(TData[] array);
```

This allows you to create a new waveform from an existing array of complex samples. Otherwise, to modify the data of an existing waveform you must invoke the `GetWritableBuffer()` method to access the waveform data.

The waveform now has IQ samples, but no sample rate defined. You will now need to create an object representing the sample interval using the `PrecisionTimeSpan` class. You can do this with a standard constructor or with the static constructor. The static constructor defines more options for time in seconds, milliseconds, etc.:

```
PrecisionTimeSpan dt = PrecisionTimeSpan.FromSeconds(1 / iqRate);
```

Finally, we can set the sample timing for the waveform. The waveform type includes support for irregular sample intervals, so you must explicitly define that the waveform has a regular sample interval with the timespan created. This is accomplished by setting the `PrecisionTiming` property of the waveform. This property is of type `PrecisionWaveformTiming` and must be created using the static constructor:

```
waveform.PrecisionTiming = PrecisionWaveformTiming.CreateWithRegularInterval(dt);
```

Your waveform is now complete and can be used successfully.

Optional Challenge | Named Sequences

For a sequenced beamformer, include an input for a sequence name and allow configurations of different sequences by name. This way, a test program could configure multiple sequences of different offsets during test initialization. During test execution, the code could rapidly switch between them to improve overall test time.

Integrating into the Application

Test your simulated beamformer by adding it to the controller GUI. Modify the code so that pressing the start button initializes an RFSG session and begins generating the phase and amplitude offsets configured by the user in the table. Validate the signal is generated correctly by using the PAVT measurement in the NI-RFmx SFP.

Week 3: Measurement Code

Tasks for the Week

- ☐ Develop your measurement code incorporating the generic model of the beamformer developed in week 1
- ☐ Integrate your measurement code into the application

Developing the Measurement Code

With your beamformer model created and a simulated device developed, you are now ready to develop and validate your measurement code. As a reminder, your goal is to develop and validate this measurement code so that when the customer's DUT is available, you can substitute the DUT into your measurement code without any changes. **Develop a `PavtMeasurement` class that executes the desired offset sequence with the beamformer and returns the results to the caller.** This class should implement the functionality as shown in Figure 6 on the next page.

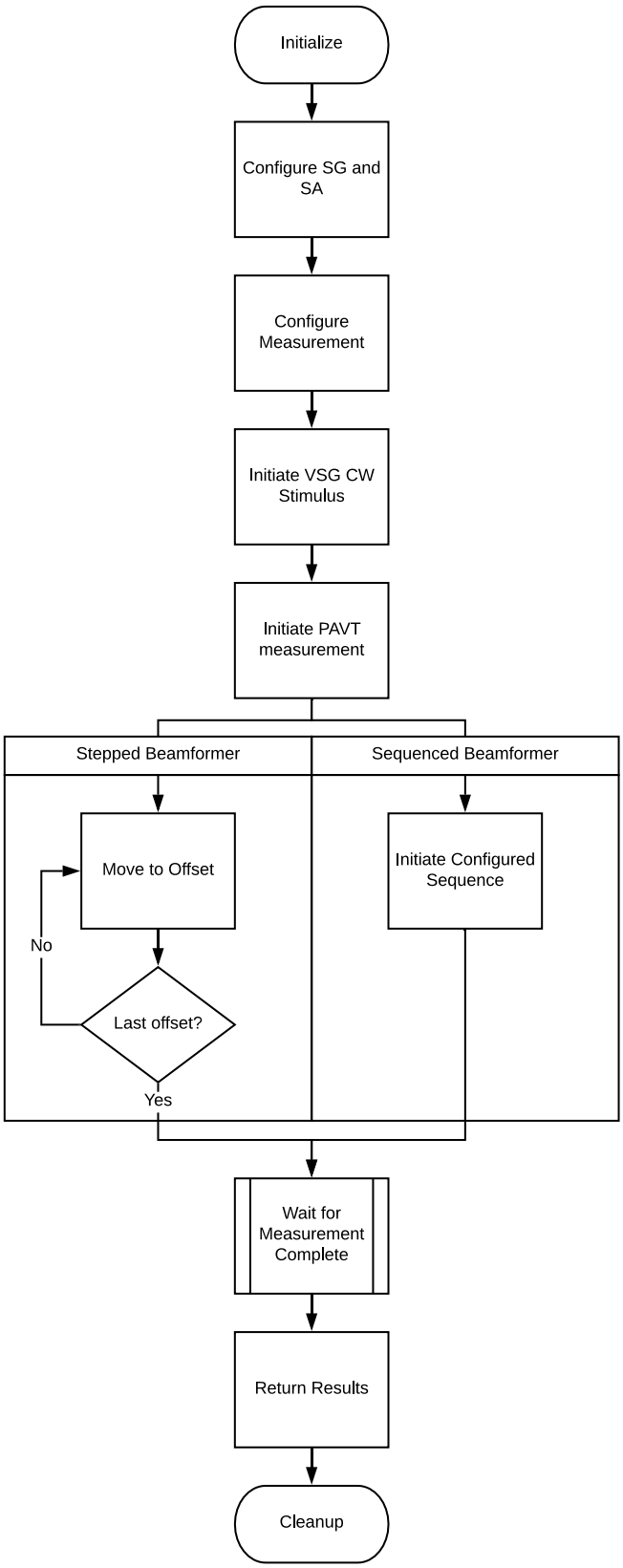

Figure 6: PAVT Measurement Class Process

Developing the RFmx Code

If you are unfamiliar with the PAVT measurement code, examine the RFmx PAVT sample code located at `C:\Users\Public\Documents\National Instruments\NI-RFmx\SpecAn\Examples\DotNET\VS2010\RFmxSpecAnPavt\cs`.

For a stepped beamformer, you will need to use the trigger measurement location type. The sequenced beamformer should use the time location type. The measurement should be configured with the number of segments defined by the user in the offsets table.

RFmx Signals in C#

In LabVIEW, you use the same session wire returned by the *RFmx Initialize* VI regardless of which personality you intend to use. Additionally, RFmx signals are created by calling the *Create Signal Configuration* VIs. These VIs return a special selector string used to target that specific signal. Using this string, you can target specific named RFmx signals using the same RFmx session handle.

This behavior is very different in C#. All signals are represented as objects, whether the unnamed default signal for a personality, or a specifically named signal within a personality. Using a specific personality and creating a named signal configuration follow the same process. The specific RFmx personality adds an extension method to the `RFmxInstrMX` class with the format `instrObj.Get[Personality]SignalConfiguration`, where `instrObj` is the `RFmxInstrMX` object created by the constructor, and `[Personality]` will be replaced by the specific personality you are trying to use. There are two overloads available for this method: one with no parameters, and another with a string parameters. The method without parameters returns the default signal object for the personality. The method with a string parameter returns a new named signal object; if the named signal already exists, it returns a reference to the existing object. This object has all of the properties and methods needed for configuring and measuring with that personality. Signals have a `Dispose()` method to destroy the signal configuration – **this does not close the instrument session.**

One important implication of this is that the **RFmx selector strings are used much less in C#**. The documentation for most configuration methods specifies to pass an empty string to the selector string input. They are primarily used in C# for retrieving specific named results, but are also used for configuration of specific properties (such as specific carriers).

Optional Challenge | Named Signals

Add an input for a sequence name when configuring the measurement and create a new RFmx SpecAn signal for this sequence name. This way, the test program can rapidly switch between different configurations for faster execution.

If you completed the earlier challenge for implementing a sequence name with the sequenced beamformers, the RFmx signal name and beamformer sequence name should match.

Integrating into the Application

Integrate the `PavtMeasurement` class into the beamformer measurement application to make a single PAVT measurement. Pressing the Start button should initialize the instruments, configure the measurement, initiate all devices, and report the measurement results to the *Relative Results* ListView.

Optional Challenge | Continuous Measurements

Instead of making a single measurement and returning, develop your controller code such that pressing the start button initiates measurements that continue repeatedly until the stop button is pressed. You will need to ensure that the UI code is not blocked by this looping measurement so that the stop or exit buttons can be pressed.

Optional Challenge | Asynchronous Code

Operations such as initializing driver sessions and waiting on the signal acquisition and measurement can take time. When this code is called from the Windows Form, this extra time can create an undesirable "locking" of the UI while the event handler is executing. To avoid this, asynchronously execute the driver initialization and measurement waiting such that the UI can continue processing events even while these tasks are executing.