Vantage
Sequence Programming Manual

Last modification February 26, 2021

Return to TOC

Table of Contents

1. Introduction

1.1 Overview of the Verasonics Research System

The Verasonics Vantage Research System consists of the Verasonics Data Acquisition Hardware connected to a Host Controller Computer. The Acquisition Hardware contains electronic modules for multi-channel transmit waveform generation, analog receive signal amplification and filtering, digital signal processing and scan sequencing. The computer contains software modules implemented within the Matlab programming environment that allow the user to program and run sequences of actions, referred to as events, in both the hardware and software environments. A sequence of events is programmed by writing a Matlab script (referred to as a Setup script) that generates a collection of objects that can be loaded into the system. The objects in a sequence are defined using Matlab structures which specify sets of attributes. When the Setup script executes, it creates a binary data file containing all the object structures and programming information which is stored in the format of a Matlab .mat file.

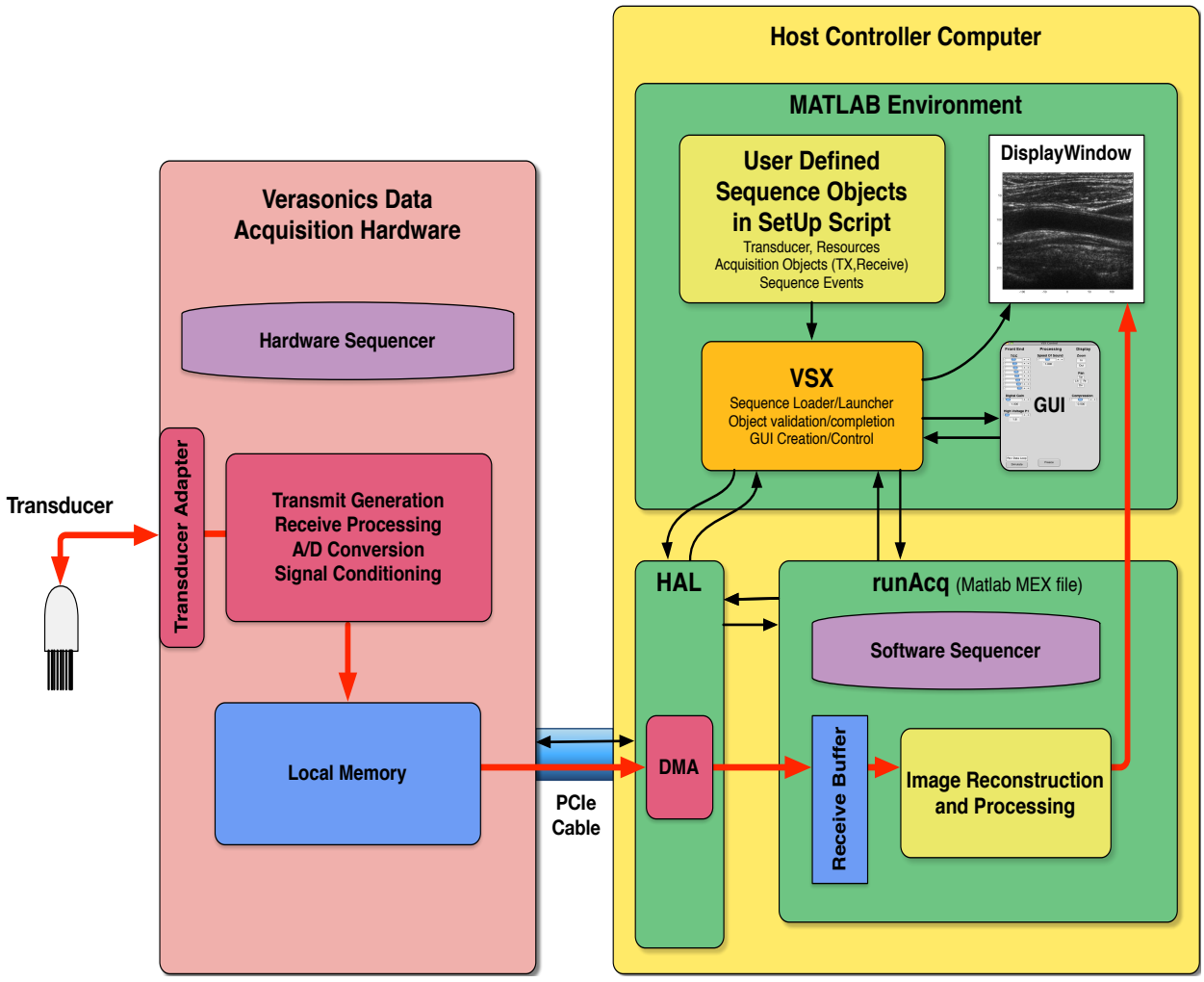


Fig. 1.1 Simplified system architecture of Verasonics Research System. The flow of ultrasound data is indicated by the bold red line starting with the transducer and ending with the display.

This .mat file can then be loaded into the system by a loader program (named 'VSX' for 'Verasonics Script eXecution') that runs as a typical script in the Matlab environment and communicates with the hardware via a Hardware Abstraction Layer (HAL). The loader program performs some checks on the structures read from the .mat file, adds some attributes needed for programming the Vantage hardware, and opens up both a GUI and a display window (if specified). Control and processing are then handed off to a Matlab External function written in the C programming language, named 'runAcq'. The sequence of events is then loaded into both the hardware and software sequencers and the execution started.

During sequence execution, almost all processing and display functions are handled by the runAcq function. This function contains the software sequencer and handles all interactions with the hardware during run time. At the end of an event sequence, or after a jump back to the beginning of the sequence, or at designated intermediate events in a sequence, runAcq returns control to the VSX program in the Matlab environment, to allow execution of user interface functions or user programs. If there are no actions to perform in the Matlab environment, VSX calls runAcq again, and the sequence repeats from the start, or continues from where it returned to Matlab.

While an event sequence is executing, changes can be made to acquisition and processing by the use of Matlab GUI controls that interact with VSX and runAcq. Basic controls such as Time Gain Control sliders, transmit power, pan, zoom and freeze, etc. are automatically added to the GUI window. The user can add additional GUI controls that can be specified in the Matlab programming script, along with the callback functions that execute when the GUI control is changed. The many example scripts help illustrate how to implement new control functions.

During the execution of the sequence events, it is also possible to have the system call a user provided Matlab function at specific points in the sequence. The user function can access acquired and processed data to perform some unique function, create their own data displays, or return the processed data to the processing software. This provides an easy way to alter the processing of the system, or extract information for offline processing.

When the user wishes to quit a loaded and running sequence, the GUI window is closed and control returns to the Matlab environment on the next return-to-Matlab by runAcq. The display window is not closed, and is available for measurements, annotation and archiving. The buffers used by the system to store acquisition data, reconstructed pixel magnitude and complex signal data, and image data are all available in the Matlab environment for inspection or further analysis with a user provided Matlab script.

The following sections describe the various structures and objects that are used to program the Verasonics system. The best way to learn how to specify all the structures needed to program a Sequence is to study one of the example scripts, such as 'SetUpL11_4vFlash.m'. By analyzing the example scripts and learning how the objects are specified, one can come to a better understanding of the programming objects, and how they are used.

It is strongly recommended that the new user begin by going through the Sequence Programming Tutorial, which can be found in the Documentation folder. The tutorial takes you through a step-by-step building of a sequence program, and contains more

information on the philosophy behind some of the programming techniques. This Sequence Programming Manual is generally aimed at providing more detailed reference information on the system and programming structures, but additional sections address more specific uses of the Hardware and Software, such as techniques for sequence control, signal averaging and high frequency acquisitions.

The best way to get started is to develop a simple sequence program such as provided in the Tutorial to get familiar with the basics of system operation. One can then start to exam and analyze various example scripts in the ExampleScripts folder to become familiar with more advanced programming techniques. When you have a new program to develop, it is almost always easier to start with an existing example script that comes close to doing what you want to do and modifying it.

It definitely helps to have prior experience with ultrasound signal processing and basic programming skills in Matlab. If you need help in these areas, there are many books available as reference, such as “Diagnostic Ultrasound Imaging: Inside Out,” 2nd Edition by Thomas L. Szabo, Elsevier Academic Press, and “Matlab: A Practical Introduction to Programming and Problem Solving,” 3rd Edition (4th edition to be released July, 2016) by Stormy Attaway, Elsevier.

1.2 Structure of the Matlab Setup Script

The typical Matlab Setup script contains the following elements, which should generally be provided in the order given.

Global System Objects (Non-Sequence Objects)

Script Info: Comments for script name, program information, software revision compatibility, etc.

Unique Script Parameters: Parameters for defining other structures, attributes to save as presets.

Global Resource Parameters: Hardware resources, speed of sound, media attenuation, simulation mode, error reporting.

Transducer Specification: Trans structure definition.

Pixel Data Specification (optional): PData and PData.Region structures (Only required if performing image reconstruction)

Resource Buffers and Display Window (if needed): Resource.RcvBuffer, Resource.InterBuffer, Resource.ImageBuffer, Resource.DisplayWindow.

Media Specification (optional): Media structure.

Sequence Objects

Transmit Specifications: TW and TX structure, TPC profile (optional).

Time Gain Control Specifications: TGC structure.

Receive Specifications: Receive and RcvProfile (optional) structures.

Reconstruction Specifications (optional): Recon and ReconInfo structure, (needed

only if performing image reconstruction).

Processing Specifications (optional): Process structure (needed if using internal or external processing functions).

Sequence Control Specifications: SeqControl structure.

Sequence Event Specifications: Event structure.

GUI Control Specifications: UI structure.

Save to .mat Specification: .mat file name for saving structures

GUI Callback Function Definitions: functions to encode with text2cell utility.

It is strongly recommended that users follow this order in programming scripts, as error checking and structure validation are performed when the program is loaded, and some of these functions rely on the default order of structure definition. The philosophy is basically to define objects before referring to them in later structure definitions, although this rule is not strictly enforced.

Not all script objects need to be provided, and typically one needs to define only the objects required to run their sequence. For example, if one wants to just acquire RF data for offline processing, the Pixel Data, Display Window, Reconstruction and Processing objects can be omitted. Also, not all attributes of the various objects need be defined, as missing attributes are filled in automatically with default parameters. If a required attribute is missing, the software will generally exit with an error message.

2. Global System Objects (Non-Sequence Objects)

System objects are those objects that are defined independent of the sequence event programming, such as those that define characteristics of the transducer and system. These objects should be specified at the top of a Setup programming script, since they are often needed to define attributes of other objects contained in the Sequence. This also provides a more natural way of generating a user Setup script, allowing the building on previously defined structures.

Typically, the first system object attributes defined are variables needed to define specific characteristics of the Setup script. For example, one might want to define variables to indicate the starting and ending depth for ultrasound acquisition, with later structures using these variables instead of hard-coded values. This allows easy modification of the range of acquisition with the change of one or two variables. If a variable defined in this way should be saved as part of a Preset (a set of parameters that can be saved to a binary file for later restoration of the program state), the variable can be prefixed with 'P.', which makes it part of the preset structure, P, which is saved to a file when a preset is created.

Next, the Parameter attributes of the Resource object are typically defined. The Resource object is used to specify hardware and software resources needed for the execution of a user sequence. The Resource.Parameter attributes identify global factors that will be used by the user script and/or the system software.

In the definitions below, attributes that are required to be specified are highlighted. Typically, an error message will be generated by VSX if these attributes are missing or empty. Other attributes listed in plain text, for example, 'speedOfSound', will assume default values if not explicitly specified.

2.1 Resource.Parameters Attributes

```
Resource =
    Parameters =
        Connector (3.2+) double     connector # (1(dflt), 2, or [n,m,...])
        connector (3.0x) double     (deprecated) cnctr # (0(both),1(dflt),2)
        numTransmit      double     number of available transmitters
        numRcvChannels   double     number of available receive channels
        speedOfSound     double     speed of sound in meters/sec (deflt 1540)
        attenuation      double     media attenuation in -dB (deflt=0dB)
        speedCorrectionFactor double corrects speed of sound (default=1.0)
        startEvent       double     event no. to start after freeze (dflt=1)
        simulateMode     double     0 = 'use VDAS'(default), 1 = 'simulate'
        initializeOnly   double     0 = false (default), 1 = 'true'
        fakeScanhead     double     (dflt=0) if 1, allow to run without probe
        UpdateFunction   string     'VsUpdate' [default] or user specified
        GUI              string     'vsx_gui' [default] or '<user specified>'
        verbose          double     0:3 specifying level of warnings/errors
```

`Resource.Parameters.Connector` – This attribute was introduced in release 3.2 and replaces the `connector` attribute (lower case 'c') used in previous releases. The value is optional, and if not provided, defaults to 1. It is only needed for Universal Transducer Adapter modules with more than one connector, such as with the 256 T/R Vantage

system that includes a dual connector Transducer Adapter module (UTA 260-D). For this UTA, the values 1 and 2 select the left or right connector, respectively, and either connector can accommodate a 128 element (128 transmit/128 receive) transducer. The value of [1,2] can be used with a custom transducer that connects to both connectors, which can support up to 256 elements (256 transmit/256 receive). For UTAs with multiple connectors, the value of Trans.Connector is an array of the connector numbers to be activated. Software releases prior to 3.2 should use the `connector` attribute, which provides only three options - 0 (both connectors), 1 (left connector), or 2 (right).

`Resource.Parameters.numTransmit/numRcvChannels` – These values specify the number of transmit and receive channels available in the hardware through the selected connector(s). These need to be provided, and are checked when the Setup script is run with the Vantage hardware. If the actual hardware resources don't match with those specified, the sequence program may fail to execute, or execute with warning messages.

`Resource.Parameters.speedOfSound` – This value specifies the speed of sound in the media to be insonified. If it is not provided, the default value is 1540 m/s. Since many length parameters of sequence objects are specified in wavelengths, it is important that the speed of sound, if different from the default, be specified at the top of the script.

`Resource.Parameters.attenuation` – This is the attenuation of the media in dB/cm/MHz. A typical value is -0.5 dB/cm/MHz, but the default is 0 dB. If no value is set, but a `Media.attenuation` value is provided, that value is also used for the `Resource.Parameters.attenuation` value. The attenuation value is used by both the simulation and the image reconstruction software to compensate for the attenuation loss on transmit only. The attenuation loss on receive is compensated by the Time Gain Control curve (TGC).

`Resource.Parameters.speedCorrectionFactor` – This value is used to fine tune the speed of sound for media that contain slightly varying speeds of sound. It is typically only adjusted while a sequence is running, and initially the default value of 1.0 is used. This parameter only modifies the speed of sound used in an image reconstruction, and doesn't modify other length parameters.

`Resource.Parameters.startEvent` – This value specifies the starting event in a sequence. If it is not provided, the default start event will be event number 1. The startEvent is used for the initial run of the sequence, and whenever exiting the 'freeze' state. The startEvent can be changed dynamically during run-time using a GUI control and the Control.Command input to the runAcq processing (see method in section 4.0). This allows selecting different sequence event sections for execution during run-time.

`Resource.Parameters.simulateMode` – This parameter is used to turn on simulate mode, where the sequence is run by the software, and not by the hardware. This mode is entered automatically if no hardware is detected by the VSX loader program. Setting the value to 1 simulates acquisition and uses the Media model to describe the simulated media. The specified TX pulse from each pulser is simulated and the resulting returns from media targets are accumulated in the ReceiveBuffer. Setting the value to 2 turns off the media simulation, so that a previously acquired or loaded ReceiveBuffer can be continuously processed. For ReceiveBuffers with many frames, mode 2 effectively offers playback of captured RF cineloops.

`Resource.Parameters.initializeOnly` – This attribute, when set to 1, causes the software to exit after initialization. Initialization adds all the missing default attributes and validates structures. Additional hidden structures such as DMAControl for specifying transfers of data to host memory are also created. This is useful for debugging a script that crashes Matlab when run. Once initialized, one can examine all the structures for added attributes and correct sequencing.

`Resource.Parameters.fakeScanhead` – This optional parameter (default is 0) is set to 1 when it is desired to run a script using the Verasonics hardware when no scanhead is plugged into the system. This option is useful when one wants to probe signals on the I/O pins of the transducer connector.

`Resource.Parameters.updateFunction` – This optional parameter (default is 'VsUpdate') can be specified to invoke a different update function than the system default. The update function is used to update and add various attributes to structures at initialization, and during run-time changes. The specification of a different update functions should only be attempted by advanced users.

`Resource.Parameters.GUI` – This optional parameter (default is 'vsx_gui') can be specified to invoke a different Graphical User Interface function than the system default. The GUI function is used to draw the GUI window and default system controls. It also provides the means for users to create their own controls. The specification of a different GUI function can be used to create custom user interfaces for specific applications, but this task is challenging and should only be attempted by advanced users.

`Resource.Parameters.verbose` – The verbose attribute sets the level at which VSX and other functions report errors, warnings, and status messages on the Matlab command line. The supported values are:

0: Error messages only are displayed.
1: Error and warning messages are displayed.
2: Error, warning, and status messages are displayed. (default)
3. Error, warning, status, and debug status messages are displayed.

2.2 Transducer Object

The next system object that is typically defined in a new script is the Trans object, which describes the characteristics of the transducer that will be used. The type of transducer (linear, curved linear, phased array, etc.) generally dictates the shape of the scanning region and to some extent, the method of scanning. Distance units for the `Trans` structure can be specified in wavelengths of the center frequency (provided by `Trans.frequency`), or millimeter units, selected by means of the `Trans.units` attribute. The speed of sound given in the `Resource.Parameters.speedOfSound` attribute determines the conversion from mm to wavelengths. For example, a 5MHz center frequency transducer would have a wavelength of .308 mm at a speed of sound of 1540 m/sec.

The transducer characteristics can be set automatically for known transducer types, using the utility function, 'computeTrans.m'. First specify the name, then call the utility function. If one wants to use a center frequency other than the default, specify it as well before calling computeTrans.m. (For receive data acquisition, the system will automatically select a demodulation frequency and associated sample rate as close as possible to Trans.frequency, unless you specify a specific sample rate in the Receive structure. See section 3.3.1.1 for a list of supported 'demod' frequencies.) Finally, if the speed of sound in the media to be imaged is different from the default 1540 m/s, also make sure to set Resource.Parameters.speedOfSound to the correct value prior to calling computeTrans.

Refer to the "Example Scripts Matrix and Description" documents for a listing of probes (and associated example scripts) that are supported by computeTrans in the current Vantage software release. These documents are located in the "Biomedical" and "NDE_NDT" subfolders of the "Example_Scripts" folder in your Vantage software installation.

After specifying Trans.name and optionally Trans.units, a modified center frequency or speed of sound, the computeTrans.m utility function can be called as shown below to fill in the other attributes of the Trans structure.

```
Trans = computeTrans(Trans);
```

For unknown transducers, the user must generate their own Trans structure, and should provide at least the yellow highlighted attributes in the Trans structure below. The attributes highlighted in blue are read-only "dependent" fields that will be created by system software during script initialization and added to the Trans structure. Unless specifically stated otherwise, all numeric Trans structure fields are defined as Matlab doubles.

```
Trans =
  name                string    % Probe name ('L22-14v', 'C5-2v', etc.)
  units               string    % 'mm' or 'wavelengths'
  id                  double    % Probe id read from personality eprom
  frequency           double    % center frequency in megahertz
  Bandwidth        [1x2 double] % -6db lower and upper cutoff pts in MHz
  type                double    % 0=Lin(y=z=0),1=CrvdLn(y=0),2=2D,3=ann,4=R/C
  numelements         double    % number of transducer elements
  ElementPos       [nx5 double] % [position(wvlngths), az,el(radians)]
  elementWidth        double    % width in mm or wavelengths (spacing-kerf)
  elementLength       double    % only needed for row/col arrays
  spacingMm           double    % element spacing in mm
  spacing             double    % element spacing in wavelengths
  radiusMm            double    % curved array radius in mm (n/a for lin)
  radius              double    % curved array radius in wvl (n/a for lin)
 elevationApertureMm double    % (optional) 1D array elev. aperture in mm
  elevationFocusMm    double    % (optional) 1D array elev. focus depth in mm
  ElementSens    [1x101 double] % sens. curve for single element -pi/2to+pi/2
  lensCorrection      double    % 1 way delay in wavelengths thru lens
  elBias              double    % (optional) DC element bias supply voltage
  maxHighVoltage      double    % (optional) max. high voltage limit
  maxAvgPower         double    % (optional) max. total avg power (all chans)
  impedance           complex   % (optional) complex impedance vs frequency
  IR1wy            [nx1 double] % (optional) transducer 1 way impulse response
  IR2wy            [nx1 double] % (optional) transducer 2 way impulse response
  HVMux              structure  % for HV mux probes only (see section 2.2.2)
  ConectorES       [nx1 double] % EL to ES map (see text below & sec. 2.2.1)
  Connector        [nx1 double] % Overall mapping from elements to system
                                % channels (see text below and section 2.2.1)
  connType            double    % specifies type of Probe connector (see text)
```

Trans.name: This attribute specifies the name of the probe as a string variable of arbitrary length. For calling the computeTrans function, the Trans structure provided as an input argument must include Trans.name as a minimum since it is used to identify the probe for processing in computeTrans. For this to work properly, the spelling you use for Trans.name must match exactly the name string as listed within computeTrans. If you are specifying the Trans structure explicitly in your SetUp scripts and not using the computeTrans function, you can set Trans.name to any string value you like. The system will recognize the specific name of 'custom', however; this indicates that the probe ID value will not be checked by the system (allowing use of a probe that has no ID or an un-programmed or unknown ID). Note that an alternative approach to tell the system to ignore the probe ID is to set Trans.id to -1; this will allow you to assign a meaningful Trans.name for the probe instead of 'custom'.

Trans.Units: This is a string variable, with the only recognized values being 'mm' and 'wavelengths'. This attribute specifies the units that will be used for the `elementWidth`, `ElementPos`, and `lensCorrection` attributes of the Trans structure. For some Trans attributes, such as `Trans.spacing(Mm)`, both wavelength and mm units are always specified. If Trans.units is not specified, the system will assign a default value of 'mm'.

The use of wavelengths to specify the transducer dimensions when units are set to `'wavelengths'` can be somewhat confusing when imaging media with different speeds of sound, since obviously the probe dimensions must remain fixed. The

`Resource.Parameters.speedCorrectionFactor` can be used to change the wavelength value used in the media, but not for the probe. Define the probe dimensions with regard to the center frequency of the probe, and a nominal speed of sound, such as 1540 m/sec. If the media speed of sound in the media happens to be slower than the nominal, such as 1500 m/sec, then set the speedCorrectionFactor to 1540/1500 = 1.0267 (the time to a media target is longer by this factor). The wavelength calculations in the media will then be modified by this same factor (a slower speed at the same frequency results in a shorter wavelength), but the probe dimensions will remain unchanged.

It should be noted that for image reconstruction, the position of transducer elements is solely determined by the `Trans.ElementPos` array. Other attributes, such as `Trans.spacing(Mm)` and `radius(Mm)` can be defined for convenience in calculations, but are not used by the image reconstruction software.

Trans.id: This attribute specifies the Probe ID code as read by the system. While defined as a Matlab double, the only recognized values are integers in the range 0 to 16,777,216 (i.e. a 24 bit unsigned integer value). For probes that do not have an ID (and probe connector interfaces that do not provide an ID), set `Trans.id` to the special value of -1. This value notifies the system to not attempt to check the probe ID (or ignore it even if an ID value is available). Note also that even though `Trans.id` is often referred to or displayed as a hex string, it must be specified in the Trans structure as a Matlab double numeric value.

Trans.frequency: Center frequency of the transducer, in MHz. This value is used by the system to determine the scaling of all system variables using wavelength units.

Trans.Bandwidth: This is a 1 X 2 Matlab double array, specifying the lower and upper -6 dB round trip transducer bandwidth cutoff points in MHz. This attribute is optional and defaults to 60% of the `Trans.frequency` value. This attribute is used to set the Analog Front End (AFE) amplifier anti-aliasing filter cutoff - the default anti-aliasing cutoff will be the lowest available filter cutoff that is higher than `Trans.Bandwidth(2)`. The Bandwidth values are also used in the simulation of the transducer's transmit waveform from the pulser parameters specified. This simulated burst waveform is used when the Verasonics system is operating in simulate mode. Early Vantage software releases used a `Trans.bandwidth` format of a single bandwidth value in MHz that was assumed to be centered on `Trans.frequency`. For backward compatibility, if a script has specified `Trans.bandwidth` it will automatically be converted to `Trans.Bandwidth`.

Trans.type: This attribute is required to specify the geometry of the transducer. Linear arrays, where the center of the element positions can be specified with only x coordinates (y = z = 0), are type 0. (See Fig. 2.2.1 below) Curved arrays, including ring arrays, where the center of the elements must be specified with x and z coordinates (y = 0) and an azimuth angle (angle from the normal to the positive z axis), are type 1. For curved arrays, the origin is at the surface of the elements at the center position, and the curvature is in the x,z plane.

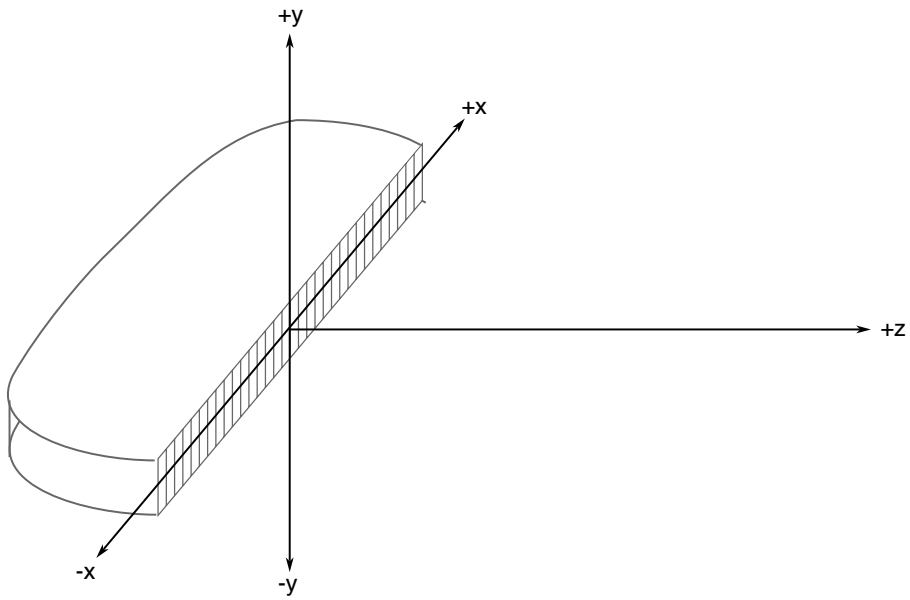


Fig. 2.2.1 Coordinate system for specifying linear array element positions.

Two dimensional arrays, where the center of the elements can be specified with x, y, z coordinates, are type 2. For 2D arrays, the origin is typically defined at the center of the 2D array, with the elements located in the x,y plane. However, some arrays are non-planar (bowl arrays), and elements can be defined at arbitrary 3D locations, although some locations are preferred for creating reconstruction regions. (See Figure 2.2.2 for the location of the origin and the coordinate axes for different scan formats.) For type 2 arrays, the Trans.radius attribute can be used to further describe a bowl shaped array, but the Trans.ElementPos array must be defined as it will be used by the reconstruction software; the az and el (azimuth and elevation) angles of the normal to the element surface are used to orient the element sensitivity (directivity) function.

Starting with software version 3.4.1, `Trans.type = 3` can be used to specify an annular array transducer with ring shaped elements centered around the z axis. In this case, the 5 elements of the Trans.ElementPos array take on new definitions, as follows:

```
        Trans.ElementPos(N, :) = [ Ri Ro Rc Zc Azimuth ]
Where
   Ri = radius from the Z-axis to inner edge of the element
   Ro = radius from the Z-axis to outer edge of the element
   Rc = radius from the Z-axis to the equal-area 'center' of the element
   Zc = Z coordinate of the equal-area 'center' of the element
   Azimuth = angular orientation of element with respect to the Z-axis.
```

For type 3 arrays, the `Trans.ElementSens` array is defined similarly to types 0, 1 and 2 as a two dimensional function around an element's normal, but is not axial rotatable. Instead the function is defined in a plane that contains the Z axis and is normal to the ring described by an element's equal area 'center'. The function is centered around the element's normal, as specified by the Azimuth value in Trans.ElementPos. Since annular rings for bowl shaped arrays may have fairly large dimensions for the ring's

'width', the Trans.ElementSens function may be quite narrow, and fall off rapidly. An accurate ElementSens function will allow simulation to determine the effective range over which the focal point can be moved. Note: With software version 4.2.0, type 3 arrays are not yet supported for image reconstruction or RF data simulation.

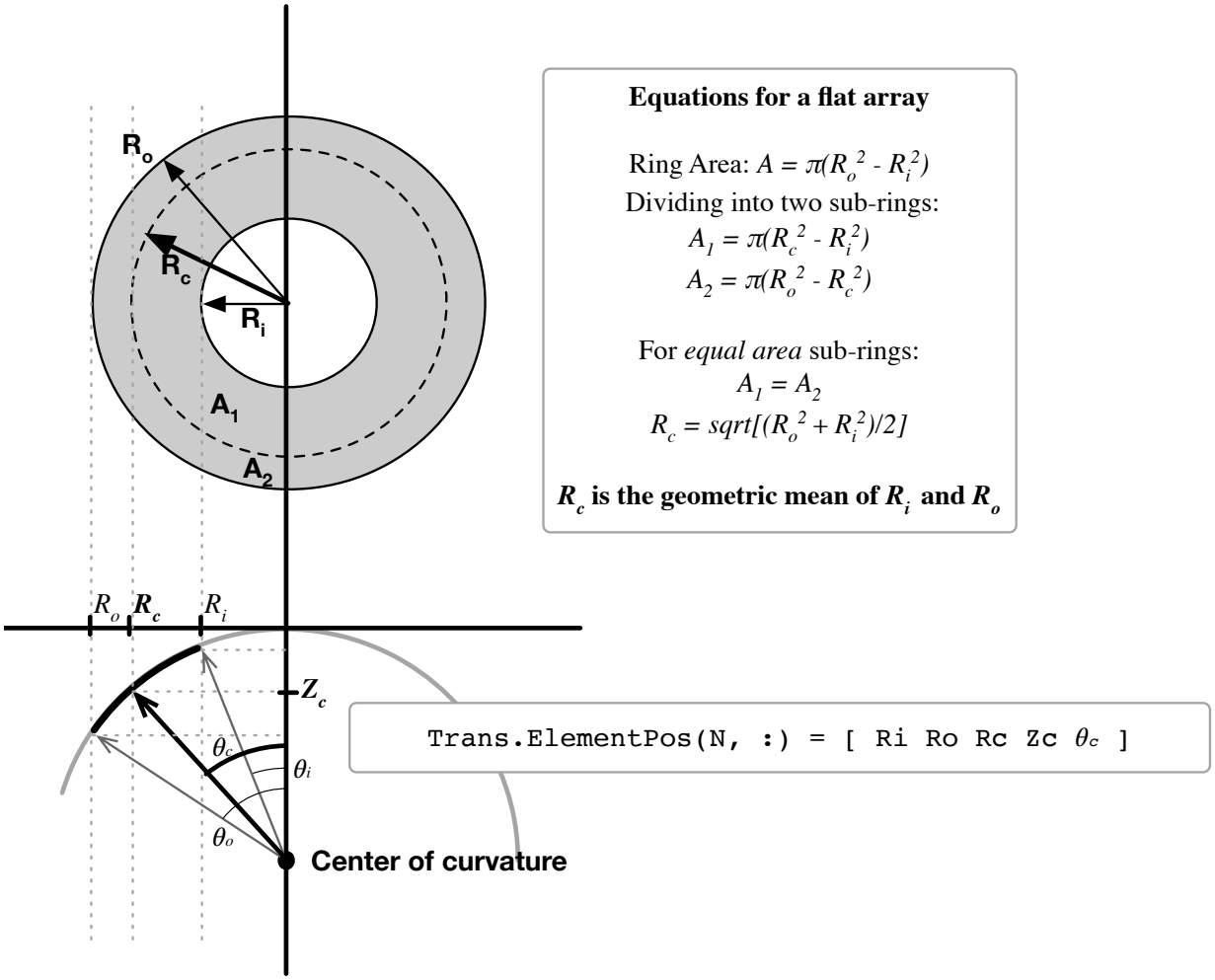


Fig. 2.2.2 Parameters for specifying annular array element positions.

With software version 4.2, `Trans.type = 4` can be used to specify a row/column array type. Row/column arrays are essentially two linear arrays that are overlayed, with one array at 90 degrees from the other. The `Trans.ElementPos` array should specify the location of the center of the bar elements, which are located along the x and y axes. The z coordinate of all elements should be zero. The length of the bar elements should be specified in the `Trans.elementLength` attribute. In software version 4.2 and above, row/column arrays are fully supported in reconstruction.

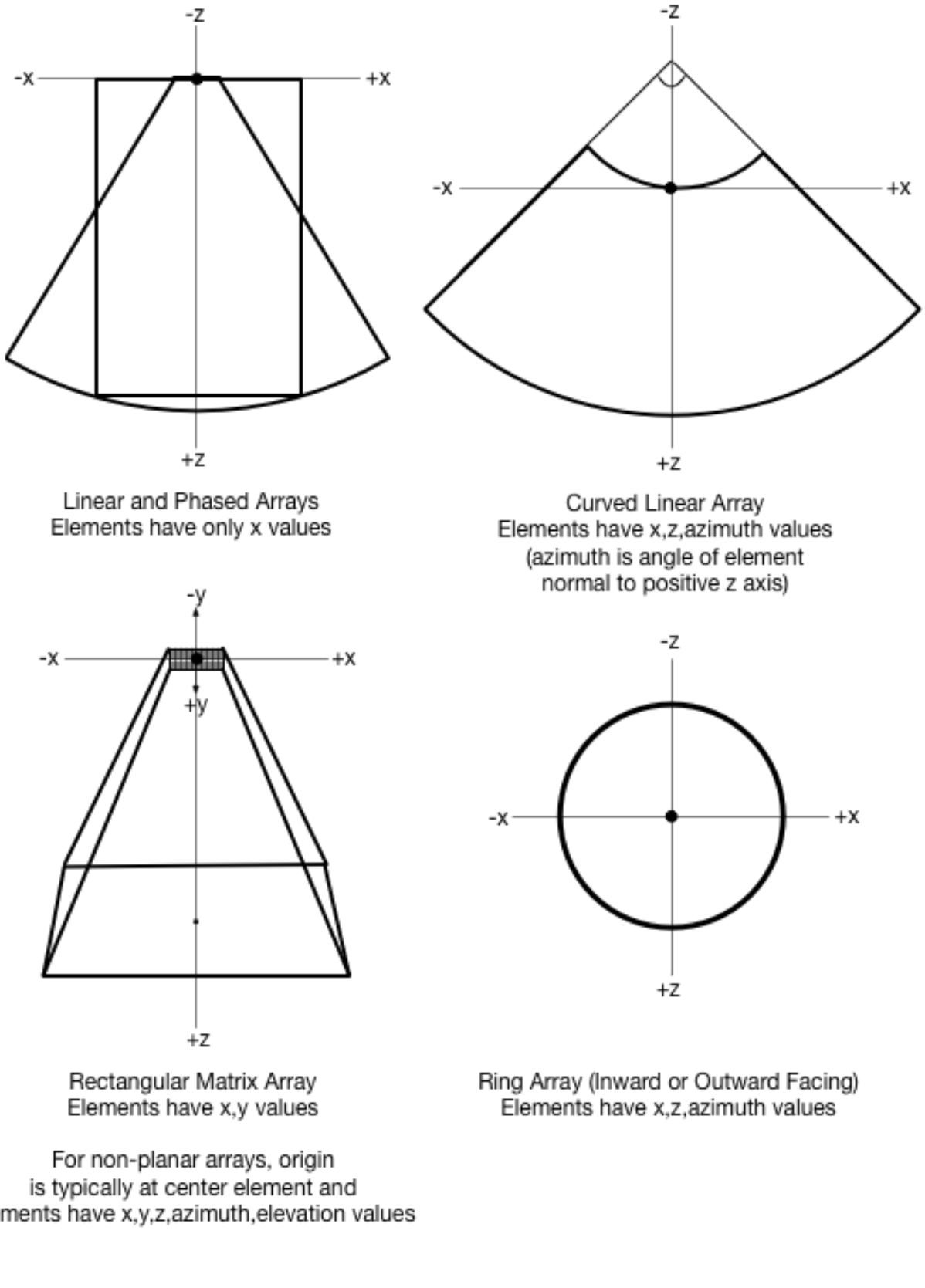


Fig. 2.2.2 Origin and coordinate system for various array types

Trans.numelements: This attribute simply specifies the total number of elements in the transducer and thus the number of rows in the `Trans.ElementPos` array. This number, along with the `Trans.spacingMm` (for millimeters) or `Trans.spacing` (for wavelengths) sets the total aperture of a linear array transducer. The `Trans.elementWidth` attribute defines the actual width of a transducer element, which is generally smaller than the spacing by the kerf of the cut separating elements. The element width is used in the default calculation of an element's directional sensitivity function.

Trans.ElementPos: The position and orientation of each element in the transducer is defined in the `Trans.ElementPos` array. The `Trans.ElementPos` array has a row index corresponding to the element number and up to five column values specifying the x, y, and z coordinates (in mm or wavelengths) of the center of the element, and the two angle parameters, azimuth and elevation. The azimuth angle is the angle between the normal of the element and the positive z axis and is needed when specifying element positions for curved arrays. The elevation angle is the angle of the element normal to the x,z plane, and is only needed for specifying element positions for certain 2D arrays. The azimuth and elevation angles need not be specified if not required and/or if zero for all elements. See Fig. 2.2.3 below for determining azimuth and elevation angles for an element. Note that for the annular array transducer type, a different meaning is assigned to the five values in `Trans.ElementPos`. See the `Trans.type` section above.

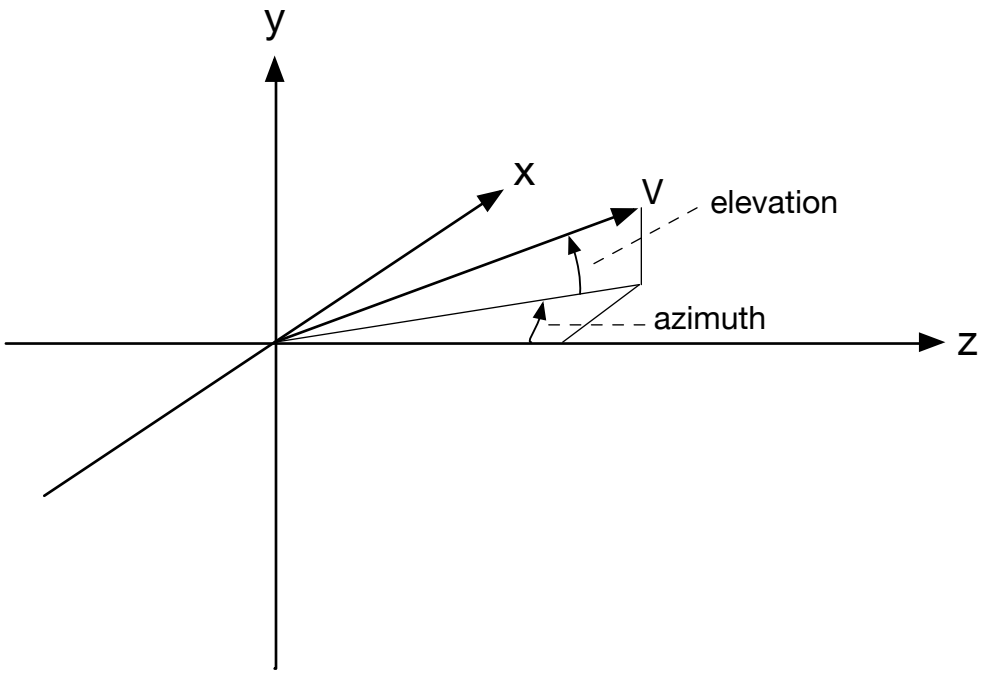


Fig. 2.2.3 Azimuth and elevation angles. These angles are specified as if the origin of the x,y,z axes was at the position of the element center.

Trans.spacing, Trans.spacingMm: This attribute specifies the center-to-center spacing between elements of the transducer. The `Trans.numelements` attribute, along with `Trans.spacingMm` (for millimeters) or `Trans.spacing` (for wavelengths) sets the total aperture of a linear array transducer. Note that the element spacing can be derived

directly from `Trans.ElementPos;` the `Trans.spacing` attributes are only provided for convenience in calculations related to the element geometry such as in defining the PData structure.

Trans.elementWidth: This attribute defines the actual width of a transducer element, which is generally smaller than the spacing by the kerf of the cut separating elements. The element width is used in the default calculation of an element's directional sensitivity function.

Trans.radius, Trans.radiusMm: This attribute applies only to curved array transducers (`Trans.type` value of 1), to specify the radius of curvature for the curved array. As with `Trans.spacing`, the `Trans.radius` values could be derived from `Trans.ElementPos` and are only provided as a convenience.

Trans.elevationApertureMm, Trans.elevationFocusMm: For 1D linear or curved array transducers, these two optional fields can be used to specify the elevation aperture (i.e. the width of the transducer elements in the Y dimension) and the elevation focus depth set by a lens on the face of the transducer. These attributes can be used to estimate the overall beam response of the transducer in the elevation dimension, outside the X-Z imaging plane, though they are not used in the Vantage software for 4.2.0 and are provided for information only. Note these values are always in mm units and are not affected by `Trans.units`.

Trans.ElementSens: The directivity pattern of an individual transducer element is specified in the `Trans.ElementSens` array. This array provides the relative sensitivity of the element to targets at various angles from the normal to the element. The `Trans.ElementSens` array contains 101 sensitivity values at equally spaced angles from -pi/2 to +pi/2. The value at angle 0 is normalized to 1.0. The default setting for the `Trans.ElementSens` array uses the Matlab formula

```
X = Trans.elementWidth*pi*sin(Theta);
Trans.ElementSens = abs(cos(Theta).*(sin(X)./X));
```

where

```
Theta = (-pi/2:pi/100:pi/2);
```

with the value of Theta at 0 given a small value to avoid a divide-by-zero. This formula serves reasonably well for a linear array element oscillating in bar mode. For other element geometries, one can compute or empirically enter the `Trans.ElementSens` values, as long as the number of values remains at 101. For 2D arrays, the sensitivity function for an element is assumed to be axially symmetric about the element normal. The current use of the sensitivity values is during image reconstruction, where it is possible to eliminate an element's contribution if its sensitivity to the pixel location is below a specified threshold.

Trans.lensCorrection: The `Trans.lensCorrection` parameter is used during image reconstruction to compensate for the travel time through the transducer lens. Lens are typically made from a material that has a slower speed of sound than the medium, so that they can assume a convex curvature. The `Trans.lensCorrection` parameter should be set to the one way value in wavelengths that the lens adds to the acoustic path that would be taken without the lens. For example, assume the speed of sound in the lens is 3/4 the speed of sound in the medium, and that the average thickness of the

lens is approximately 12 wavelengths (with the 12 wavelengths representing wavelengths based on the speed of sound in the medium). In this case, the lens correction parameter would be (12 * 4/3) - 12, or 4 wavelengths, which is the effective increase in path length from the sound traveling one way through the lens. The default value of this attribute is zero.

Trans.elBias: For probes that apply a DC bias voltage to the transducer elements using a bias voltage supply provided by the system, this optional attribute is used to enable the bias voltage supply output and specify the desired voltage in Volts. If `Trans.elBias` is not specified, the system will create it and assign a default value of zero indicating the system's bias voltage supply feature is not being used.

Trans.maxHighVoltage: Optionally, using the attribute `Trans.maxHighVoltage`, one can specify a high voltage limit for the transmitter for this transducer, up to the limit set in the `computeTrans` utility function. The peak-to-peak transmit voltage will be twice this value, since the transmitter has a bi-polar drive waveform. Specifying a limit is a good idea, since the maximum high voltage limit set by default (typically 50 volts) may allow excessive acoustic power to be delivered to the transducer, resulting in excessive heating of the probe and media. For newer Vantage systems, the maximum high voltage limit can be set higher than the default of 50 volts, up to the hardware limit of the Transmit Power Controller, which is currently 96 volts; however, if the transducer has a power limit set in computeTrans for preventing damage to the transducer, this limit will be enforced. (CAUTION: The maximum hardware high voltage limit is capable of damaging some transducers and/or producing harmful acoustic power levels in biological tissue. To prevent damage, do not modify the high voltage limits in computeTrans. In scanning, use high voltage levels only as high as needed for a specific application.)

Trans.maxAvgPower: Optionally, the attribute `Trans.maxAvgPower (Watts)` can be used to limit the total average power into the transducer (summed over all active channels) during the acoustic sequence. The average power is a function of many programmed parameters, including the transmit voltage, the burst length, the PRF, and the transducer impedance. This attribute can be used to protect a transducer from overheating, or to limit the average power delivered to the acoustic media (the value must be derated by the transducer's efficiency). **Note:** If this attribute is not defined by the user in the Setup script, no limit checking for this parameter will be performed.

Trans.impedance: The `Trans.impedance` attribute is an optional specification (default value is 20 ohms) that is used by the software of the Verasonics system to estimate the high voltage and power requirements for various transmit sequences, using a utility function (TXEventCheck.m) that can analyze a user's sequence script. This value can be specified for unknown probes in different ways. If only the real component is known for the center frequency, the value can be specified in ohms. If the complex impedance at the center frequency is known, it can be specified as a single complex value, eg. `31.2-2.05i`. If the impedance is known at various frequencies, Trans.impedance can be specified as an N x 2 complex array where the first column specifies frequencies in MHz and the second column is complex impedance in Ohms at those frequencies, eg. `[4.5 32.8-72.6i; 4.75 39.2-66.2i; 5 46.1-69.6i; 5.25 46.5-72.4i; 5.5 41.9-71.6i]`. Any number of frequencies (value of N) can be specified at any arbitrary frequency spacing but they must be listed in increasing order. For extended transmit and especially for HIFU applications, specifying Trans.impedance accurately is

critical to getting appropriate estimates of the transmitter operating state from TXEventCheck, since the default value for impedance is quite low.

Trans.IR1wy, Trans.IR2wy: These attributes are used to define the one-way and two-way impulse response of the probe and transmitter circuit. Each array is a vector of Matlab doubles representing the impulse response, sampled at 250 MHz. These arrays are used to define the simulation transmit waveform (used when running the system in simulate mode) and also by the computeTXPD function to evaluate transmit beam response, and by the Arbwave Toolbox when synthesizing a tristate transmit waveform with compensation for the probe impulse response. These attributes are optional; if they are not specified, VSX initialization will create them and add them to the Trans structure, using a default impulse response based on the values of Trans.Bandwidth (as a second-order bandpass filter defined to match the cutoff frequencies from Trans.Bandwidth).

Trans.ConnectorES is a required attribute used to specify the connections from individual transducer elements to Element Signal pins at the transducer connector. This attribute did not exist in Vantage software releases prior to 4.0; those releases used `Trans.Connector` instead. In all Vantage software releases, `Trans.Connector` is used to define the overall mapping from transducer elements to system channels. The difference is that in releases prior to 4.0, `Trans.Connector` had to be specified by the user's SetUp script, while in current releases it is created by the VSX initialization software and added to the Trans structure as a "read-only" field. (See section 2.2.1 below for more information on the overall mapping from elements through the probe connector pins and then to system hardware channels, using these two arrays.)

The `Trans.ConnectorES` array size is one column by `Trans.numelements` rows, with the row index corresponding to the element number. The value for a given row index is the Element Signal (connector pin index) to which the element is connected. If not specified, a default `Trans.ConnectorES` assignment will be a created that assumes the transducer elements are assigned to Element Signals at the connector on a 1-to-1 basis, that is, element 1 is connected to Element Signal 1, etc. (Element numbers must correspond with the index of the Trans.ElementPos array specifying their physical location.) For a custom probe, one might want a different assignment of element numbers to connector Element Signals, and in this case, the `Trans.ConnectorES` array can be used to set the mapping. For example, for a 128 element probe where the elements are connected to Element Signals in reverse order, the `Trans.ConnectorES` array would be defined in Matlab as `Trans.ConnectorES = (128:-1:1)'`. (note the tick at the end to convert to a column array.)

If a transducer (non-multiplexed) uses fewer connector Element Signals than are available, then a `Trans.ConnectorES` array must be defined, specifying the Element Signals to be used. For example, a 64 element probe connected to the connector Element Signals 33 to 96 should set `Trans.ConnectorES = (33:96)'`.

The `Trans.ConnectorES` array must also be defined for probes with HVMux switching; in this case it will identify all available paths through the HVMux switches to Element Signals at the connector. For example, a 256-element linear array mapped through HVMux switches to a 128-channel connector might set

```
Trans.ConnectorES = [1:128, 1:128]';.
```

This would indicate that either element 1 or element 129 can be switched to Element Signal 1; element 2 or element 130 can be switched to Element Signal 2, etc. Note that `Trans.Connector` was not used for HVMux probes in software releases prior to 4.0, but in current software releases since 4.0 Trans.ConnectorES and Trans.Connector are used for all probes including those with HVMux switching.

Refer to section 2.2.1 below for more information on the mapping from elements to system channels, and to section 2.2.2 for use of the Connector attributes with HVMux probes.

Trans.connType: In early Vantage SW releases the system only supported one kind of probe connector, so nothing was needed in a user script to identify it. Newer Vantage systems utilize a user removable transducer connector module, referred to as a UTA (Universal Transducer Adapter). With different UTA modules, additional connector options are available, which have different element mappings than the original 260 pin, 128 channel connectors. From the 2.11 software release, utilities are now available in the Tools/ElementToChannelMapping directory to show the element and channel mapping for the different connectors.

The field `Trans.connType` identifies the connector type required by the user script. Trans.connType is now required since the 2.11 system SW; it is an integer value used as an index to identify one of the connector types supported by the Universal Transducer Adapter system. The following connector type index values are recognized:

- Trans.connType = 1 (default if not provided) identifies the HDI-format connector (Cannon DL-260 connector with 128 element signals), identical to the probe interface used on pre-UTA Vantage systems. For backward compatibility to scripts developed with earlier SW releases, VSX will create Trans.connType with a default value of 1 if it was not defined in the user SetUp script (but only if the value of Resource.Parameters.numTransmit matches the requirements of the HDI connector, i.e. either 128, or 256 with `Resource.Parameters.connector = [1 2];` to select both connectors.
- Trans.connType = 2 identifies the interface from either a “break-out board” UTA adapter (providing direct access to the element signals, with no specific probe connector at all), or a custom UTA adapter built by a user.
- Trans.connType = 3 identifies the MS family connector interface and pinout (Cannon DL-360 connector with 256 element signals).
- Trans.connType = 4 specifies the Verasonics UTA-408 connector interface.
- Trans.connType = 6 or 12 specifies the “NDT” Hypertronix 160 pin, 128 channel connector interface used on the UTA 160-DH/32 Lemo and UTA 160-SH/8 Lemo adapter modules.
- Trans.connType = 7 specifies the GE D-series probe connector interface, using a 408 pin, 256 channel connector. This connector is used on the UTA 408-GE adapter module.
- Trans.connType = 8 specifies the probe connector interface used on the UTA 1024-MUX and UTA 256-Direct adapter modules. This is a “direct connect” interface where the probe cable is terminated in small connector circuit boards that are semi-permanently connected directly to the UTA adapter module.

- Trans.connType = 9 specifies the 156 pin, 128 channel connector interface used by Ultrasonics probes. This connector interface is used on the UTA 156-U adapter module.
- Trans.connType = 10 specifies the Ipex 160 pin, 128 channel connector interface used on the UTA 160-SI/8 Lemo adapter module.
- Trans.connType = 11 specifies an array of Lemo single-element coax connectors, used on the UTA-64 Lemo and UTA-128 Lemo adapter modules.

There are two other 'special' Trans.connType values that can be defined in a user's SetUp script:

- Trans.connType = 0 identifies a simulation-only script that is not intended to be used on a HW system at all, and thus there is no need to identify a probe connector interface. When this value is present in the script, none of the system HW compatibility constraints will be applied (for example, the number of transmit and receive channels can be set to any desired value in the range from 1 to 1024). Setting Trans.connType to 0 has the same effect as setting Resource.Parameters.simulateMode to 1 (if a user script has set Trans.connType to 0 and Resource.Parameters.simulateMode is undefined, VSX will set a default value of 1 for simulateMode). This new field also provides additional flexibility while using simulate-only scripts: it will allow use of RF cineloop playback of the contents of a predefined receive data buffer for a simulate-only script by setting simulateMode to 2. (In earlier Vantage SW releases, setting simulateMode to 2 caused the system to initialize the script for use with the HW system, and as a result would prevent a simulate-only script from running if it was incompatible with the HW system constraints.) VSX will assign a default Trans.connType value of zero for any script that has not defined it but has set Resource.Parameters.numTransmit to a value incompatible with the HDI (260-pin) connector; the result will be that the script will only run in simulation mode. To use a UTA adapter with any connector type other than the original HDI, the SetUp script must explicitly set the connType value.
- Trans.connType = -1 is for use in test scripts that are intended to be compatible with multiple UTA adapters. When VSX initializes a script with this value for use on a HW system, the -1 will be replaced with the index value representing the connector type that is actually present on the UTA adapter attached to the system. This allows test scripts or other utilities to be written in a 'generic' format that will automatically adapt to the actual system HW configuration when they are run.

2.2.1 Defining the Mapping from Transducer Elements to System Channels

Most of the arrays used to program the functionality of the Vantage system on a per-element basis (such as `Receive.Apod`, `TX.Apod`, and `TX.Delay`) are defined with a direct mapping to individual transducer elements as defined in the Trans structure. However, to program the actual transmit and receive channels of the Vantage hardware system the software must know which hardware system channel is connected to each transducer element. That relationship is identified here as "element to channel

mapping". To understand this mapping it is important to recognize that the overall system is made up of two independent units:

- The transducer assembly, which includes the transducer itself plus wiring to connect each individual transducer element through the cable to a pin at the probe connector, and
- The hardware system, which connects individual element signals from pins at the probe connector to system channels, through the wiring within the system and the particular UTA module that is being used.

In Vantage software releases prior to 4.0, this overall mapping was defined in the single array `Trans.Connector`. With the growing number of UTA modules and system configurations available, the tasks of creating, managing, and understanding that mapping have grown increasingly complex. To make those tasks simpler, Vantage software releases starting with 4.0 provide separate, explicit definitions of the mapping within the probe assembly and the mapping within the UTA and hardware system. While the process of mapping individual transducer elements to their associated hardware system channels is conceptually simple, it can become very confusing if one is not precise and consistent with array and signal naming conventions. In that spirit, the mapping arrays and signal naming definitions will be described first, followed by a more detailed discussion of how the mapping functionality is defined and used.

Signal Naming: As the signal from a transducer element propagates through the probe wiring and system wiring to reach the associated hardware system channel, it will be associated with three different naming conventions as defined here:

- **Elements** (often abbreviated as "**EL**"): The numbering of transducer elements as defined in the Trans structure through the fields `Trans.numelements` and `Trans.elementPos`. Note that element indexing always starts at one and goes to `Trans.numelements`.
- **Element Signals** (often abbreviated as "**ES**"): The signal names assigned to pins at the probe connector. Verasonics UTA modules that have been designed to support a commercially available family of probes will use the Element Signal numbering convention defined by the manufacturer of that probe family. Note however that some manufacturers use Element Signal numbers starting from zero. In these cases, the Verasonics documentation will add one to all element signal numbers and this will be noted in the documentation for that connector interface. The Element Signal numbering *always starts* at 1 and proceeds up to the total number of Element Signals supported by the connector interface. Note that some probes may have fewer elements than there are Element Signals at the connector, and thus some of the Element Signals will not be used. Conversely, HVMux-based probes typically have more elements than there are Element Signals and the HVMux switches will provide a path from each element to one of the available Element Signals.
- **System Channels** (often abbreviated as "**CH**"): The signal name and numbering for the channels of the Vantage hardware system that are actually being used with a particular UTA module and probe. These numbers always start at one and proceed up to the total number of active channels. There is always a one-to-one mapping between the System Channel numbers and the columns of the Receive Data buffers as defined in the Matlab workspace. The number of System Channels in use will be

identified by the field `Resource.Parameters.numTransmit`, after the system has finished initialization for use with a particular probe, UTA module, and SetUp script. Note that because of constraints in the Vantage system hardware design, there may be cases where the total number of active system channels is greater than the number of active Element Signals. In this case, the unused channels will still be present in the Receive Data buffer, but the data will be all zeros. (This situation may occur because the system channels can only be enabled in contiguous blocks of 32. For example, a probe with 24 elements would probably be mapped to 32 system channels (and 32 columns in the receive data buffers) with 8 of those channels unused.)

Signal Mapping Arrays: The following three arrays are used by the system to define the mapping between Elements, Element Signals, and System Channels:

- **Trans.ConnectorES**: This array defines the mapping from transducer elements to the Element Signals as identified at the probe (transducer) connector. It is defined as a column vector in the Matlab workspace, with length equal to `Trans.numelements`. The Nth entry in the array defines the mapping for element number N, and the value in that entry identifies the Element Signal number to which that element is connected at the probe connector. Note that `Trans.ConnectorES` defines the signal mapping within the probe assembly itself and is independent of any mapping that may occur within the Vantage system on the other side of the connector. `Trans.ConnectorES` is a required field that must always be present in the Trans structure. However, for many probes with an element count equal to the number of signals at the probe connector there will be a direct one-to-one mapping from Elements to Element Signals (see the L11-5v, for example). In this situation you do not need to define the `Trans.ConnectorES` array, and the system will automatically create it with a default one-to-one mapping.
- **UTA.TransConnector**: This array defines the mapping from Element Signals at the probe connector to system hardware channels (and columns of the Receive Data buffer). It is created automatically during system initialization by the computeUTA helper function; the actual mapping depends on the Vantage system configuration and UTA module being used, which connector(s) have been selected, and the needs of the specific probe and user script being used. `UTA.TransConnector` is a column vector in the Matlab workspace, with an entry for each Element Signal, and the value at that entry is the system channel number to which that Element Signal is connected. Unused Element Signals will have an entry of zero, indicating that they are inactive and have not been associated with a system channel.
- **Trans.Connector**: This array defines the composite overall mapping from transducer elements to system channels. As with `Trans.ConnectorES`, it is defined as a column vector in the Matlab workspace, with length equal to `Trans.numelements`. The Nth entry in the array defines the mapping for element number N, and the value in that entry identifies the system channel number to which that element is connected in the hardware system (and the associated column of the Receive Data buffer). To create the overall mapping from transducer elements to system channels, VSX initialization will combine `Trans.ConnectorES` with `UTA.TransConnector` to create `Trans.Connector` as follows:

```
Trans.Connector = UTA.TransConnector(Trans.ConnectorES);
```

Note that `Trans.Connector` is a dependent, read-only field in the Trans structure, created by the system software during VSX initialization. If the SetUp script has defined a `Trans.Connector` array, it will be ignored and overwritten with the values created by the system. (An exception is made for backward compatibility with SetUp scripts written for software releases prior to 4.0; see the "Backward Compatibility" heading below.)

Backward Compatibility, for user scripts written for Vantage software releases older than 4.0: The `Trans.ConnectorES` field did not exist in these older releases, but `Trans.Connector` did exist and had the same meaning as defined above: it defines the overall mapping from transducer elements to system channels. The difference with the older releases is that the user SetUp script was required to create the overall mapping, while in releases since 4.0 the user script only defines the element-to-connector mapping (in `Trans.ConnectorES`) and the system software will create the composite mapping in `Trans.Connector`. To preserve backward compatibility as much as is possible, the Vantage software in releases starting with 4.0 will first look for `Trans.ConnectorES` from the user script, but if it is not found and `Trans.Connector` does exist, the software will assume this is an older script with `Trans.Connector` already providing the composite mapping. In this case `Trans.ConnectorES` will not be created at all, and `Trans.Connector` will be used as-is. Note however that this attempt at backward compatibility will fail if the UTA module or connector selection being used differs from the configuration used in the original script.

When considering whether a pre-4.0 user script will be compatible with the current software, another important factor to consider is the connector type being used. On older Vantage hardware systems built before the UTA feature was available, there was only one connector type available: the 260 pin, 128 channel connector interface designed for compatibility with ATL-Philips "HDI format" probes using the Cannon DL-260 connector. For UTA systems, this same connector interface is provided through the UTA 260-S and UTA 260-D modules. For any of those configurations while using a single connector, the mapping from Element Signals to system channels (in `UTA.TransConnector`) is one-to-one and as a result, `Trans.ConnectorES` and `Trans.Connector` will be identical. In this situation, full backward compatibility is ensured. However, for the majority of other UTA modules that have been added since the UTA 260-S or D, the `UTA.TransConnector` mapping is not one-to-one and is likely to change depending on connector selection and system configuration. In these situations it may be necessary to redefine the script to create `Trans.ConnectorES`, so the system can then create `Trans.Connector` automatically. One final note on backward compatibility: if you are using a script that calls computeTrans to create the Trans structure, then backward compatibility with regard to `Trans.Connector` will be taken care of automatically, since computeTrans in current releases automatically creates `Trans.ConnectorES` instead of `Trans.Connector`.

Note that backward compatibility from one Vantage software release to another applies to the SetUp scripts themselves, not the .mat files they create. When migrating from one Vantage software release to another you should never attempt to carry .mat files between releases. Instead, move your SetUp scripts to the new release and run them again to create .mat files specific to that release.

2.2.2 Transducer Connections for the Canon Connector UTAs

The connections of the probe elements that use the Canon 260 pin ZIF connector to the system transmitters and receivers is dependent on the Verasonics system configuration. The Canon ZIF connector is part no. DL5-260PW6A and probe elements are connected to the appropriate IO pins on the connector, whose names and pin designations are shown in Table 2.2.1.1 below. For a Vantage 32LE system with the UTA-260-MUX Cannon connector UTA, the connection of IO channels to transmitters and receivers is shown in Fig. 2.2.1.1 below. The connections for the Vantage 64 system with the 260-MUX UTA are shown in Fig. 2.2.1.2. The connections for a Vantage LE 64 channel and a Vantage 128 channel system (with a single Canon connector UTA), are shown in Fig. 2.2.1.3 and 2.2.1.4 respectively below.

For a 256 channel Vantage system with the Cannon connector UTA, there are 256 transmit and 256 receive channels, connected to two scanhead connectors as shown in Fig. 2.2.1.5 below. A 128 element transducer can be connected to either connector, and the SetUp script can treat the system as if it had 128 transmit and 128 receive channels, specifying in the `Resource.Parameters.connector` attribute which connector to use. The loader program, VSX, will then determine which set of 128 transmitters and receivers to use to select the proper connector, and the transmit and receive channels can be specified as if only a single connector was available. A custom probe with up to 256 elements can use both connectors at once, by specifying in the SetUp script a connector number 0. In this case all transmit and receive channels can be used, and the second connector extends the transmit and receive channel count from 129 to 256.

A custom probe with no probe ID EPROM should set the `Trans.name` attribute to ‘custom’. This will cause VSX to skip the probe ID check for a connected transducer, thus allowing the script to run. The Vantage system monitors whether a scanhead is connected, and for this monitoring to work properly, pin A1 on the custom probe connector must be grounded.

Return to TOC

Pin	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10
Name	AGND	AGND	PDAT	PCLK	PSIO	PWEN	PPWR	AGND	AGND	AGND
Pin	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10
Name	AGND	n/c	n/c	n/c	n/c	n/c	n/c	n/c	n/c	AGND
Pin	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
Name	AGND	n/c	n/c	n/c	n/c	n/c	n/c	n/c	n/c	AGND
Pin	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10
Name	AGND	muxD0	muxD1	muxD2	muxD3	muxD4	muxD5	muxD6	muxD7	AGND
Pin	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
Name	AGND	muxHvP	muxHvN	muxSw	n/c	muxDLa	MuxDCl	n/c	AGND	AGND
Pin	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
Name	AGND	n/c	n/c	n/c	n/c	n/c	n/c	n/c	n/c	AGND
Pin	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10
Name	AGND	n/c	n/c	n/c	n/c	n/c	n/c	n/c	n/c	AGND
Pin	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
Name	AGND	n/c	n/c	n/c	n/c	n/c	n/c	n/c	n/c	AGND
Pin	J1	J2	J3	J4	J5	J6	J7	J8	J9	J10
Name	AGND	n/c	n/c	n/c	n/c	n/c	n/c	n/c	n/c	AGND
Pin	K1	K2	K3	K4	K5	K6	K7	K8	K9	K10
Name	AGND	IO93	IO94	IO95	IO96	IO97	IO98	IO99	IO100	AGND
Pin	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10
Name	AGND	IO92	IO91	IO90	IO89	IO104	IO103	IO102	IO101	AGND
Pin	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
Name	AGND	IO85	IO86	IO87	IO88	IO105	IO106	IO107	IO108	AGND
Pin	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10
Name	AGND	IO84	IO83	IO82	IO81	IO112	IO111	IO110	IO109	AGND
Pin	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
Name	AGND	IO77	IO78	IO79	IO80	IO113	IO114	IO115	IO116	AGND
Pin	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
Name	AGND	IO76	IO75	IO74	IO73	IO120	IO119	IO118	IO117	AGND
Pin	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
Name	AGND	IO69	IO70	IO71	IO72	IO121	IO122	IO123	IO124	AGND
Pin	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
Name	AGND	IO68	IO67	IO66	IO65	IO128	IO127	IO126	IO125	AGND
Pin	U1	U2	U3	U4	U5	U6	U7	U8	U9	U10
Name	AGND	IO4	IO3	IO2	IO1	IO64	IO63	IO62	IO61	AGND
Pin	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
Name	AGND	IO5	IO6	IO7	IO8	IO57	IO58	IO59	IO60	AGND
Pin	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10
Name	AGND	IO12	IO11	IO10	IO9	IO56	IO55	IO54	IO53	AGND
Pin	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10
Name	AGND	IO13	IO14	IO15	IO16	IO49	IO50	IO51	IO52	AGND
Pin	Y1	Y2	Y3	Y4	Y5	Y6	Y7	Y8	Y9	Y10
Name	AGND	IO20	IO19	IO18	IO17	IO48	IO47	IO46	IO45	AGND
Pin	Z1	Z2	Z3	Z4	Z5	Z6	Z7	Z8	Z9	Z10
Name	AGND	IO21	IO22	IO23	IO24	IO41	IO42	IO43	IO44	AGND
Pin	AA1	AA2	AA3	AA4	AA5	AA6	AA7	AA8	AA9	AA10
Name	AGND	IO28	IO27	IO26	IO25	IO40	IO39	IO38	IO37	AGND
Pin	BB1	BB2	BB3	BB4	BB5	BB6	BB7	BB8	BB9	BB10
Name	AGND	IO29	IO30	IO31	IO32	IO33	IO34	IO35	IO36	AGND
Pin	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
Name	AGND	AGND	REF+	SNSS1	SNS2	SNS3	SNS4	AGND	AGND	AGND

Table 2.2.1.1 Canon transducer connector I/O channel to pin number mapping.

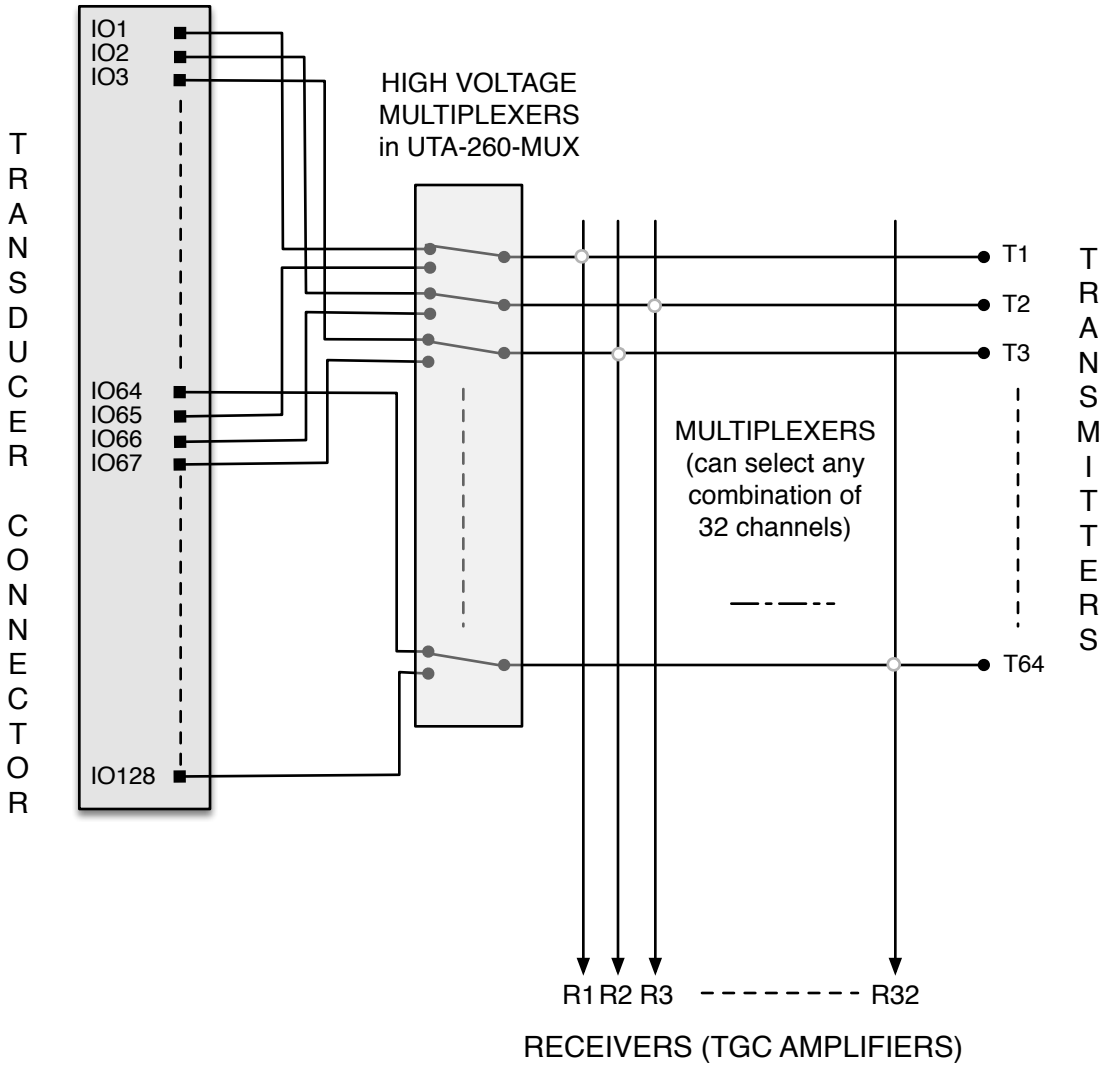


Figure 2.2.1.1 Vantage 32LE system front end connections.

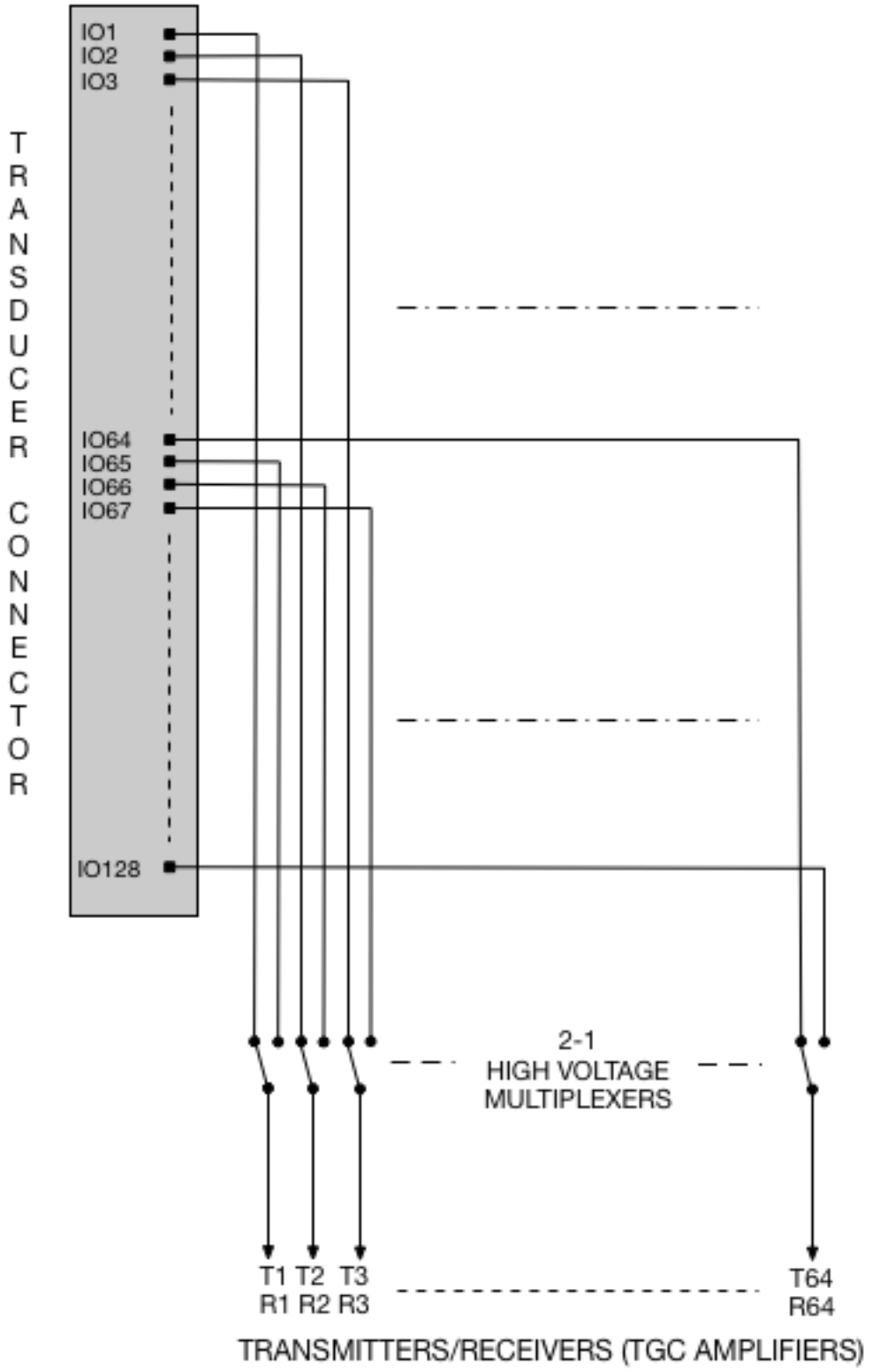


Figure 2.2.1.2 64 channel Vantage system front end connections.

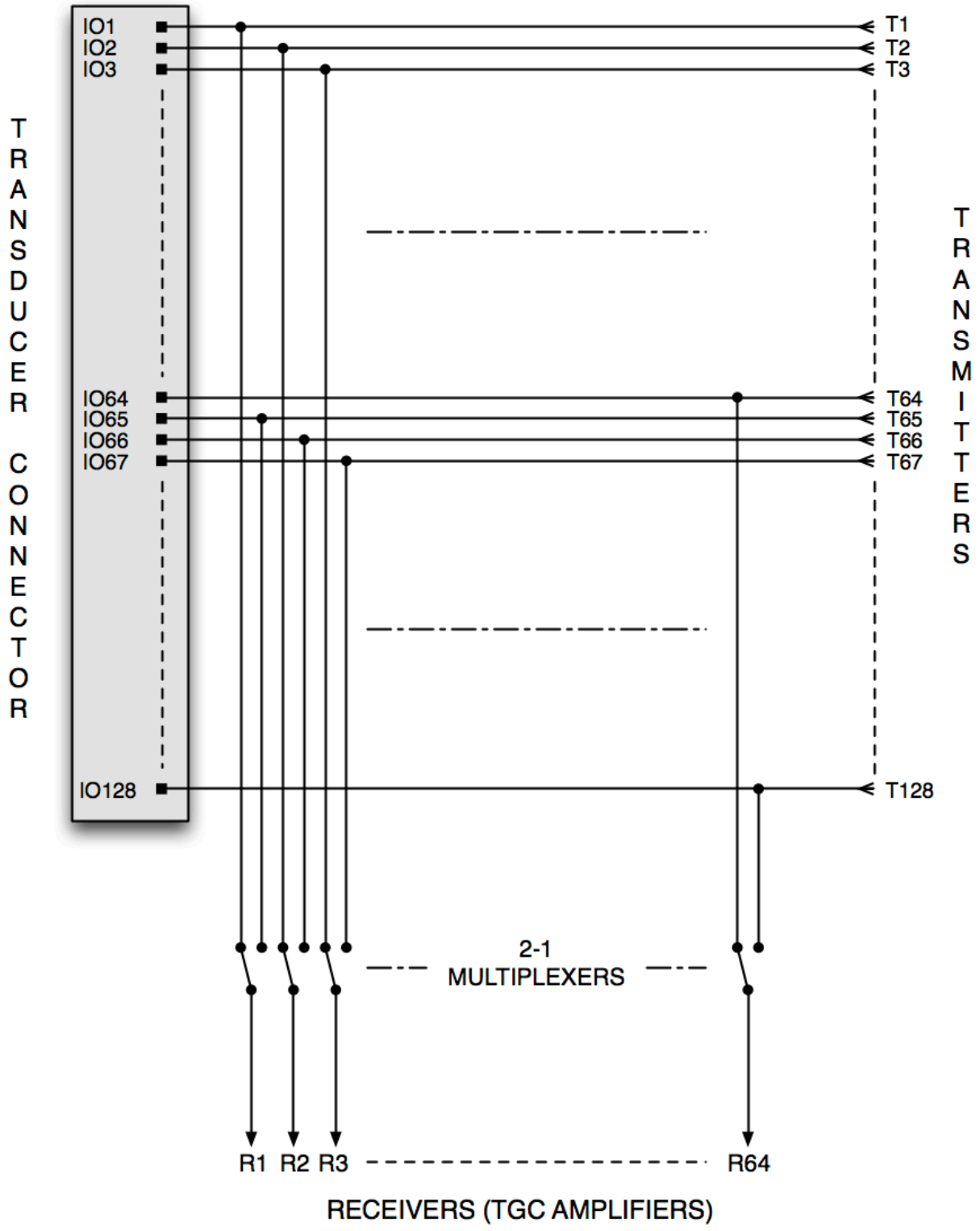


Figure 2.2.1.3 64 channel Vantage LE system front end connections.

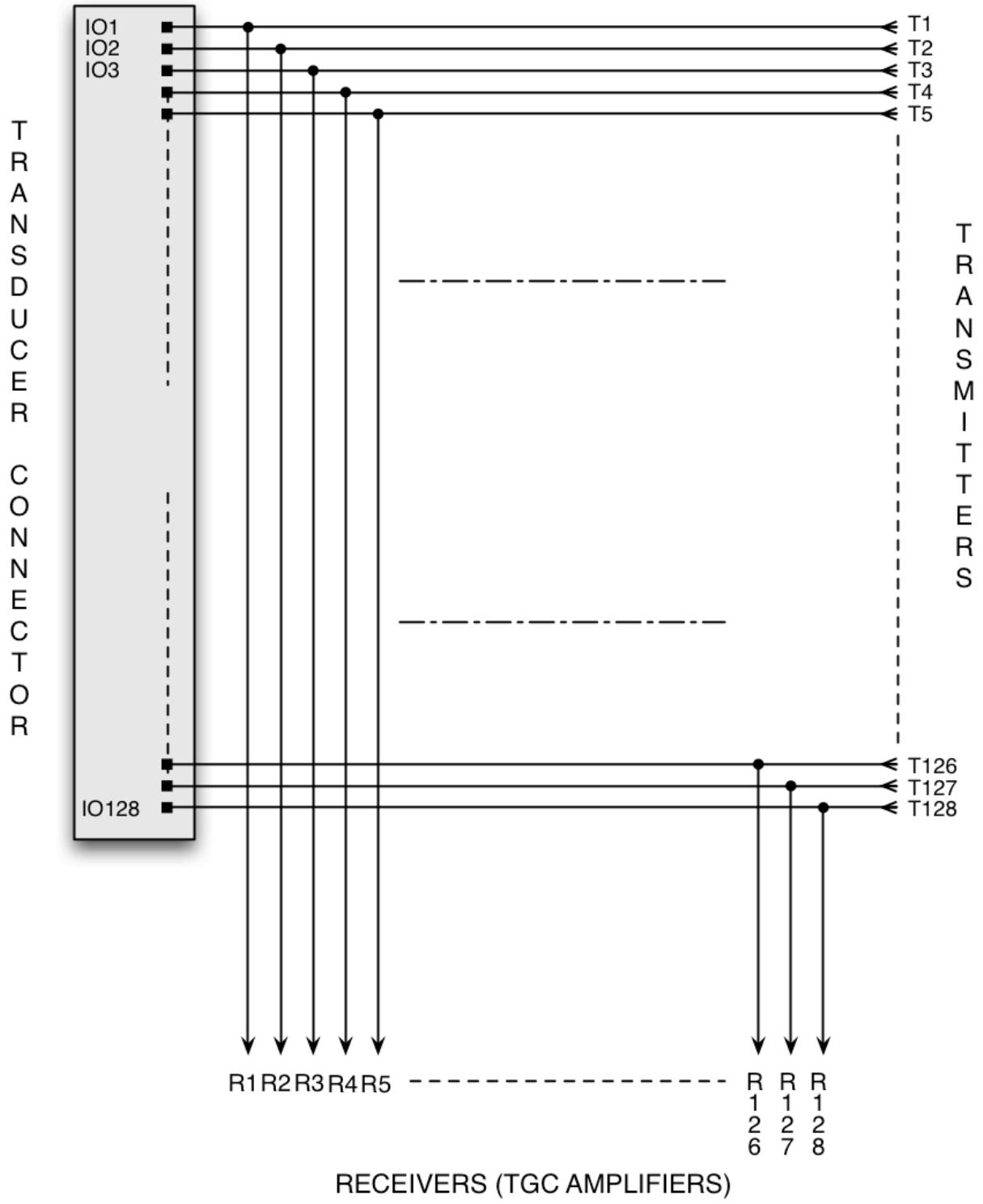


Figure 2.2.1.4 128 channel Vantage system front end connections.

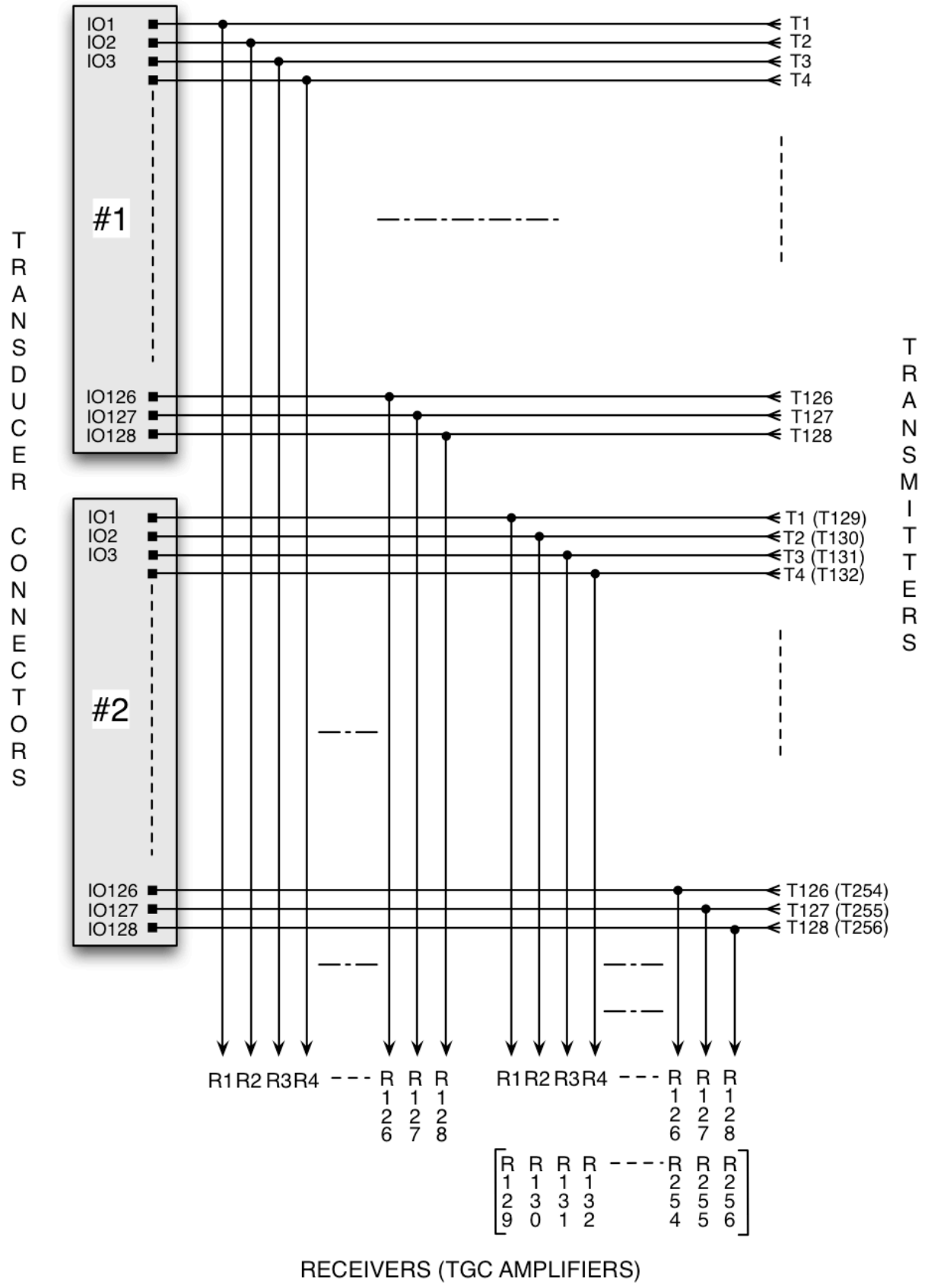


Figure 2.2.1.5 256 channel Vantage system front end connections.

2.2.3 Transducers with High Voltage Multiplexers

The Vantage system provides support for transducers with high voltage multiplexers in the probe, most often located inside the connector housing. This type of probe requires several power supply levels for the multiplexer components and digital control signals to set the switch patterns. Transducers with larger element counts than the number of element signals available at the probe connector can be supported, such as the 192 element L12-3v with HVMux switching to 128 system channels. The support for multiplexed probes includes the following capabilities:

1. Hardware-level support for HVMux switches within the probe: The Vantage hardware system provides the HVMux DC power supplies and logic-level control signals at the probe connector, using programmable signal formats and voltage levels that are specified in the `Trans.HVMux` structure.
2. HVMux Aperture definition: For any transmit or receive activity through an HVMux probe, the desired subset of transducer elements to be selected through the HVMux switches must be specified. The Vantage software provides utilities to define an array of HVMux Aperture selections to be used in an acquisition sequence, and the corresponding set of HVMux programming tables to control the HVMux switches within the probe.
3. Per-Event HVMux Aperture selection: For an HVMux probe, an ‘aperture’ field is added to every TX and Receive structure to identify which HVMux Aperture will be used while that TX or Receive is being executed by the system. The Vantage hardware system provides control logic to program the HVMux switches within the probe prior to the start of the acquisition event, to select the desired Aperture. Note that when desired, a different Aperture can be selected for the Transmit and for the Receive activity within the same acquisition Event.
4. Fault detection and probe protection: By their nature, HVMux switches are fragile devices. If transmit-level signals are applied to an HVMux probe while the HVMux power supplies are not enabled to the correct levels, the HVMux switches can be easily destroyed. To minimize the chances of an occurrence such as this, the system includes fault detection features to abort transmit activity if the HVMux power supplies are not enabled and functioning properly. If the presence of an HVMux probe has been detected (as identified by computeTrans through the presence of the `Trans.HVMux` structure), the system will not allow use of an incompatible SetUp script and will not allow use of Extended Transmit or HIFU features that may use transmit power levels that could be damaging to the HVMux devices.

Trans.HVMux Structure Definition: When a multiplexed transducer name is specified in a user script, the computeTrans.m function adds the following structure to the Trans object.

```
Trans.HVMux =

    highVoltageRails  double  % positive / negative HV supply in Volts
    logicRail         double  % logic supply rail in Volts
    clock             double  % shift clock rate in MHz
    clockInvert       double  % 0 (default) = false, 1 = true
    polarity          double  % data level to turn on a switch (1 or 0)
    latchInvert       double  % 0 (default) = false, 1 = true
    settlingTime      double  % switch transition time in usec
    type              string  % 'perEL', 'perCH', or 'preset' (see text)
    utaSF             double  % 0 for HVMux probe, >0 Mux UTA (see text)
    MuxMap(nEL,n)     double  % EL to mux switch bit map (see text)
    VDASAperLgth      double  % Length in bytes of VDASAperture columns
    ApertureES(nEL,n) double  % Aperture definitions, ES mapping (see text)
    Aperture(nEL,n)   double  % Aperture definitions, CH mapping (see text)
    VDASAperture(m,n) [uint8] % HVmux programming tables (see text)
```

The first seven attributes of `Trans.HVMux` as listed above specify the power supply levels and operating characteristics of the high voltage multiplexer circuits in the probe. These parameters will be programmed into the hardware system when the probe is selected. The Vantage system has made these parameters programmable to accommodate the wide variety of HVMux switch devices that have been used in commercial HVMux probes (or that may be used in a custom probe a Vantage customer is developing). It is critical that these values be set to match the requirements of the devices being used; incorrect settings may make it impossible to program the HVMux switches correctly, and incorrect supply voltage levels could be destructive to the HVMux devices.

Trans.HVMux.type is a string variable defining one of three levels of programmability in the tables:

- **'perEL'** means there is a separate SPST HVMux switch for each element, and each switch has its own bit in the `Trans.HVMux.VDASAperture` table. This allows you to select multiple elements in parallel connected to the same system channel.
- **'perCH'** means each bit in the `Trans.HVMux.VDASAperture` table controls an SPDT switch where the 0 state selects a different element than the 1 state. As a result, you cannot select both of those elements in parallel.
- **'preset'** means the probe uses a hard-coded HVMux programming scheme that cannot be modified through the Vantage software. In this case, you can only select from the predefined set of available Apertures and cannot take advantage of the Dynamic HVMux Programming features to define an arbitrary Aperture.

Trans.HVMux.utaSF: This field is set to the "special feature" index value in the UTA module ID, for UTA modules that incorporate HVMux switching. (For example, it will be set to 2 when the UTA 260-Mux is in use). For non-Mux UTA's running with a probe that has internal HVMux switching, this field must be set to zero. This field allows the

system to distinguish HVMux probes from HVMux UTA's, to facilitate determining whether the system configuration is compatible with the user script's intent (and to help enforce the restriction that an HVMux UTA cannot be used with an HVMux probe).

Trans.HVMux.MuxMap: This array identifies which bit in the HVMux programming table needs to be set to select a particular element through the HVMux switches. It is used by the Vantage software to create the `Trans.HVMux.VDASAperture` array.

Trans.HVMux.VDASAperLgth specifies the length in bytes of each HVMux programming table. It is used by the Vantage software to create the `Trans.HVMux.VDASAperture` array.

Trans.HVMux.ApertureES: This is a 2D array of doubles. Each column in this array defines one of the active Apertures that has been defined for the associated transducer. Within each column there is an entry for each element in the transducer. The value at each entry will either be zero, indicating that element is not included in the active Aperture, or a non-zero value which both indicates the element is active (i.e. connected through the HVMux switches) and also identifies the Element Signal to which the element is connected. Note that each column of `Trans.HVMux.ApertureES` is simply a direct copy of `Trans.ConnectorES`, but with the entries for de-selected elements set to zero. For transducers with predefined apertures, these apertures will be those that allow moving the 128 connector channels across the face of the array by one element at a time.

For an example, consider the L12-5 38mm transducer, which has the first 64 connector channel connections multiplexed to either elements 1 through 64, or elements 129 through 192. To define the aperture as the first 128 elements in the 192 element array, the `Trans.HVMux.ApertureES` array would be defined as:

```
Trans.HVMux.ApertureES(1:192,1) = [1:128,zeros(1,64)];
```

In this case, the first 64 connector channels are connected to the first 64 transducer elements, and the last 64 connector channels are connected to elements 65-128.

To define the aperture as the last 128 elements in the 192 element array, the `Trans.HVMux.ApertureES` array would be defined as:

```
Trans.HVMux.ApertureES(1:192,65) = [zeros(1,64),65:128,1:64];
```

In this case, the first 64 channel connections are connected to the last 64 elements in the 192 element array, while the last 64 channel connections are connected to the middle 64 elements.

Known transducers such as the L12-3 38mm transducer have predefined ApertureES arrays defined by computeTrans.m. These defined apertures will allow moving the 128 connector channels across the face of the transducer by one element at a time. For the 192 elements of the L12-3 38mm, there will be 65 columns in the `Trans.HVMux.ApertureES` array. To illustrate the mapping, the first 10 Aperture arrays are shown below. (The last aperture, number 65, was given above.)

```
Trans.HVMux.ApertureES(:,1) = [1:128,zeros(1,64)];
Trans.HVMux.ApertureES(:,2) = [0,2:128,1];
Trans.HVMux.ApertureES(:,3) = [0,0,3:128,1:2];
Trans.HVMux.ApertureES(:,4) = [zeros(1,3),4:128,1:3];
Trans.HVMux.ApertureES(:,5) = [zeros(1,4),5:128,1:4];
```

```
Trans.HVMux.ApertureES(:,6) = [zeros(1,5),6:128,1:5];
Trans.HVMux.ApertureES(:,7) = [zeros(1,6),7:128,1:6];
Trans.HVMux.ApertureES(:,8) = [zeros(1,7),8:128,1:7];
Trans.HVMux.ApertureES(:,9) = [zeros(1,8),9:128,1:8];
Trans.HVMux.ApertureES(:,10) = [zeros(1,9),10:128,1:9];
```

For specifying the ApertureES array to select the central 128 elements out of the 192 element array, we have the following definition:

```
Trans.HVMux.ApertureES(:,33) = [zeros(1,32),33:128,1:32,zeros(1,32)];
```

From a user perspective, the selection of a predefined aperture to use with a multiplexed transducer simply involves specifying the value of the second index of the `Trans.HVMux.ApertureES` array, which typically corresponds to the first element in the selected aperture (for linear arrays). In a user script, the aperture to use for a transmit/receive event is specified by the following structure attributes in the `TX` and `Receive` objects:

```
TX.aperture = n;
Receive.aperture = n;
```

For predefined apertures, the value of n would be equal to the number of the first element (counting from 1) in the start of a contiguous aperture of size equal to the number of connector channels, which is typically 128. For example, for the L12-3 38mm linear array, a value of n = 33 would specify an aperture that started at element 33 and ended at element 160. This aperture selects the central 128 elements of the 192 element array.

The use of different apertures on transmit and receive is allowed, so one can specify a different `Receive.aperture` number than the `TX.aperture` number for an Event with both transmit and receive specifications. In this case, the system will program the transmit aperture prior to the start of the event, then perform the transmits at the programmed delays. After the last transmitter has finished, the HV mux circuits will be programmed to the `Receive.aperture` number, and this reprogramming can take several microseconds to complete. The receive data sampling is in progress during this transition, since sampling started at the start of the transmit delay period. This means that until the HV mux circuits have been reprogrammed, there will not be valid receive data. Typically, the delay of the travel time through the transducer lens is sufficient to allow valid receive data to be acquired starting from the surface of the lens. If the attribute `Receive.aperture` is the same as the `TX.aperture`, it is still required in the `Receive` structure, since the aperture is needed to determine how to permute the `Receive.Apod` values to the individual receive channels.

Trans.HVMux.Aperture: This is a “dependent”, or read-only field that will be created by the Vantage software when the user script has been initialized for execution on a particular hardware system configuration. `Trans.HVMux.Aperture` contains the composite mapping from transducer elements to system channels, created by combining `Trans.HVMux.ApertureES` (mapping from elements to Element Signals at the connector) with `UTA.TransConnector` (mapping from Element Signals through the UTA module to system channels). Note that the relationship between

`Trans.HVMux.ApertureES` and `Trans.HVMux.Aperture` parallels the relationship between `Trans.ConnectorES` and `Trans.Connector` for non-HVMux probes.

Trans.HVMux.VDASAperture is another “dependent”, or read-only field that will be automatically created by the Vantage software when initializing the system. `Trans.HVMux.VDASAperture` is the array of uint8 byte values that will be sent to the hardware system for programming the HVMux switches. This will be a 2D array of size NB x NA, where NB is the number bytes in the HVMux programming table for one aperture (specified by `Trans.HVMux.VDASAperLgth`), and NA is the number of Apertures that have been defined (so NA will always be equal to the number of columns in `Trans.HVMux.ApertureES` and `Trans.HVMux.Aperture`).

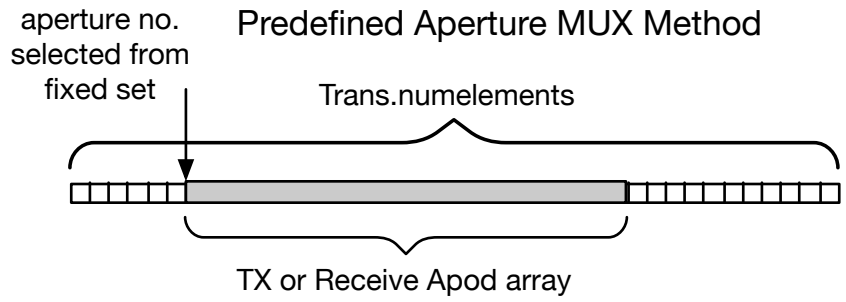
2.2.4 MUX programming methods

2.2.4.1 Predefined HVMux “Active Aperture” functionality:

The “Active Aperture” HVMux programming scheme, as used in all previous Vantage software releases, is still fully supported in software release 4.0.0 and later. The Active Aperture programming scheme is intended primarily to support linear or curved “1D” array probes, where the HVMux switching is used to select a contiguous group of elements with the size of the group set to match the number of system channels available from the system. As explained above, the predefined mux apertures are contiguous apertures, with the number of the first element in the aperture used to specify the aperture in `TX.aperture` or `Receive.aperture`. The size of the aperture is equal to the number of I/O channels supported by the system (generally 128 or 64). The number of available aperture choices for a 128 channel system and a 192 element transducer would be 65, with the first aperture starting at element 1 and the last aperture starting at element 65. For “Active Aperture” programming with the Vantage system, all of the following constraints must be met:

- The complete set of available contiguous apertures must be predefined in the Trans structure (this is done automatically by computeTrans).
- The length of every available aperture selection must be identical, and equal to the number of system channels being used by the probe.
- The aperture index value for each available aperture is equal to the first element number within that Aperture.
- Apod arrays used in the script must have length equal to the number of system channels being used by the probe. There is a one-to-one mapping from entries in the Apod array to the elements included in the selected Aperture.
- The user script must specify the desired Aperture through the ‘aperture’ field in every TX and Receive structure.

When using the Active Aperture method, the size of the TX.Apod and Receive.Apod arrays should be the same as the number of I/O channels (128 for the Vantage 128/256 and Vantage 64LE, 64 for the Vantage 64 and 32LE). The apodization function is applied to the contiguous elements in the mux aperture (see figure below).



2.2.4.2 Dynamic HVMux Apertures -

Many multiplexed transducers can support non-contiguous HVMux apertures. These transducers have multiplexer circuits that are effectively single-pole-single-throw switches that can connect one or more elements to a single channel. This allows selecting a complex, non-contiguous aperture, as long as it is supported by the hard wiring of the multiplexer circuits. If the dynamic mux method is used for programming the mux apertures, it must be the only method used, since the `TX.Apod` and `Receive.Apod` arrays must all be the length specified by `Trans.numelements`.

To use the dynamic mux method, the TX.Apod and Receive.Apod arrays must be defined to be the same size as the number of elements in the transducer (`Trans.numelements`). The aperture pattern is then determined by the non-zero values in the Apod arrays, using a utility function - computeMuxAperture.m, which takes as arguments the Apodization array and the transducer structure (Trans). If a user specified pattern cannot be implemented with the transducer's multiplexer wiring, an error will be generated when the function is executed.

As an example, consider the case where the user wants to use every other element of a 128 transducer with the Vantage 64 using the 260-MUX UTA. Since the MUX wiring of the 260-MUX UTA allows a channel to be connected to either element n or n+64, it is not possible to select all the odd elements of the transducer. However, it is possible to select odd elements from 1 to 63 and even elements from 66 to 128. To accomplish this, the `TX.Apod` or `Receive.Apod` arrays can be programmed as follows:

```
TX.Apod = zeros(1,Trans.numelements);
TX.Apod(1:2:63) = 1;
TX.Apod(64:2:128) = 1;;
```

One would then use the `computeMuxAperture.m` utility function to compute an aperture number.

```
TX.aperture = computeMuxAperture(TX.Apod, Trans)
```

The computeMuxAperture function will define an HVMux programming Aperture based on the non-zero entries in Apod. It will then search through the existing Trans.HVMux.Aperture array, to see if the new Aperture matches any of the existing ones. If so, the return argument 'aperture' will be set to the index of that matching Aperture or if not, the newly created Aperture will be appended to the end of the Trans.HVMux.Aperture array and 'aperture' will point to that one. You should assign the returned aperture value to the associated TX.aperture or Receive.aperture field in the script. In addition to returning the aperture index value, computeMuxAperture will also

write the updated Trans structure back to the caller's workspace so the system will have access to the full Trans.HVMux.Aperture array.

When computeMuxAperture is called to create an Aperture, it will check to see if more than one element associated with the same system channel is being selected and if so, will determine if selecting parallel elements is possible for the HVMux probe or UTA being used. If the HVMux switching cannot support the Apod array that has been specified, computeMuxAperture will exit with an error message identifying the problem.

In summary, to use the dynamic mux method, 1) size the Apod arrays to be the same size as the number of elements in the transducer. 2) Specify an apodization pattern that is consistent with the transducer or UTA multiplexer wiring. 3) Call the computeMuxAperture utility function to compute the MUX programming bit pattern and return the aperture number. 4) Use the aperture number in TX.aperture or Receive.aperture whenever the MUX aperture is desired. There is no need to call the computeMuxAperture function if the aperture number of a previously defined aperture is being re-used.

Regardless of how the TX.aperture and Receive.aperture fields were defined, when VSX initialization calls VsUpdate to process each TX and Receive structure it will check the Trans.HVMux.Aperture column selected by 'aperture' to confirm it supports all active elements in the associated Apod array. If this is not the case the script will exit with an error message identifying the affected structure, so you can correct it.

After the VsUpdate processing of all TX and Receive structures is complete, the "computeHvMuxVdasAperture" utility function will be called by VSX, to create the Trans.HVMux.VDASAperture array containing the data for programming the physical HVMux chips to implement the corresponding Trans.HVMux.Aperture.

2.2.5 Connecting a custom probe to the system

If you have a transducer you would like to use with the Vantage system that is not supported by the computeTrans function, there are two tasks you must complete before you can develop SetUp scripts to run with the probe: First, determine how you will connect the probe to the system and second, develop the Trans structure to be used with the probe in your scripts. These three steps are described separately below:

Connecting the probe: The approach to be taken here depends on the nature of your custom probe, as described in the bullet items below.

- **Commercial Probe, supported by existing UTA module:** If your probe is from a commercial manufacturer, and is from a probe family supported by commercially available ultrasound systems, the first step is to determine whether there is a Vantage UTA module available that supports that probe family. Refer to the application note "AppNote Vantage UTA Modules" for a listing of all UTA Modules that are currently available and the commercial probe family supported by each module. (This application note is included with software releases 4.2.0 and later; contact Verasonics Support for a copy if you are using an earlier software release.) If your probe is supported by one of the listed modules, all you need to do is obtain the required UTA module from Verasonics (if you don't already have it) and plug in the probe. At that point you can run the utility "ProbeConnectorStatus" from the Matlab command prompt. It will identify the UTA

module and connector types on that UTA, and identify the probe that is plugged in. If the probe has an ID that is recognized by computeTrans, the name of the probe will be listed and you will be able to use computeTrans to create the required Trans structure- you are ready to start developing SetUp scripts for the probe. If the probe is not supported by computeTrans, the ProbeConnectorStatus utility should report it as "unrecognized" but will list the probe ID if an ID is available from the probe.

- **Commercial Probe, not supported by any Vantage UTA module:** If your commercial probe uses a connector interface that is not covered by any existing UTA module, you have essentially two options as describe below. But for either option, you must first determine how the probe is wired to its connector in terms of element signal pinout and any other features (such as HVMux switching) that are required for use of the probe. This may be a difficult task and is beyond the scope of this document; contact Verasonics support for guidance and other application notes that may be useful.
    - **Build or obtain an adapter:** If you know the connector pin assignments and signal requirements for your probe, you can build an adapter circuit board containing a receptacle that is compatible with your probe, and a plug that is compatible with one of the existing Vantage UTA modules. The circuit board should contain full ground planes in between each signal layer, with the ground planes directly connected to every ground pin at both connectors. Element Signals and other required ID or control signals can be routed between the associated pins at each connector. If you don't have the resources to develop and build an adapter, there are some third-party vendors who may be able to do this for you (or who may even have a compatible adapter already available). Contact Verasonics Support for guidance, and some vendors that we are aware of.
    - **Remove the connector**, and connect the probe cable to a new connector that is compatible with the Vantage system. This may be a very difficult task but it can be done if you have access to design information for the probe, and the resources and facilities needed to re-wire the cable to the new connector interface. Verasonics has "backshell kits" available for some UTA module connector types, to facilitate this task. Note that one significant disadvantage of this approach is that you will no longer be able to use the probe with the commercial system it was designed for.
- **Custom Probe**, that you are either developing yourself or are having built for you by a probe vendor. In this case you already have the detailed design information for the probe, in terms of number of element signals and any other features needed by the probe. The simplest approach is to identify a Vantage UTA module with a connector interface that supports the required number of element signals and other features required by the probe, and for which there is a back shell connector kit available from Verasonics that you can purchase to simplify the task of connecting the probe cable to the connector. If you are working with a probe vendor, they may already be aware of the Vantage back shell kits and/or have their own connector designs that are compatible with the Vantage system.

Developing the Trans structure: Once you know how your probe will connect to the Vantage system and how the element signals and other features will be wired, you will need to create a Trans structure specific to your probe. The simplest way to do this is to just define the Trans structure directly within your SetUp script. If you plan to write multiple SetUp scripts for use with the same probe, it may be easier to define a Matlab function that provides the Trans structure and just call that function from within each individual SetUp script (that way, you only have to define the structure once and if you decide to change some Trans parameter you won't have to duplicate that change in multiple scripts). Refer to the opening paragraphs of section 2.2 for detailed definitions of each Trans structure field, and which fields are required.

If your custom probe will include HVMux switching, contact Verasonics support for some application notes that will provide more guidance on how to determine the required values for all of the `Trans.HVMux` attributes.

Writing SetUp scripts for your probe: After you have the ability to connect your custom probe to the system, and have developed a Trans structure to support the probe, you are ready to write some scripts to use with the probe. A good way to start is to first develop a very simple imaging script such as the "Flash" or "Flash Angles" examples included with the system. You can start by finding an existing example script for a probe that is similar to your custom probe- make a copy of that script, rename it to match your probe and edit it to use the Trans structure for your probe and make any other adjustments needed to get the script working well. It is usually much easier to start with a known, working script and make incremental changes to it rather than creating an entire new script from scratch. Once you have a basic imaging script working well with your probe you will be ready to develop additional scripts tailored to the specific applications you are interested in.

2.3 PData Object

The PData object (short for 'pixel data') describes the pixel region(s) to be processed by the image reconstruction software. If no image reconstruction is required, this object can be omitted from a setup script. When image reconstruction is performed, the PData object specifies one or more pixel regions in a larger image space, which is defined relative to the transducer coordinate system. Image reconstruction is only performed at the pixel or voxel locations included in the regions specified. The structure definition is as follows:

```
PData =
    sFormat         double          % (deprecated) no. of SFormat structure
    Coord           string          % ['rectangular'],'polar','spherical'
    PDelta          [1x3 double]    % spacing between pixels in all dimensions
    Size            [1x3 double]    % rows, cols and sections
    Origin          [1x3 double]    % % x,y,z of top lft corner (for 2D) or top
                                      lft far corner (3D view from z axis).
    Region          [numRegions Structs] % Region structures
```

As mentioned previously, the `sFormat` field is deprecated since SFormat is no longer needed to define Regions within the PData space. The new method, explained below, uses PData.Region.Shape definitions to define reconstruction regions, and allows much greater flexibility in specifying regions of interest.

The `PData.Size` attribute is defined in rows, columns, and sections (for 3D volumes). For typical 2D scans in rectangular coordinates, the row dimension corresponds with the Z axis (increasing rows correspond to increasing Z values), while the columns dimension corresponds with the X axis. The section size is set to 1. In polar coordinates, the rows correspond to the R dimension, while the columns correspond to the Theta dimension. For 3D volume scans, the axes arrangement is changed, with the row dimension corresponding to the Y axis and the sections dimension corresponding with the Z axis. In this case, a section is a plane that parallels the plane of the X-Y axes.

The `PData.Origin` attribute defines the location of the PData region in the coordinate system of the transducer. For 2D scans, the `Origin` is always in the X-Z plane, with a y coordinate of zero, and for rectangular coordinates identifies the top left corner of the scan when viewing the scan above the X-Z plane with increasing Z going from top to bottom. For polar coordinates, the `Origin` is the location of the polar origin at R=0. For 3D volume scans, the `Origin` is typically in the X-Y plane, but can also have a z component. If one looks back towards the X-Y axes from a positive Z point of view, the origin of the PData 'box' is the far upper left corner of the volume.

For `PData.Coord = 'rectangular'` (the default), the PData array is always a rectangular area (for 2D scans) or rectangular parallelepiped (for 3D scans). The increment between pixels in the x, y, and z dimensions is provided in the `PData.PDelta` attribute. [For backwards compatibility, the increment can still be set by the single attribute, `PData.pdelta`, or the individual attributes of `pdeltaX`, `pdeltaY`, and `pdeltaZ`, although the use of these attributes is no longer recommended.] In linear array scans, where the depth resolution is typically better than the lateral resolution, one can specify a `PDelta(3)` or `z` increment which is appropriate for the depth resolution, and a `PDelta(1)` or `x` increment appropriate for the lateral resolution. When a non-square pixel PData region is rendered to a display window, the pixels will be interpolated to square pixels at a pixel delta set by the display window specification.

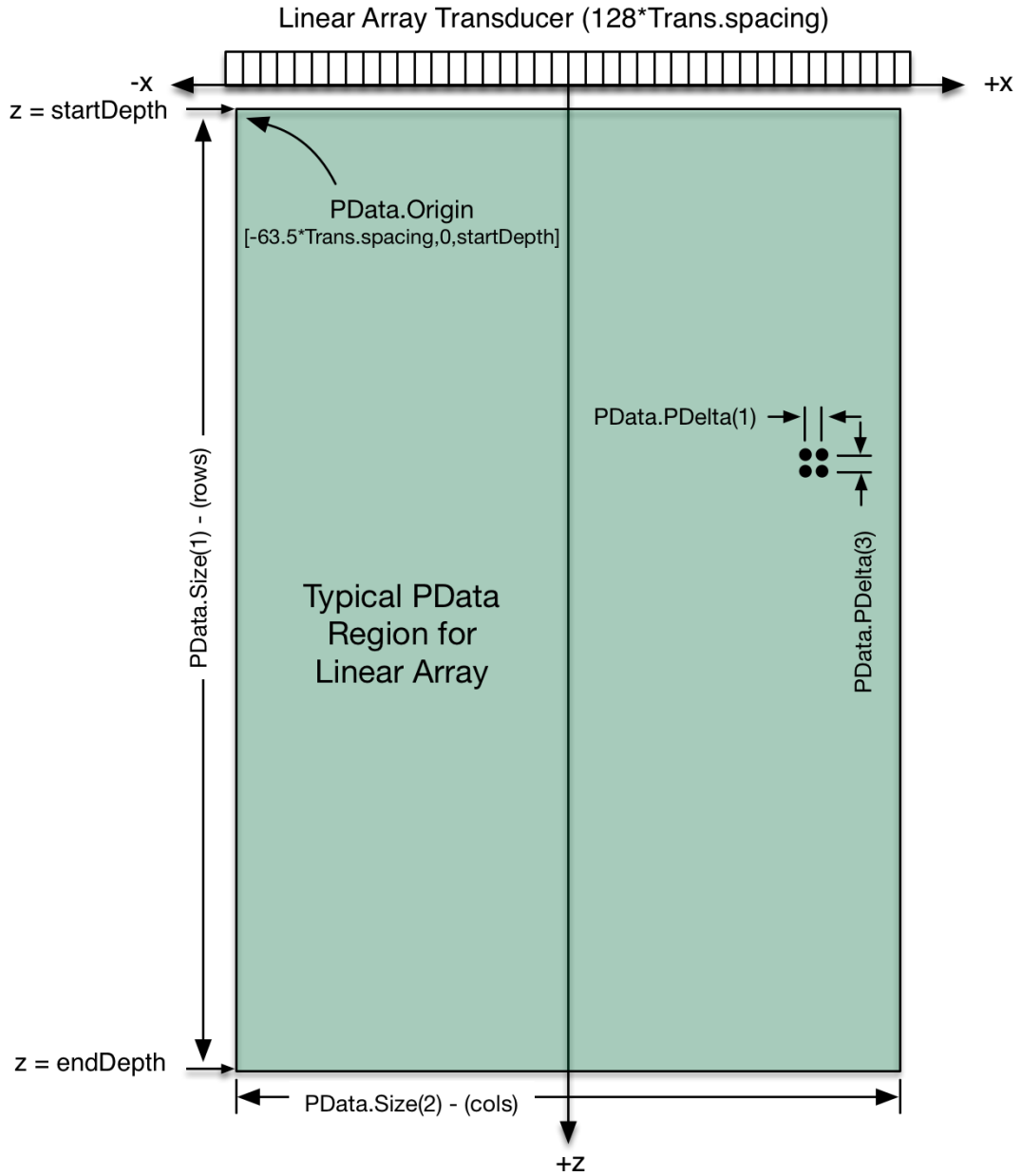


Fig. 2.4.1 Rectangular coordinate PData specification for linear array.

For `PData.Coord = 'polar'` the `PDelta` values are interpreted as theta, r, and z (z is only needed for cylindrical coordinates, which are not yet supported). In this case, the PData area is a section of a circle, with the apex at the `PData.Origin` coordinates, which are specified in the coordinate system of the transducer (typically rectangular). The angle increment, theta, specifies the increment in the counter-clockwise angle from the vertical line through the apex. The theta lines are always positioned symmetrically about a line through the apex that is parallel to the z axis (see Fig. 2.4.2 below).

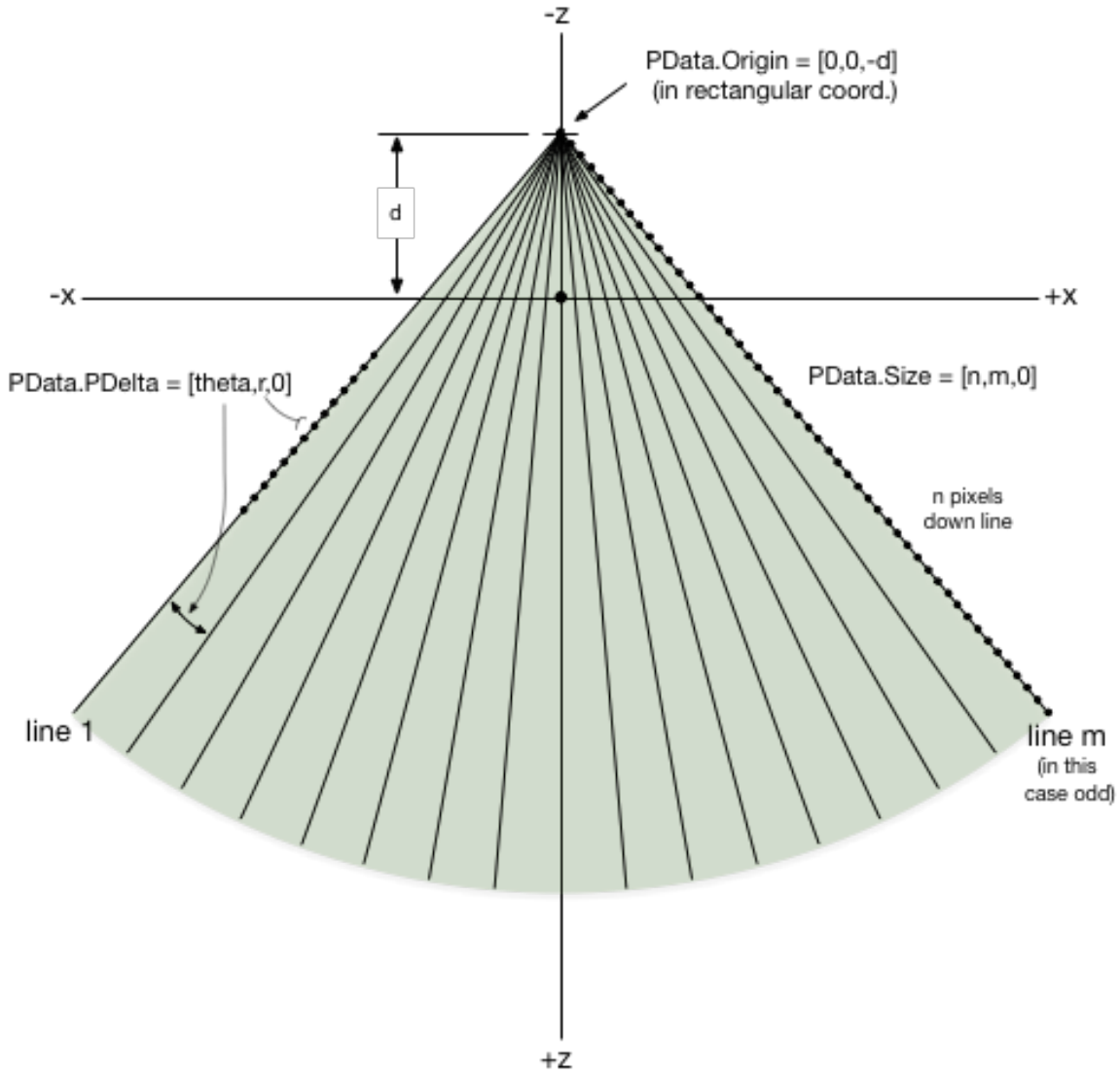


Fig. 2.4.2 Polar Coordinate PData Specification

The `PData.Size` specification sets the size of the pixel area in pixels for 2D scans or voxels for 3D scans. For rectangular coordinates, the row dimension corresponds to the z direction and the column dimension to the x direction, with a 1 specified for the third dimension for 2D scans. For 3D scans, the row and column indices correspond to the x and y dimensions respectively, with the third or sections index equal to the z dimension. For polar coordinates, the row index corresponds to the r coordinate and the column index corresponds to the theta dimension. Since the polar region is symmetric about a vertical line through the apex, the theta = 0 direction is always in the center of the `PData` columns. For example, if the theta size, `PData.Size(2) = 100`,

the theta = 0 line would be halfway between column 50 and 51. If the theta pixel delta, `PData.PDelta(1) = 0.01` radians, the `PData` region would be a section of a circle with apex angle of 0.01*(100-1) radians or 56.72 degrees. Column one would be at an angle of -49.5 * .01 radians or -28.36 degrees, while column 100 would be at +28.36 degrees.

There can be multiple regions defined inside the area of the `PData` array with various geometric shapes. Image reconstruction is only performed on the pixels or voxels included in the regions defined.

2.3.1 PData.Region Objects

The Region objects specify the pixels that belong to the various pixel regions that are contained in a `PData` object. The `PData.Region` structures describe the subset of pixels in the `PData` array that will processed, and this subset is typically determined by the type of transducer and the format of the scan of the media. For example, a sector format for a phased array transducer can be specified in a `PData` rectangle by creating a `PData.Region` structure that specifies a sector shaped subset of pixels to be processed. With the release of the 3.x software, the pixels to be processed are defined by a list of the linear indices of pixels from within the `PData` array. This allows the creation of arbitrary shapes and even regions that are not contiguous.

The simplest Region specification is a specification of the linear addresses of the pixels in the `PData` array that make up the Region. The attributes are as follows:

```
PData.Region =
    numPixels   double    % no. of pixels in region (or voxels in 3D section)
    PixelsLA   [1xnumPixels]int32 % linear address of each pixel (from 0)
```

The `PixelsLA` array in `PData.Region` contains a list of the linear addresses of the `PData` array for the pixels contained in the region, where the indices start from zero instead of 1 (subtract 1 from the Matlab indices). In this case, there are no additional attributes needed, as the definition is sufficient to define the `Region`. Multiple `Region` structures can be defined in the same `PData` space with different `PData.Region()` indices. The `ReconInfo` structures used for image reconstruction will reference the `PData.Region` index number to specify the `Region` to be processed.

Alternately, the user can specify only a shape for the `Region` structure, using the `Region.Shape` attribute. In this case, the shape attribute specifies a structure with additional attributes that are required to set the position, size and orientation of the shape. When specifying a `Shape`, the `numPixels` and `PixelsLA` attributes can be empty or missing, as they will normally be added or replaced by calling the 'computeRegions' utility function. For example, a rectangular shape is defined as below.

```
PData.Region.Shape =
    struct('Name','Rectangle','Position',[50,0,25],'width',32,'height',65));
```

Additional 2D shape names are supported for rectangular coordinate `PData` arrays, including 'Parallelogram', 'Trapezoid', 'Circle', 'Annulus', 'Triangle', 'Sector', and 'SectorFT' (flat top). 3D shapes supported are 'Cone' , 'Pyramid', 'Frustum', 'Section' and 'Slice'. Each shape has it's own unique attributes that define it's characteristics.

Currently, the only shapes supported for polar coordinate PData arrays are 'Sector' and 'SectorFT'.

The user can also define their own regions by specifying the shape name as '`Custom`'. In this case, the `PData.Region` attributes of `numPixels` and `PixelsLA` must be provided to identify the pixels defining the region. Again, provide the int32 linear index of the pixels in `PixelsLA` starting from zero rather than one.

If the Region structure contains a supported Shape attribute, the utility function 'computeRegions.m' can be used to compute the `numPixels` and `PixelsLA` attributes for the Region. The utility is automatically called by VSX if the `numPixels` and `PixelsLA` attributes are not found in the user's setup script or the `PData.Region` structure is missing or empty. The computeRegions.m utility (note name change from previous releases) no longer divides up the `Region` into multiple parts for reconstruction thread processing, as this in now done automatically by the reconstruction routine. If no `PData.Regions` are specified, a single `PData.Region.Shape` is added by the computeRegions utility, named 'PData', which specifies the entire `PData` space.

Shapes can be combined to create more complex Regions. In the `Shape` structure, if the attribute `'andWithPrev'` is provided, with a value equal to the index of a previously defined shape, the new shape is 'anded' with the previous shape, giving the intersection between the two shapes. For example, one might want to steer the beams of a linear array for Doppler sensing, but keep the processing for the beams within the region for imaging where the beams have no steering. The Shape for the 2D image `Region` would be defined first as a rectangle matching the format of the 2D scan. Then the `Region` for the steered beams can be defined as a parallelogram and 'anded' with the first Region, as shown in the figure below.

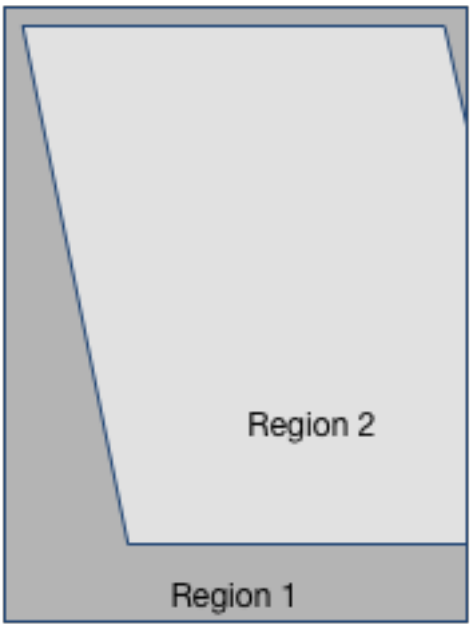
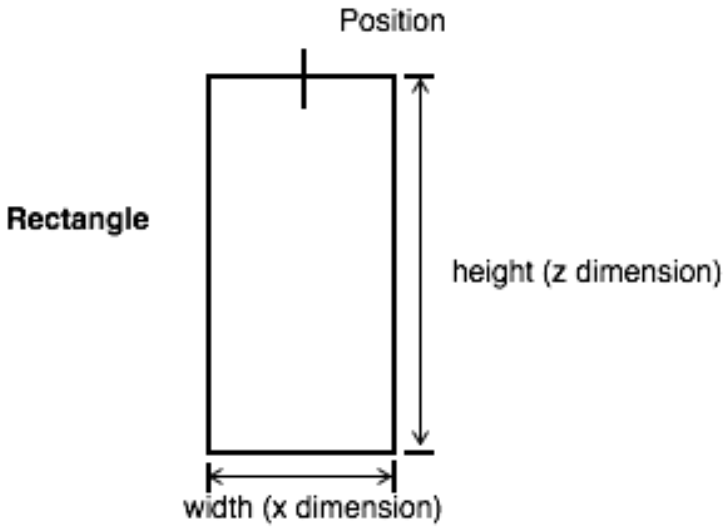


Fig. 2.4.1.1 Using `'andWithPrev'` to intersect two Regions.

The supported `PData.Region.Shape` definitions are provided in the text below.

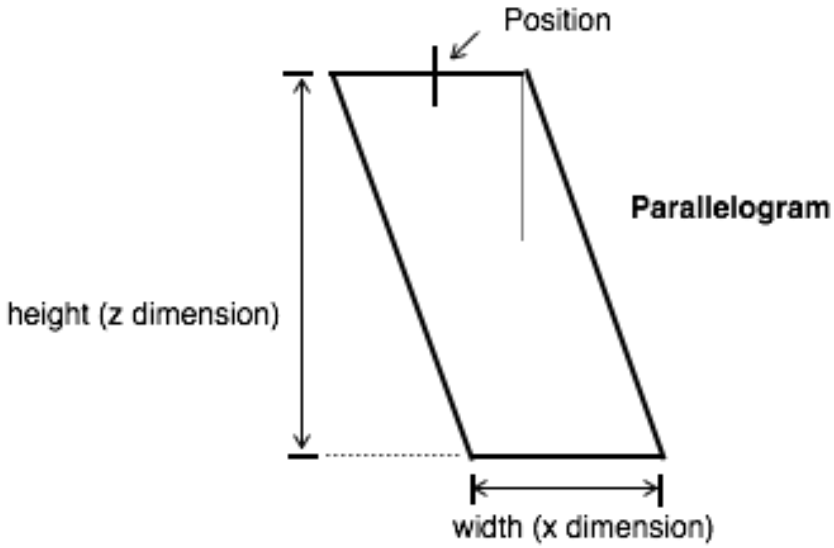
The `Rectangle` shape structure is defined as follows:

```
Shape =
    Name            string       % 'Rectangle'
    Position        [1x3 double] % x,y,z coordinate of center of top segment
    width           [double]     % x dimension
    height          [double]     % z dimension
```



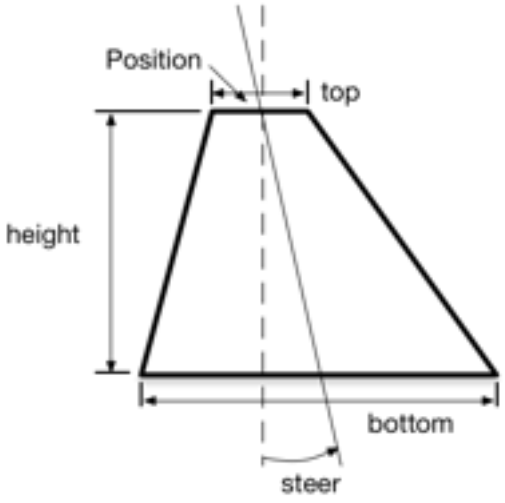
The Parallelogram shape simply adds a steering angle to the Rectangle attributes.

```
Shape =
    Name            string       % 'Parallelogram'
    Position        [1x3 double] % x,y,z coordinate of center of top segment
    width           [double]     % x dimension
    height          [double]     % z dimension
    angle           [double]     % angle between vertical and left side
```



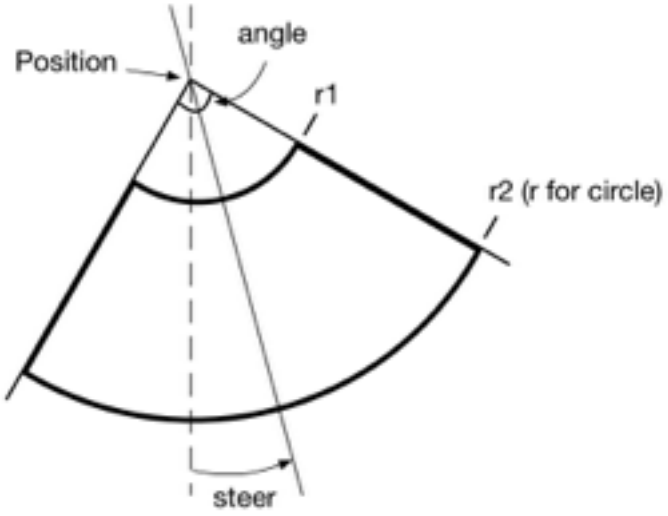
The Region.Shape structure for a trapezoid or triangle shape is defined as follows (for a triangle shape, either the top or bottom length should be set to zero):

```
Shape =
    Name          string       % 'Trapezoid' or 'Triangle'
    Position      [1x3 double] % x,y,z coordinate of center of top segment
    top           [double]     % length of top segment
    bottom        [double]     % length of bottom base
    height        [double]     % height (z dimension)
    steer         [double]     % angle between z axis and line thru midpoints
```



The Region.Shape structures for 'Circle', 'Annulus' and 'Sector can all be defined with the following attributes (if an attribute is not needed, it can be missing):

```
Shape =
    Name          string       % 'Circle', 'Annulus' or 'Sector'
    Position      [1x3 double] % x,y,z coordinate of center of circle or apex
    r             [double]     % radius ('Circle' shape only)
    r1            [double]     % inner radius ('Sector' or 'Annulus' only)
    r2            [double]     % outer radius ('Sector' or 'Annulus' only)
    angle         [double]     % angle between lft and rt side of 'Sector'
    steer         [double]     % angle between z axis and centrln of 'Sector'
```

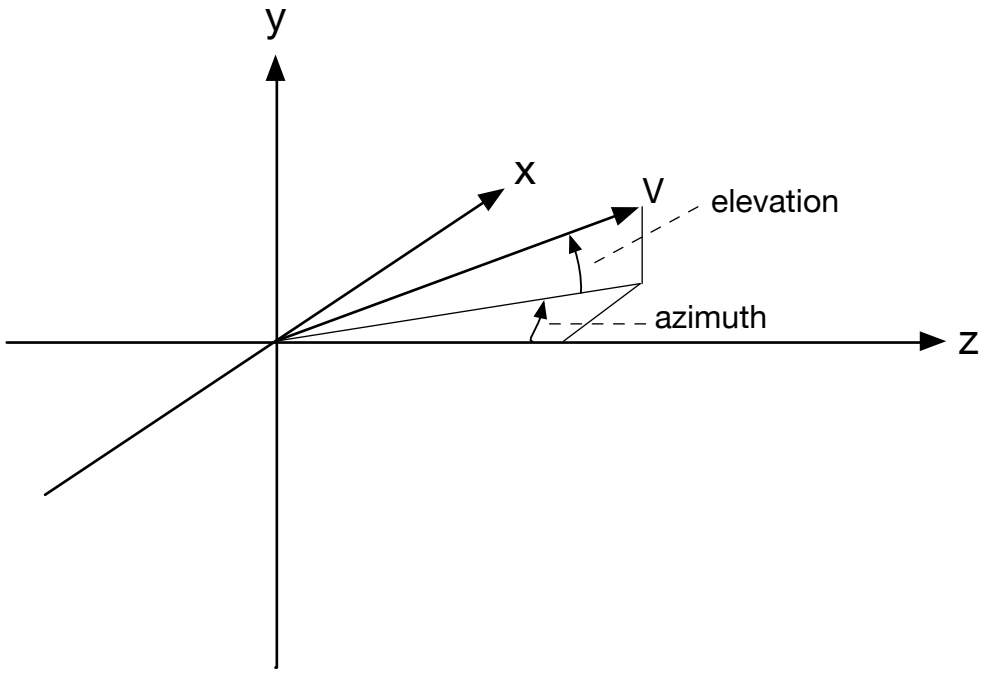


The ‘SectorFT’ shape definition is similar to the ‘Sector’ definition with the exception that the `r1` and `r2` attributes of the ‘Sector’ definition are replaced by `z` and `r` attributes. The `z` attribute defines the position of a horizontal line that cuts off the top of the sector.

If the `PData` coordinate system is defined as ‘polar’, the full `PData` region is not a rectangle, but instead a sector of a circle, symmetric about a vertical line through the `PData.Origin`. In this case, the Sector shape must have the same `Shape.Position` attribute as the `PData.Origin`.

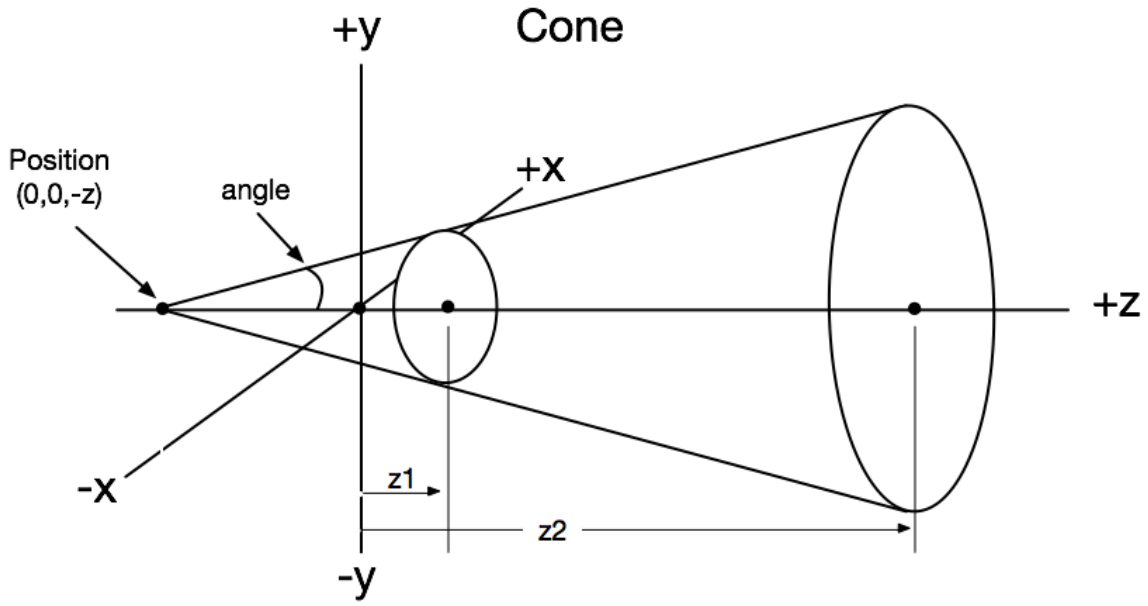
For 3D volume scanning, the PData.Size attribute contains 3 values - rows, columns and pages (pages are sometimes referred to as sections). The rows are parallel to the x axis and increase in the negative y direction, while the columns are parallel to the y axis and increase in the positive x direction. A page or section is then parallel to the x,y plane, with the rows and columns in their normal orientation when viewed from the positive z axis looking back towards the origin. The 2D transducer array elements are typically arranged in the x,y plane, centered around the origin of the z axis, but this arrangement is no longer mandated, and transducer elements can be positioned anywhere in the 3D space.

The coordinate system for 3D scans is shown below. Orientations of transducer element normals and vectors are specified using two angles - azimuth and elevation. These angles are measured from the positive z axis to the projection of the vector in the x, z plane (azimuth) and from the x, z plane to the vector (elevation). In the case of transducer element normals, the azimuth and elevation angles are specified as if the element center was located at the origin.



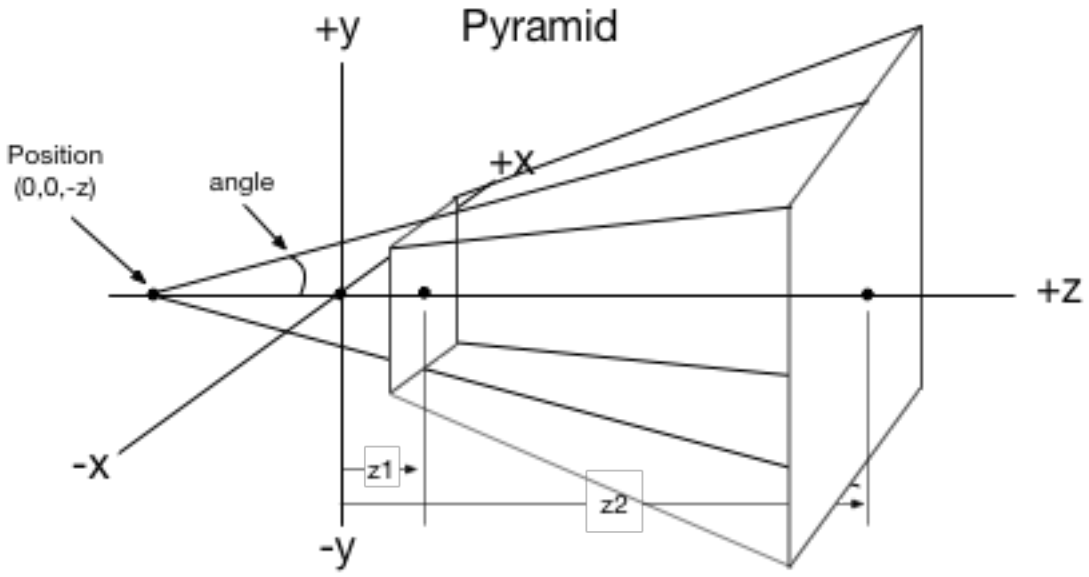
The 'Cone' `Region.shape` can be defined to specify a conical shape in the `PData` volume. The apex of the cone must lie on the negative z axis, with the angle and steering attributes specifying the conical shape. When steered, the cone is anchored at its apex, and still maintains a circular intersection with planes parallel to the x,y plane. The cone can also be optionally truncated (conical frustum) at two distances from the apex with planes parallel to the x,y plane. The 'Cone' `Region.Shape` structure is defined as follows:

```
Shape =
    Name            string       % 'Cone'
    Position        [1x3 double] % x,y,z coordinate of apex of cone
    angle           [double]     % angle - centerline to outer surface
    z1              [double]     % optional - distance to 1st cross section.
    z2              [double]     % optional - distance to 2nd cross section.
    Steer           [1x2 double] % optional - centerline azimuth and elevation
```



The ‘Pyramid’ Region.shape can be defined to specify a pyramidal shape in the PData volume. The apex of the pyramid must lie on the negative z axis, with the angle and steering attributes specifying the shape of the pyramid. When steered, the pyramid is anchored at its apex, but still maintains a square intersection with planes parallel to the x,y plane. The pyramid can also be optionally truncated at two distances from the apex with planes parallel to the x,y plane. The ’Pyramid’ Region.Shape structure is defined as follows:

```
Shape =
    Name          string       % ‘Pyramid’
    Position      [1x3 double] % x,y,z coordinate of apex of pyramid
    angle         [double]     % angle - centerline to outer surface
    z1            [double]     % optional - distance to 1st cross section.
    z2            [double]     % optional - distance to 2nd cross section.
    Steer         [1x2 double] % optional - centerline azimuth and elevation
```



Three dimensional PData arrays can have a Region that is composed by a range of pages or sections. The ‘Section’ shape is defined as follows:

```
Shape =
    Name          string       % ‘Section’
    Sections      [1x2 double] % start section, number of sections
```

The start section plus the number of sections must be less than the total sections of the `PData` array.

A three dimensional PData array can also be sliced by a two dimensional plane parallel to one of the planes formed by two axes. The ‘Slice’ Region is defined as follows:

```
Shape =
    Name          string       % ‘Slice’
    Orientation   string       % slice plane - ‘xz’,’yz’, or ‘xy’
    oPAIntersect  double       % intersect coordinate with out of plane axis
```

2.4 Media Object

The Media Object is only used when operating the system in Simulate mode, and describes the location and characteristics of simulated targets in the geometrical space of the transducer coordinate system. All objects in this space are defined by one or more point targets, whose location and reflectivity can be individually specified. The structure for the Media Object has the following attributes:

```
Media =
    model           string        % (optional) name of media model
    function        string        % (optional) Matlab function to execute
                                    for moving or modifying media points.
    FlowObj         struct        % one or more FlowObjs for Doppler acq.
    attenuation     double        % attenuation in dB/cm/MHz
    numPoints       double        % no. of media pts (incl. FlowObjs)
    MP              [numPointsx4 double] % x,y,z,reflectivity(0-1.0)
```

The `Media.function` attribute is used for dynamic media models, where the targets change their positions or characteristics between successive acquisitions. When present, and the `Receive.callMediaFunc` attribute is true (=1), the Matlab function (m file) specified by the string is called prior to simulation of the received data. The path to the file should be included if the file is not present in the default directory.

2.5 Resource Object

The Resource Object contains information about the configuration of the system and storage buffers. The form is given below (required attributes highlighted).

```
Resource =
    RcvBuffer =
        datatype          string      'int16' (default)
        rowsPerFrame      double      number of rows in frame
        colsPerFrame      double      number of columns in frame
        numFrames         double      number of frames (1 or even number)
        lastFrame         double      last frame transferred into buffer
    InterBuffer =
        datatype          string      'complex double(default)/single'
        rowsPerFrame      double      no. of rows (optional if PData defined)
        colsPerFrame      double      no. of cols (optional if PData defined)
        sectionsPerFrame  double      sections (optional: 1 or PData defined)
        pagesPerFrame     double      pages (optional: default=1)
        numFrames         double      number of frames
        firstFrame        double      1st frame processed of multi-frame buffer
        lastFrame         double      last frame processed for multi-frame bufr
    ImageBuffer =
        datatype          string      'double' (default)
        rowsPerFrame      double      no. of rows (optional if PData defined)
        colsPerFrame      double      no. of cols (optional if PData defined)
        sectionsPerFrame  double      sections (optional: 1 or PData defined)
        numFrames         double      number of frames
        firstFrame        double      first image frame of multi-frame buffer
        lastFrame         double      last frame of multi-frame buffer
    DisplayWindow =
        Type              string      'Verasonics' or 'Matlab'
        Title             string      'Window title'
        mode              double      2D=0 (default), 3D=1
        Orientation       string      'xz'(default),'yz', or 'xy'
        AxesUnits         string      'wavelengths'(default) or 'mm'
        Position          double[1x5] [left, bottom, width, height,[depth]]
        ReferencePt       double[1x3] upper lft corner(x,y,z) in trans coor. sys
        pdelta            double      pixel delta in wavelengths (default=0.25)
        Colormap          double[256x3] default Colormap is 'grey'
        numFrames         double      no. of frames in display cineloop (dflt 1)
        firstFrame        double      first frm in cineloop (system provided)
        lastFrame         double      last frm in cineloop (system provided)
        clrWindow         int         if !0, clrs the dsply wndw b4 nxt update
    Parameters =
        Connector (3.2+) double       connector # (1(dflt),2,or [n,m,...])
        connector (3.0x) double       connector # (0(both),1(dflt),2)
        numTransmit       double      number of available transmitters
        numRcvChannels    double      number of available receive channels
        speedOfSound      double      speed of sound in meters/sec (deflt 1540)
        speedCorrectionFactor double corrects speed of sound (default=1.0)
        startEvent        double     event no. to start at after freeze (dflt=1)
        simulateMode      double      0='use VDAS'(default), 1 = 'simulate'
        fakeScanhead      double      (dflt=0) if 1, allow running without probe
        GUI               string      'vsx_gui' [default] or 'vsx_guiOld'
        UpdateFunction    string      'VsUpdate' [default] or user specified
        verbose           double     0:3 int specifying level of warnings/errors
        initializeOnly    double      0(default), set to 1 to init. and exit
```

```
    waitForProcessing double    0(default), 1=synchronous (see 3.6.1.3)
    ProbeConnectorLED double         see User Manual for description
    ProbeThermistor double           see User Manual for description
    SystemLED      cellarray[1x4]    see User Manual for description

VDAS =
    numTransmit    double     hardware supported numTransmit
    numRcvChannels double     hardware supported numRcvChannels
    testPattern    double     if 1, use test pattern instead of A/D
    dmaTimeout     double     timeout for DMA trnsfrs in msec, dflt 1000
    sysClk         double     frequency of system clock
    halDebugLevel  double     value for enabling debug features
    watchDogTimeout double    timeout for refreshing watchDog timer
HIFU =
    externalHifuPwr double    use of External HIFU Option
    verbose         double    enable command line output
    extPwrComPortID string    ID of power supply control port
    voltageTrackP5  double    imaging profile tracks profile 5
    TXEventCheckFunction string name of limit check function
    extPwrConnection string   'series' or 'parallel'
SysConfig (read only) =
    System          string    type of system, e.g. "Vantage 256 UTA"
    Option          cell     array of strings identifying options present
    SWconfigFault   double    non-zero if SW configuration faults present
    HWconfigFault   double    non-zero if HW configuration faults present
                              empty if no hardware present.
    FPGAconfigFault double    non-zero if any FPGA revision incorrect,
                              empty if no hardware present.
    UTA             double    one of four possible values:
                              empty - no HW present, or SHI/baseboard
                              fault condition if HW system is present.
                              0 - HW system with no UTA baseboard or SHI.
                              1 - UTA baseboard module is present.
                              -1 - pre-UTA SHI module is present.
    UTAtype        1x3 double identifies the specific UTA adapter module
                              installed, or empty if not a UTA system
    UTAname        string     identifies the UTA adapter module installed
    SWversion      1x3 double identifies the system SW revision level
    HALversion     string     identifies the HAL revision level
    DriverVersion  sting      identifies the WinDriver revision level
    VDAS           double     non-zero if acquisition module(s) present.
    AcqSlots       1x4 double array with each entry set to one or zero
                              based on the presence / absence of an
                              acquisition module in the associated slot.
    TXindex        double     integer index value identifying the
                              transmit signal path configuration of the
                              acquisition modules.
    RXindex        double     integer index value identifying the receive
                              signal path configuration of the
                              acquisition modules
    simLicense,hwLicense,
    extendedTXLicense,arbTxLicense,
    arbToolboxLicense,triggerLicense,
    reconLicense   double     License status: 0 or 1
```

Before describing the `Resource` attributes more completely, a few words about the data flow in the Verasonics system are in order. The Verasonics system can best be understood as implementing data transformations from one memory buffer type to another. The order of data flow with the various processes is as follows:

Acquisition – RF data acquired in Vantage hardware modules' local channel memories are transferred to a RcvBuffer in the computer host's memory.

Pixel reconstruction – Processing takes input RF data from a RcvBuffer and output goes to InterBuffer (for I,Q pixels) and/or to ImageBuffer (for intensity pixels).

Processing - From one InterBuffer or ImageBuffer to another (or perhaps the same buffer).

Display – From one or more ImageBuffers to the DisplayWindow buffer.

To allow capturing RF ultrasound data at rates limited only by the speed of sound, digitized channel data is stored temporarily in high speed local memory on the acquisition modules of the system. After some amount of data has been acquired (typically all the acquisitions for a frame), the channel data is transferred to memory in the host computer. The memory buffer for receiving data transferred to the host computer is the `RcvBuffer`, which is organized into samples for each receive channel, for each frame. Multiple `RcvBuffers` can be allocated to hold different types of acquisition data, but in general, it is best to store all the acquisition data needed to process an image frame, such as 2D and Doppler acquisitions, in the same buffer. Multiple buffers are defined with an array index following the `RcvBuffer` – `Resource.RcvBuffer(m)`. A characteristic of the `RcvBuffer` (and indeed all buffer types) is that each defined buffer consists of frames that contain similar content and are equally sized in rows, columns (and perhaps sections and/or pages). The number of frames defined is generally dependent on the method of scanning and the number of frames desired for playback processing in an RF or image data cineloop.

The pixel based image reconstruction method works directly on data in one or more `RcvBuffers` to compute either baseband I,Q data, which is used for synthetic aperture and Doppler applications, or intensity data, which is used for normal 2D imaging. The result of these computations is stored in an InterBuffer (for complex data), or an ImageBuffer. It is important to remember that the output of the reconstruction consists of I,Q data or Intensity data at pixel points in the rectangular grid specified by the `PData` object, which are defined at a specific relationship to the transducer. There is no concept of a ray line, where the reconstruction output is a line of data points at some orientation with respect to the transducer; although for some Doppler applications, it may be desirable to define 'pixel' points along a steered line.

When additional processing is needed (such as computing Doppler frequency shift data from I,Q pixel data), an `Interbuffer` or `ImageBuffer` may be used as the source buffer, and an `ImageBuffer` used as destination. Again note that the processing is performed on the pixel points, and there is usually a one-to-one correspondence of the elements in the input and output buffer. Finally, when it is desired to write an image to the display, the data in an ImageBuffer are transformed into pixel data that ends up in a `DisplayWindow` buffer. The `DisplayWindow` buffer consists of a single frame of pixel data that is copied directly to the output window. The `DisplayWindow` typically has more pixels than specified in the corresponding ImageBuffer, since the pixels often need

to be interpolated up to a higher density to provide a useful image size on a high resolution display.

The various `Resource` buffer definitions create actual Matlab arrays which hold the respective data. The corresponding Matlab arrays created have the following names:

RcvBuffer(:) – Matlab cell array RcvData{n}(i,j,k), where n is the buffer number, i is the row index, j is the column index, and k is the frame no.

InterBuffer(:) – Matlab cell array IQData{n}(i,j,k,l,m), where n is the buffer number, i is the row index, j is the column index, k is the section number, l is the page number, and m is the frame number.

ImageBuffer(:) – Matlab cell arrays ImgData{n}(i,j,k,m) and ImgDataP{n}(i,j,k,m), where n is the buffer number, i is the row index, j is the column index, k is the section number, and m is the frame number.

2.5.1 RcvBuffer Attributes

The `RcvBuffer` receives the RF data transferred from the local memories on the VDAS modules. The organization of the buffer is shown in Fig. 2.6.1.1 below. A `RcvBuffer` can contain one frame or an even number of identical frames (in this example, 6 frames). Each frame consists of a number of columns that correspond to the number of channels used to acquire the frame. A channel is usually a physical acquisition channel, which is connected to a single element of the transducer for a transmit/receive acquisition event (the connection may change with subsequent acquisitions for multiplexed transducers). Down a column or channel, each row represents a set of samples of RF data, with time increasing down the column. If there are multiple transmit/receive acquisition events required to compose the frame, the samples for each event are stacked vertically down the column, as shown by the different colors in Fig. 2.6.1.1. In this case, 8 acquisition events are used for the frame. Acquisition events in a multiple acquisition frame can have different number of samples, and different sample mode attributes. The total number of rows in a RcvBuffer, N, must be greater than the total number of samples from all acquisitions (in this case, greater than 8*M). If the `RcvBuffer.rowsPerFrame` value exceeds 524288 samples, DMA buffer size restrictions on some compute platforms may come into effect.

The `RcvBuffer.lastFrame` attribute is set by the system when exiting a sequence running with the VDAS hardware. It is set to the last frame transferred into the buffer (note that this is not necessarily the last frame processed). This allows a separate processing function to ‘unwrap’ the RcvData array, so that the first frame is the oldest frame in the buffer.

The `RcvBuffer.datatype` specifies the bit size of the samples, and for the current Vantage hardware defaults to ‘`int16`’. This is currently the only datatype supported.

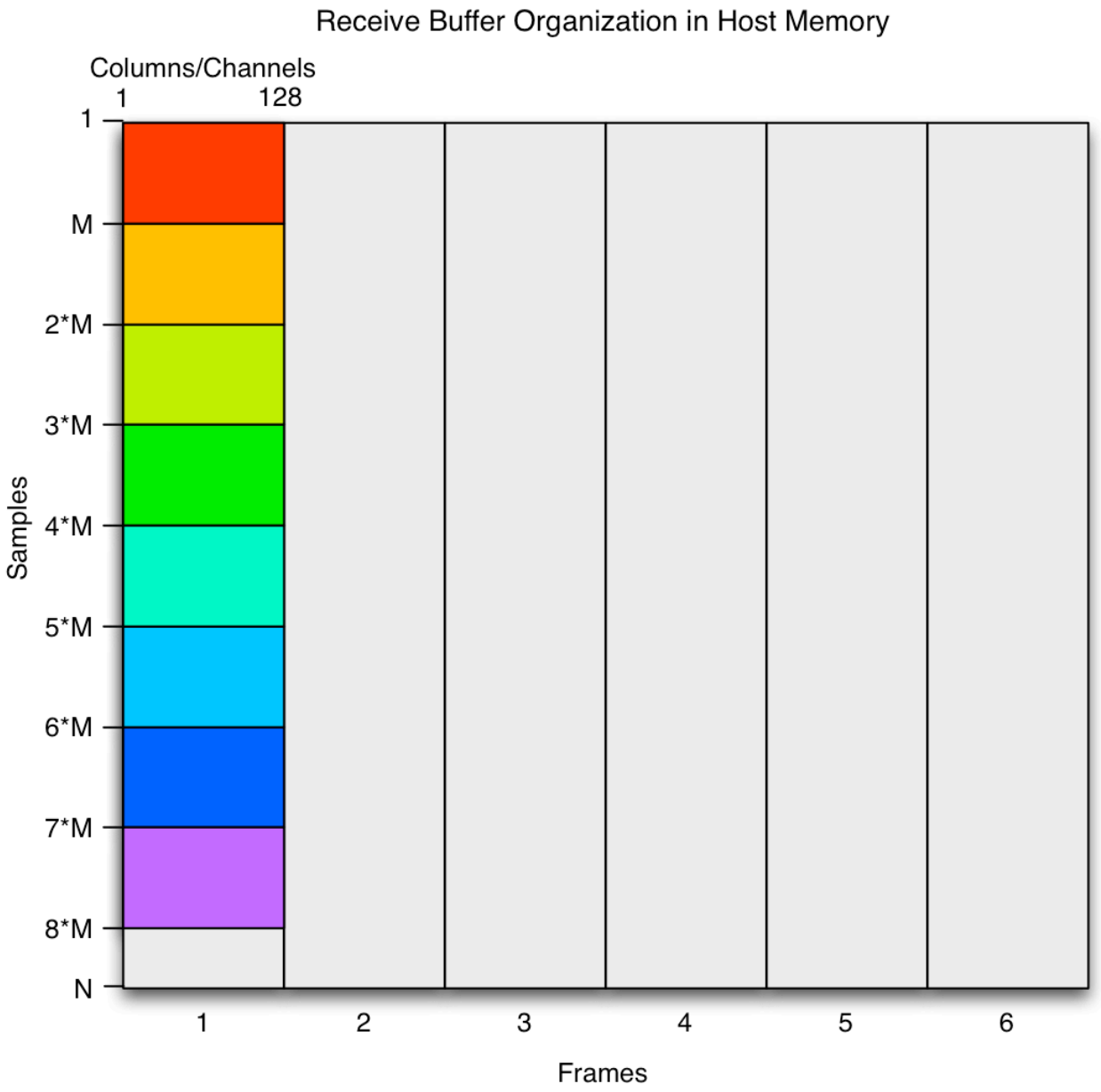


Fig. 2.6.1.1 A RcvBuffer with 6 frames and 8 acquisition events per frame.

2.5.2 InterBuffer Attributes

The `InterBuffer` receives the complex output of the pixel reconstruction processing, and consequently should always have a datatype of 'complex'. The datatype can be either 'complex double' or 'complex single'. Note that currently, the complex single datatype cannot be used with the internal image processing routines, and is meant for external processing functions that may execute on a GPU. The elements of the `InterBuffer` have a one-to-one correspondence to the elements of the `PData` (Pixel Data) array, which define the location of the reconstruction points relative to the transducer. The `InterBuffer` can have a larger size (more rows and/or columns) than the `PData` array, but not less. If more rows or columns are defined, the linear indices of the `InterBuffer` frame will align with the linear indices of the `PData` array. Normally, it is preferred to not specify the rows and columns of the `InterBuffer` frame in the

`Resource.InterBuffer` definition, in which case, these attributes will default to the same size as the `PData` array. In other words, if not defined:

```
Resource.InterBuffer.rowsPerFrame = PData.Size(1);
Resource.InterBuffer.colsPerFrame = PData.Size(2);
Resource.InterBuffer.sectionsPerFrame = PData.Size(3);  % for 3D scans
```

The `InterBuffer` can be defined with multiple frames, and within a frame, there can be two additional index dimensions beyond rows and columns. For 3D volume scans, the next dimension after the row and column dimensions is for sections. In 3D scans, a section is a xy slice through the scan volume which is typically parallel to the surface of the 2D transducer array at some distance, z, from the transducer. When there are multiple sections, they are typically spaced at the same spatial increment as the pixels within a section. The second additional index is the pages index. This index is typically used in Doppler acquisitions, such as for color flow imaging. The pages index would be used to store the results of the multiple Doppler acquisitions (typically referred to as an ensemble) used to acquire the Doppler signal for a frame or region of interest.

2.5.3 ImageBuffer Attributes

The `ImageBuffer` receives the intensity output of the pixel reconstruction method, but also can be the target output buffer for signal processing methods. The only datatype currently supported is 'double'. The number of columns in the `ImageBuffer` is always the same as the number of pixels on a row of the rectangular pixel grid defined, but the number of rows can be greater than the number of rows in the grid. If a `PData` array is defined for data in the `ImageBuffer`, the `rowsPerFrame`, `colsPerFrame` and possibly the `sectionsPerFrame` attributes can be omitted, and will default to the values in the `PData.Size` specification.

In the processing of an ImageBuffer to display image data, a secondary buffer of the same size as the `ImageBuffer` is created, called ImgDataP, which is used for spatial filtering and persistence processing. The ImageP buffer is then used to render pixels to the DisplayWindow. This allows the `ImageBuffer` (ImgData) to retain the original reconstructed output data for possible re-processing.

The `ImageBuffer` resource object contains `firstFrame` and `lastFrame` attributes which can are used for a multi-frame `ImageBuffer`. The `firstFrame` number (from 1) is the number of the oldest frame in the buffer, while the `lastFrame` number is the most recent. If the processing has filled the buffer, the `lastFrame` number will be one frame behind the `firstFrame` number, and both will advance with new frames, wrapping around the end of the buffer. Setting the `lastFrame` attribute to 0 indicates an empty buffer and triggers a clearing of the buffer before any render operation that writes to the buffer. When specifying a frame of the `ImageBuffer` to write to, a value of -1 is allowed, which indicates that the next frame from the `lastFrame` should be written to. Most processing routines will automatically increment the `lastFrame` and `firstFrame` pointers at the completion of the write. When specifying a frame of the `ImageBuffer` to read from, the value of -1 indicates that the most recent frame written to the buffer should be used. This would be the frame number of the `lastFrame` attribute.

2.5.4 DisplayWindow Attributes

The `DisplayWindow` structure contains the various attributes that are used by the system software to generate the output display window. There are two types of DisplayWindows available - 'Verasonics' and 'Matlab'. For `Resource.DisplayWindow.Type = 'Verasonics'`, a custom JAVA window is opened for display output. The JAVA window has most of the features of the 'Matlab' window, but offers a considerably higher frame rate (up to 60 fps). The 'Matlab' window is limited to 20 fps and often has a lag in updating so that the image is not synchronized with the current position of the transducer. Most example scripts are set to use the 'Verasonics' DisplayWindow.

The pixel resolution of the reconstructed image in an ImageBuffer is typically set to adequately sample the spatial resolution of the ultrasound scan, and not larger, as this would take more processing time for image reconstruction, and slow the frame rate. To render the reconstructed image to a high resolution display requires interpolating the pixels up to a higher density, according to the size of the image desired on the display. The DisplayWindow processing performs this interpolation, converting the reconstructed pixel resolution up to a new pixel delta, specified by `Resource.DisplayWindow.pdelta`. The DisplayWindow can also have a specified position on the computer's screen, and a different size and position relative to the transducer than the reconstruction window. The position and size of the window in pixels of the computer screen is given by `Resource.DisplayWindow.Position`, which has the values of [x, y, width, height], where x and y are the coordinates of the lower left corner. The position relative to the transducer is given by the attribute, `Resource.DisplayWindow.ReferencePt`, whose values are in wavelengths, and specify the location of the upper left corner of the DisplayWindow in the transducer coordinate system. The DisplayWindow region will display the portion of the `PData` array region that falls within the DisplayWindow.

The `Resource.DisplayWindow.mode` attribute will be needed for when a 3D display routine is integrated. For now it defaults to 2D. When scanning and displaying a 3D volume in 2D mode, the `Orientation` attribute determines how the DisplayWindow aligns with the `PData` volume region. For example, with `Orientation = 'xz'`, the DisplayWindow plane is parallel to the x,z plane. In this case, the ReferencePt y value determines the y location of the x,z plane. For a C-scan, the `Orientation` would be set to `'xy'`, and the `ReferencePt` z value would determine the depth of the plane in the z direction. The `ReferencePt` locations for the various `Orientation` choices are shown in the figure below. The `ReferencePt` is always in the most negative direction for the two axes and the row direction is always along the axis in the first position of the pair of axes.

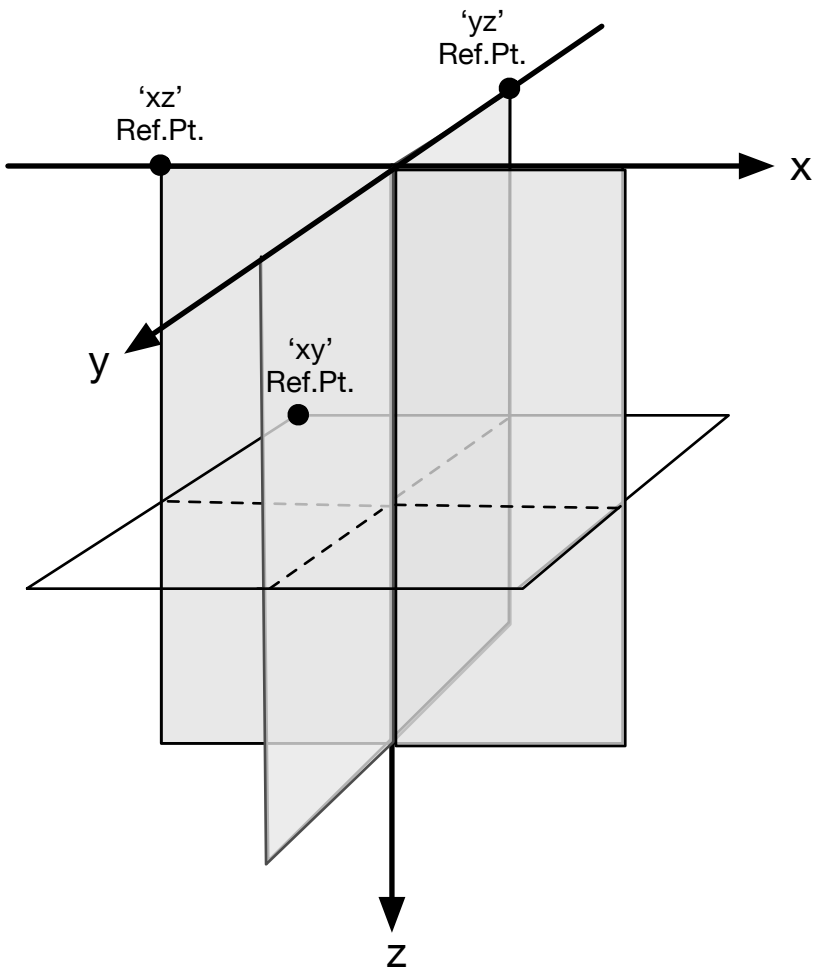


Fig. 2.6.4.1 DisplayWindow 2D planes for a 3D volume.

The slice of the 3D volume displayed is always the intersection of the DisplayWindow plane with the `PData` volume. Currently, the out of plane axis value will be used to find the nearest voxel plane in the PData 3D volume (no interpolation is done between voxel planes). It is the user's responsibility to insure that the DisplayWindow plane intersects with a portion of the PData volume that is being actively reconstructed, otherwise, no image data will be displayed.

The `Units` attribute specifies the units for the annotation on the DisplayWindow's horizontal and vertical axes ('wavelengths' or 'mm'). If not provided, the `Units` attribute is defaulted to 'wavelengths'.

The `numFrames` attribute specifies the number of frames in the DisplayWindow cineloop buffer. The frames stored in the cineloop are the same as the frames displayed on screen during runtime. In releases prior to 3.0, the cineloop frames were generated by reprocessing the frames in the ImageBuffer using the Process/imageDisplay attributes, and the cineloop number of frames was the same as the number of frames in the ImageBuffer. While this older method can still be used with some custom programming, the cineloop controls provided on the GUI window now control a history buffer of displayed frames, with a number of frames that is independent of the ImageBuffer number of frames. The number of frames in the InterBuffer and ImageBuffer can now

be set to only the number needed for double buffering or unique processing. At exit, the firstFrame and lastFrame attributes of the DisplayWindow are copied to the Matlab workspace so that the oldest and most recent frames displayed can be determined. A new runAcq command called 'cineDisplay' (see section 4.2), which has parameters of 'displayWindow' and 'frameNumber', can be used to display a particular frame from the cineloop buffer. This runAcq command can be called from a freeze state or external function. The cineloop UIControl works the same as before, and is rendered only when the numFrames defined for the DisplayWindow is greater than one.

VSX will now only create a DisplayWindow if a DisplayWindow structure is defined. This contrasts with previous behavior where if an ImageBuffer is defined, a default DisplayWindow is created. This allows a DisplayWindow to be created and used for custom data, if desired, or a custom display routine to be used with a user Imagebuffer.

The `Resource.DisplayWindow.Colormap` attribute specifies a standard colormap in the Matlab environment. The colormap is a 256x3 array of R,G,B values, which range from 0 to 1.0. The typical output image processing for intensity values from the image reconstruction is to compress the values using a power or log function, and display the compressed values using a linear grayscale map. Additional compression or expansion can then obtained by modifying the grayscale map, which can be performed by use of the colorMap tool or direct manipulation of the DisplayWindow's map using the menus provided by the Matlab toolbar.

2.5.5 Parameter attributes of Resource Object

These attributes were described previously in section 2.1.

2.5.6 VDAS attributes of Resource Object

The VDAS attributes of the Resource Object are generally automatically inserted by the VSX loader program. If the user provides an attribute, the loader program will not perform the automatic generation, and the user attribute will be used instead. This lets the user override attributes for use in complex scripts or for debug and testing. It is recommended that users consult with Verasonics support before modifying most Resource.VDAS attributes.

2.5.7 HIFU attributes of Resource Object

The HIFU attributes of the Resource Object are used by the system to manage operation using TPC profile 5 with either the Extended Burst Option or External HIFU Option installed on the system. For user scripts that do not exercise profile 5, the `Resource.HIFU` structure is not used and will not be created or parsed by the script loader program, VSX. Refer to section 3.2.4 "High Power Transmit (HIFU)" of this document for more information on these options and the use of the individual parameters in the `Resource.HIFU` structure.

3. Sequence Objects

The Sequence Object is simply the collection of objects used to define all of the events and attributes needed to acquire and process a sequence of ultrasound acquisitions.

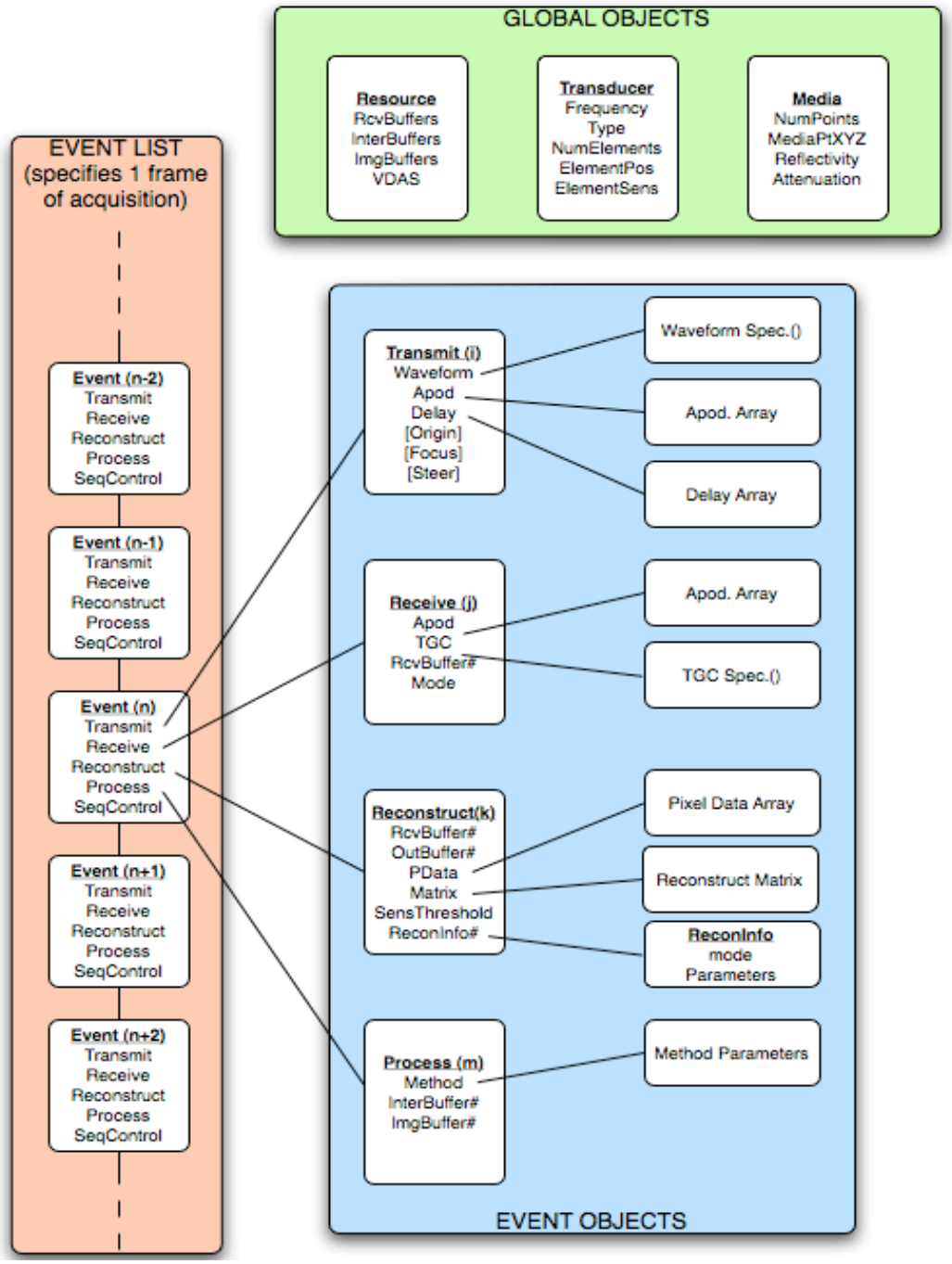


Fig. 1. An Event List and its associated Objects.

The Events of the Sequence Object are read sequentially to determine the actions to be carried out by the acquisition and processing system. Each `Event` contains other objects that describe in detail the various components of the acquisition and processing

of the ultrasound information. Typically, a sequence of events is generated to specify the acquisition of one or more frames of ultrasound data. The end of a sequence typically has a jump back to the first event, so that the sequence is repeated indefinitely.

The format of the Sequence Object container is the .mat file structure of the Matlab programming language. This format is a collection of Matlab structures and variables which are bundled together and saved as a binary file. The Matlab programming language is used to define the `Events` and their contained objects in an offline Setup script, which saves the information to a .mat file. This file is then loaded into the run-time programming environment by the loader program, VSX, which validates a script's structures and adds missing attributes. VSX then initiates the execution of the script in the hardware and software sequencers, performing hardware acquisition and processing, or acquisition simulation and processing.

The following sections provide a description of the various objects used in a Sequence Object and its associated `Event` List. The objects are defined as structures in the format of the Matlab programming language. As mentioned earlier, not all attributes of the various structures need be defined, as VSX will provide appropriate default values for missing attributes. For some structures, there are attributes starting with the characters 'VDAS----' that are only defined by VSX. These attributes are needed for programming the hardware when using the Vantage Research System for ultrasound acquisition, and are ignored when running in simulation mode. The attributes added by VSX can be examined after exiting a script that has been loaded and run. They can also be viewed after loading and without running a script if `Resource.Parameters.initializeOnly` is set to 1.

3.1 Event Objects

The sequence of events to be executed for acquiring one or more frames of ultrasound data are defined by a collection of Event Objects. Typically, the Event Objects are the last structures defined in a Setup file, but to define the objects used by the Events, one must have a plan of actions. It is convenient to first sketch out the sequence of actions in the Event list prior to defining the sequence objects referenced by the Events. The attributes of the `Event` Object are shown below.

```
Event =
    info:        string     'optional descriptive text for this event'
    tx:          double     # of TX structure array to use for transmit
    rcv:         double     # of Receive structure array to use for rcv.
    recon:       double     # of Recon structure array for reconstruct.
    process:     double     # of Process structure array to use.
    seqControl:  double     # of Sequence Control structure array to use
```

Except for the first attribute (info), all of the attributes specify the index of a particular structure array with the same name as the attribute, which should be previously defined (The software will check to see if the referenced structure array index is within the range of indices defined.). These structure arrays are objects that may contain additional object pointers or indices, and will be described fully in sections below. All of the above attributes are not needed for each event. For example, the event may only specify a reconstruction, in which case, the `Event.recon` attribute would have a Recon index, while `Event.tx`, `Event.rcv`, `Event.process` and the `Event.seqControl` structure numbers would be set to zero. (In Matlab, structure array indices start at 1, so an `Event` attribute index value of zero indicates that no operation is needed.) The `Process` structure array is used to specify buffer-to-buffer processes, and specifies any processing actions that are desired following the acquisition and/or reconstruction components of the event. The `Event.seqControl` attribute is used to specify Sequence Control actions, such as branching or pauses (which can be conditional, depending on control actions). This object can also contain hardware specific items that are used to control sequence flow, including interrupt actions to initiate data transfers.

Event structures define actions which typically execute sequentially. The first action is acquisition - a transmit followed by a receive period. The recon, or reconstruction action can operate on newly acquired data to produce an output, and the process action can operate on the reconstruction output. The seqControl action typically happens after other actions, although there are exceptions, such as the ‘pause’ for ‘extTrigger’ seqControl command, which will pause the sequence to wait for an external trigger before any actions are executed for the event in which it is specified. While it is often useful to combine multiple actions in the same event, it is not required; one can perform acquisitions (tx and rcv) without any reconstructions or processing. An `Event` can also be a transmit only event, with no rcv specified, but a rcv must always be accompanied by a tx reference. (A receive only event can be created by turning off all transmitters in the `TX.Apod` array of the referenced TX structure.)

To define a sequence of events, multiple `Event` structures are created, with ascending indices - `Event(1)`, `Event(2)`, `Event(3)`, ... `Event(n)`. The n Event structures are then sequentially loaded into the both the hardware and software sequencers for execution; or in simulate mode, sequentially read to determine the next simulator action. Additional information on the Event object can be found in Section 3.7.

3.2 Transmit Objects

This section describes the objects needed to specify the transmission of ultrasound pulses. There are three areas to consider when programming the system for transmit pulses:

1) The specification of a transmit waveform (type, frequency, duty cycle, duration, and polarity) for each of the transmitters available in the system, using the TW structure array.
2) The specification of the beam characteristics of each transmit action to be used in one's sequence, including which transmitters are active in the aperture (apodization), and the delay time before transmit for each active transmitter, using the TX structure array.
3) The specification of the transmit power level to use for a specific transmit event, using the High Voltage setting and the Transmit Power Controller (TPC) profile settings.

The transmitter for each channel of the Vantage system is designed to be as flexible as possible while also preserving excellent performance in terms of channel-to-channel uniformity, low phase noise, and high efficiency to enable high transmit power output levels. To meet these objectives, it uses a very simple "switching" design that can only do three things: drive the output to a symmetric positive or negative output voltage (set by the programmable HV transmit power supply in the system), or actively drive the output back to zero. To achieve good transient response and linearity while driving a reactive transducer load impedance, the output source impedance remains constant for all three output levels (i.e. the output does not just "turn off" when set to the zero state; the transitions at the leading and trailing edges of a transmit pulse will be very similar). This type of transmitter is commonly referred to as a "three level" or "trilevel" design.

Even though the output is restricted to only those three levels, the amount of time the transmitter remains at each level is user-programmable in units of the 250 MHz system clock (i.e. 4 nsec time resolution). Selecting which one of the other two levels to transition to at the end of an interval is also user programmable.

3.2.1 TW Objects

A transmit waveform for each transmitter in the system can be specified with the TW object. There can be multiple TW specifications defined by indexing the TW structure definitions. The only attribute that is always required in the TW structure is `TW.type`, to specify one of five waveform definition options, as shown below. If the waveform is to be generated by the Vantage hardware, the type must be specified as `'parametric'`, `'envelope'`, `'states'` or `'pulseCode'`. A simple single frequency burst can be defined with the `'parametric'` type, while waveforms that change in time require the `'envelope'`, `'states'` or `'pulseCode'` type.

```
TW(i) =
      type            string       'parametric', 'envelope', 'pulseCode',
                                   'states', 'function' or 'sampled'
```

For each of the supported TW.type values, additional TW attributes unique to that type are required to characterize it. These are listed in the subsections below for each individual type. For all TW types, either a single waveform can be defined that will then be applied to all transmit channels, or a unique waveform can be specified for each individual channel. By specifying per-channel waveforms with varying relative pulse widths from channel to channel, an amplitude weighting window function can be applied across the transmit aperture.

Additional TW attributes, as listed below, will be generated by the system software when it processes the user's script for execution either in simulation or on the HW system. These attributes should not be specified in the SetUp script, as their values will be overwritten by those produced by the system. They are listed below for reference.

```
TW(i).States          [nx2 double] Waveform definition as series of states.
TW(i).TriLvlWvfm      [mx1 double] Trilevel representation of TW waveform.
TW(i).Wvfm1Wy         [mx1 double] one way simulation wvform (sampled at
                                   250MHz)
TW(i).Wvfm2Wy         [mx1 double] two way simulation wvform (sampled at
                                   250MHz)
TW(i).peak            [double]     distance in wavelengths to peak of wave
TW(i).numsamples      [double]     no. of samples in Waveform
TW(i).estimatedAvgFreq [double]    average frequency in MHz of waveform
TW(i).Bdur            [double]     waveform overall duration in usec
TW(i).sysExtendBL     [double]     set to 0 for imaging waveforms less than
                                   25 cycles long; 1 for longer waveforms.
TW(i).CumOnTime       [double]     sum of the durations in usec of all
                                   active pulses in the waveform.
TW(i).Numpulses       [double]     total number of active pulses in the
                                   waveform.
TW(i).integralPkUsec [1x2 double] peaks of waveform integral (see text).
TW(i).fluxHVlimit     [double]     max allowed TX voltage for this waveform
TW(i).VDASArbwave     [uint16]     waveform table for programming the HW
```

`TW.States` is an array used to define the transmit waveform in a 'universal' format, for use by the system. Every user-provided `TW.type` waveform specification is translated into a `TW.States` array and added to the TW structure. `TW.States` is then used by the rest of the system, to program the transmitters for HW operation and to synthesize the transmit simulation waveform for operation in simulation mode.

`TW.TriLvlWfm` is trilevel representation of the transmit waveform, sampled in units of the 250Mhz clock.

`TW.Wvfm2Wy` is the transmit waveform used to generate the simulation waveforms. It is produced by passing the transmit trilevel wave through a second-order bandpass filter, with center frequency and bandwidth set by `Trans.frequency` and `Trans.bandwidth`. Note that `TW.Wvfm2Wy` will be based on a truncated copy of the actual transmit waveform representing only the first 20 cycles, and thus will not be representative of the actual transmit waveform if it is longer than that (this is intended to allow simulation to run, for debugging scripts using long HIFU or 'push' transmit waveforms but with the understanding that the actual simulation results will not be accurate).

`TW.peak` represents the time (in wavelengths of `Trans.frequency`) from the start of the waveform to the peak of the envelope representing instantaneous waveform

amplitude of TW.Waveform. TW.peak, when added to the transmit delay time, provides the effective time the transmit pulse leaves the transmitting element, which is needed to improve the accuracy of image reconstruction.

TW.estimatedAvgFreq is an estimated 'average' frequency over the entire waveform, obtained by counting polarity reversals in the waveform, adding one to get the the total number of half-cycles, dividing by 2 to get the total number of cycles, and dividing by total waveform duration to get average frequency in MHz.

TW.sysExtendBL is the logical flag used to inform the rest of the system whether TPC profile 5 must be used for this waveform, based on its duration.

TW.CumOnTime represents the cumulative duration in usec of all active pulses in the waveform; TW.Numpulses is simply a count of all active pulses in the waveform. These values are used by the TXEventCheck algorithms to evaluate safe operating limits for the transmit components during HIFU transmit events.

TW.integralPkUsec: The positive and negative peaks of the integral of the transmit waveform. These values represent the peak magnetic flux density that will occur in the transmit transformer over the course of the waveform; they are used to determine TW.fluxHVlimit, the limit that must be placed on transmit voltage to prevent the transformer from saturating.

TW.VDASArbwave: this is an array of uint16 values used to program the waveform generation FPGA logic in the HW system to synthesize the actual transmit output as defined by TW.States.

3.2.1.1 'parametric' Transmit Waveforms

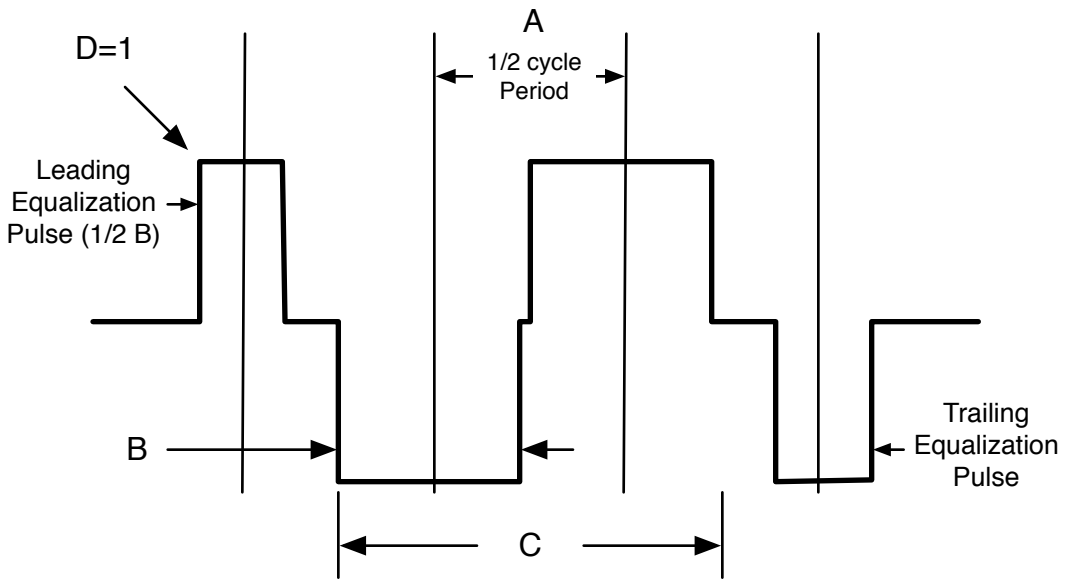
The parametric type defines a symmetric, periodic waveform of arbitrary duration. For this type, the waveform for each transmitter is defined by the four descriptors in the required TW.Parameters attribute, and the optional TW.equalize as listed below.

```
TW(i).Parameters     [Nx4 double] parametric parameters A,B,C,D (see text)
TW(i).equalize       [double]     0 or 1(default) equalization pulses
                                  off/on control (see text)
```

For this type If TW.Parameters is a 1 x 4 array, then the waveform defined by the four values as explained below will be applied to all active channels in the associated transmit event. Alternatively TW.Parameters can be defined as an N x 4 array, with each of the N rows specifying a unique waveform for each channel. In this case, the value of N must match the active aperture size being used by the transducer (typically the smaller of Resource.Parameters.numTransmit or Trans.numElements).

If we refer to the parameter values in a row of the TW.Parameters array as A,B,C, and D, the meaning of those four values is as follows:

A = Frequency of Transmit burst.
B = on time of half-cycle, expressed as fraction (range 0-1). (see note below)
C = no. of ½ cycle periods in waveform (not counting the optional equalization pulses).
D = sets the polarity of the first pulse of the waveform (-1 for negative, 1 for positive). All subsequent pulses always alternate in polarity from the first one.



The A Parameter is a scalar double representing the transmit center frequency over a range from 0.5 to 41.67 MHz. Note that the useful transmit frequency range may be less than this, depending on the frequency option purchased with the Vantage system. The actual transmit half-cycle period will be an integer number of 250 MHz clock periods, found by rounding (125 / A) to the nearest integer. Thus the precise center frequency may differ slightly from the value set by the user due to this rounding; the system SW will automatically modify the user-specified A value to the nearest settable frequency. The settable transmit frequencies supported by the Vantage system without the high frequency configuration option are shown in the table 3.2.1.1. Three addition transmit frequencies, 25, 31.25, 41.67 MHz, are supported and useful for the High Frequency Vantage system (see section 7.1.1).

The B parameter specifies the on time duration of the positive or negative drive during the ½ cycle period. During the remainder of the ½ cycle period, the transmit drivers output a zero level. This reduces the total energy transferred to the transducer, and can be used for transmit apodization or modifying the acoustic output for mode changes. The on time duration is expressed as a fraction of the ½ cycle period, with a value that ranges from 0 to 1.0. For example, a 50% on time is set with the value 0.5. [Note: The value zero should not be used to turn off a transmitter, since the minimum duty cycle is two clocks. Use the TX.Apod array to disable a transmitter.] For the Vantage High Frequency System transmits, there is a system constrain in the B parameter use - B can only be set to 1.0 for a transmit waveform at 41.67 MHz (see section 7.1.1).

On the V1 system, the use of pulse width scaling on a channel to channel basis results in a small distortion of the transmit focus, since on V1 the leading edge of the first active pulse of the waveform always starts at the end of the transmit delay- and thus the relative phase of the entire waveform shifts when the pulse width is changed. For Vantage we have eliminated this error by always starting the waveform at the leading edge of the first half-cycle regardless of the pulse width, and so if the relative pulse width is less than 1.0 the waveform generator will remain in the zero output state at the start of the waveform, until the leading edge of the first active pulse is reached.

The C parameter sets the pulse length by specifying the number of 1/2 cycle periods, and ranges from 0 to 1e07. When set to 0, the transmitter output is disabled. (If the

TX.Apod value of a transmitter is set to 0, the C parameter is automatically set to 0, regardless of the value in the TW.Parameters array.) For pulse durations longer than about 32 half cycles, the system must have the high power option installed, and the transmit waveform generated with transmit profile 5. Other hardware restrictions and heat dissipation may limit the duration of transmits, and safe operating conditions for the transmitter circuitry will be enforced by the software.

The D parameter is used to specify the polarity of the first half cycle transition, and is either 1 (first transition positive) or -1 (first transition negative). Note that the first transition with TW.equalize = 1 (the default) is that of the first half-cycle and not the equalization pulse.

Supported Transmit Frequencies			
20.8333	2.3148	1.2255	0.8333
17.8571	2.2727	1.2136	0.8278
15.6250	2.2321	1.2019	0.8224
13.8889	2.1930	1.1905	0.8170
12.5000	2.1552	1.1792	0.8117
11.3636	2.1186	1.1682	0.8065
10.4167	2.0833	1.1574	0.8013
9.6154	2.0492	1.1468	0.7962
8.9286	2.0161	1.1364	0.7911
8.3333	1.9841	1.1261	0.7862
7.8125	1.9531	1.1161	0.7812
7.3529	1.9231	1.1062	0.7764
6.9444	1.8939	1.0965	0.7716
6.5789	1.8657	1.087	0.7669
6.2500	1.8382	1.0776	0.7622
5.9524	1.8116	1.0684	0.7576
5.6818	1.7857	1.0593	0.7530
5.4348	1.7606	1.0504	0.7485
5.2083	1.7361	1.0417	0.7440
5.0000	1.7123	1.0331	0.7396
4.8077	1.6892	1.0246	0.7353
4.6296	1.6667	1.0163	0.7310
4.4643	1.6447	1.0081	0.7267
4.3103	1.6234	1.0000	0.7225
4.1667	1.6026	0.9921	0.7184
4.0323	1.5823	0.9843	0.7143
3.9062	1.5625	0.9766	0.7102
3.7879	1.5432	0.9690	0.7062
3.6765	1.5244	0.9615	0.7022
3.5714	1.5060	0.9542	0.6983
3.4722	1.4881	0.947	0.6944
3.3784	1.4706	0.9398	0.6906
3.2895	1.4535	0.9328	0.6868
3.2051	1.4368	0.9259	0.6831
3.1250	1.4205	0.9191	0.6793
3.0488	1.4045	0.9124	0.6757
2.9762	1.3889	0.9058	0.6720
2.9070	1.3736	0.8993	0.6684
2.8409	1.3587	0.8929	0.6649
2.7778	1.3441	0.8865	0.6614
2.7174	1.3298	0.8803	0.6579
2.6596	1.3158	0.8741	0.6545
2.6042	1.3021	0.8681	0.6510
2.5510	1.2887	0.8621	0.6477
2.5000	1.2755	0.8562	0.6443
2.4510	1.2626	0.8503	0.6410
2.4038	1.2500	0.8446	0.6378
2.3585	1.2376	0.8389	0.6345

Table 3.2.1.1 Vantage transmit frequencies supported with 250MHz clock (standard system). The values can be computed using the following Matlab statements:

```
A = (6:197)'; TxFreq = 250 ./ (2.*A); [A, TxFreq]
```

The TW.equalize attribute is optionally used to control the equalization pulses as shown in the sketch. If TW.equalize is not specified, the system will assign an automatic default value of 1. There are three allowed values of TW.equalize:

> TW.equalize = 0 disables the equalization pulses. In this case the value of C sets the overall period of the waveform, and thus C should be even to produce a waveform with no DC content.
>
> TW.equalize = 1 appends an equalization pulse at the beginning and end of the waveform as shown in the sketch. The duration of the equalization pulses is 1/2 that of the full pulses as set by B. If C is odd, both equalization pulses will have the same polarity and thus the overall waveform will have zero DC content for any value of C. The time from the center of each equalization pulse to the center of the adjacent full pulse will be set to A, the half-cycle period of the overall waveform as illustrated in the sketch.
>
> TW.equalize = 2 appends equalization pulses in the same manner as described above when TW.equalize = 1, except that in this case the equalization pulses will be shifted closer to the adjacent full pulses such that the “off time” between them will match the off time between full pulses, i.e. (A - B).

Disabling the equalization pulse is not recommended, as this could lead to a transmit waveform with a DC bias. A transmit pulse with a DC component can result in a longer recovery period for the Time Gain Receivers following transmit, effectively obscuring echoes near the transducer.

For parametric waveforms, the typical usage is to define a single waveform in the setup script, which will be copied by VSX into all of the available transmitter specifications. A separate parametric waveform can be defined for each transmitter if desired, but transmitting with different frequencies or pulse durations on different transducer elements can result in incoherent transmit wavefronts. Typically, the only parameter that one would consider changing over the range of transmitters is the B parameter, which can be used for transmit apodization. By reducing the half cycle pulse width for the outer transmitters in the aperture, the power output is reduced for these elements. As we shall see in the section below, the `TX.Apod` array can be used to perform this function for the ‘parametric’ waveform. A `TX.Apod` array value can range from 0.0 to 1.0, where 0 turns a transmitter off, regardless of the `TW.Parameters` set. For any non-zero `TX.Apod` array value, the associated `TW.Parameters` A, C and D values will be used, and the associated B value will be multiplied by the `TX.Apod` value. It should be noted that the `TW.Parameters` row index corresponds one-to-one with connector I/0 channel numbers, in contrast to the `TX.Apod` array indices, which correspond with the elements in the logical aperture of the transducer. This means that the weight in the nth index of `TX.Apod` may not modify the B parameter for the nth row of `TW.Parameters`. (Yet another reason for only defining a single row for `TW.Parameters` and letting the system perform any transmit apodization.)

When specifying the parametric waveform type, the only TW attributes that need be set are the `TW.type` and `TW.Parameters` (and possibly the optional parameters `equalize` and/or `extendBL`). The other attributes are then set by calling the utility function, ‘computeTWWaveform.m’, which also computes a simulation waveform in the array

`TW.Wvfm2Wy64Fc` from the parametric values. In fact, the sequence loader program, VSX, will call computeTWWaveform automatically, if no `TW.Wvfm2Wy64Fc` or `TW.peak` attributes exist.

'parametric' Transmit Waveforms, in V1 backward compatibility mode -

For backwards compatibility with scripts used on V1 systems, the TW `'parametric'` waveform type can also use A and B parameters that specify a number of transmit clock periods. If the software sees an integer value for the B parameter, it interprets the A and B parameters as a number of transmit clock periods, with a default transmit clock of 180MHz. Other values of the transmit clock can be set, using the attribute

```
TW(i).twClock   [double]   xmit master clk {45,90,180MHz(default)}
```

The interpretation and allowed ranges of the A, B, C, D values are summarized below:

A: period of one half-cycle of the transmit waveform, in units of integer periods of the transmit clock frequency set by TW.twClock. The allowed range for A is 5 to 63 clock periods.

B: period of the active transmit pulse during each half-cycle of the transmit waveform, in units of integer periods of the transmit clock frequency set by TW.twClock. The allowed range for B is 2 to A clock periods; if A is greater than 31 then B is scaled by 2 and thus the B variable value never exceeds 31.

C: waveform duration, specified as the integer number of half-cycles that make up the waveform. Allowed range is 0:31. A value of zero disables the transmit output completely. For longer duration transmits that are allowed with the high power option, the following TW attribute must be set:

```
TW(i).extendBL  [double]  0(default) or 1=multiply burst duration by 64
```

D: sets the polarity of the first active pulse in the waveform. All subsequent pulses automatically alternate in sign from this starting point. Allowed values are only 1 and -1.

When the Vantage system responds to the `'parametric' TW.type` in V1 backward compatibility mode, the V1-format A and B parameter values will be rescaled as needed to produce an equivalent waveform using the actual Vantage clock rate of 250 MHz. Thus the precise transmit carrier frequency and pulse duration will in many cases not be the same as on V1, but the difference will typically be only a few percent. The precise frequency resulting from this conversion will be listed in the TW structure using the new variable TW.frequency, a scalar double representing the transmit center frequency in MHz. By preserving the V1 constraint that every half-cycle has the same duration, the allowed carrier frequencies are restricted to integer submultiples of 125 MHz- but the benefit of this constraint is that the waveform has half-wave symmetry and thus has zero even harmonic content.

3.2.1.2 ‘envelope’ Transmit Waveforms

The ‘envelope’ transmit waveform type is a new type for Vantage systems that allows generation of a periodic waveform with time-varying amplitude modulation implemented with pulse wave modulation or a time-varying frequency to generate a ‘chirp’ waveform.

The envelope waveform definition requires three variables to specify the waveform:

`TW.envNumCycles` is the duration of the overall waveform in whole cycles. Each individual cycle of the waveform will be a symmetric pair of positive and negative half-cycle pulses of the exact same duration, ensuring no DC or even harmonic content in the resulting waveform. `TW.envNumCycles` is a scalar double which must have an integer value of at least 1 and no greater than 10,000.

`TW.envFrequency` is a row-vector that must be of length equal to `TW.envNumCycles`. Each entry is a double representing the frequency for the associated cycle in MHz, with an allowed range of 0.5 to 32 MHz. The frequency value will be modified to represent the nearest integer number of 250 MHz clock periods for the associated waveform.

`TW.envPulseWidth` is a row-vector that must be of length equal to `TW.envNumCycles`. Each entry is a double representing the relative pulse width for both half-cycles of the associated transmit cycle. The allowed range of each entry is from 0.0 to 1.0. The actual pulse width will be rounded to the nearest integer number of 250 MHz clock periods, based on the specified relative pulse width and frequency for that cycle. A negative value can be specified to invert the polarity of the associated cycle (the first half-cycle will be positive if the `TW.envPulseWidth` value is positive, or negative if the `TW.envPulseWidth` value is negative).

3.2.1.3 'pulseCode' Transmit Waveforms

The 'pulseCode' type allows the user to explicitly define any arbitrary transmit waveform by specifying the `TW.PulseCode` array directly. This is a new variable in the Vantage TW structure, used to provide a universal mechanism for specifying the transmit waveform. For this type, `TW.PulseCode` is the only required input variable, and will be used as-is by the system.

The `TW.PulseCode` array is generated by the `computeTWWaveform` function in response to any `TW.type` definition that was provided by the user. `TW.PulseCode` is then used by the update and arbwaveCompile functions to produce the actual waveform descriptor table that will be supplied to the HW, and is also used by computeTWWaveform to generate the `TW.Wvfm2Wy64Fc` array used for simulation. `TW.PulseCode` is also intended to provide a readily accessible and straightforward format that can be easily used as input to other waveform analysis functions as desired by the user.

`TW.PulseCode` is a Matlab array of double-precision values, of size (N x 5) for a single waveform definition that will be applied to all channels, or optionally of size (N x 5 x numTXchannels) for a unique waveform definition for each channel. Each of the N rows in the array specifies one active transmit 'pattern' or waveform segment. The five values within the row that define each segment, referred to here as [**Z1, P1, Z2, P2, R**], are defined as follows:

Z1 is a non-negative integer value (range 0-1023) representing the number of 250 MHz clock periods for which the waveform is held in the zero state prior to active pulse **P1**.

P1 is a signed integer (range 0-175) defining an active output pulse, following the **Z1** zero-state interval. The polarity of the pulse is set by the sign of **P1**, and the duration in 250 MHz clock periods is set by the absolute value of **P1**.

Z2 defines the duration of the zero-output state of the waveform between active pulses **P1** and **P2**, using the same specification given above for **Z1**.

P2 defines an active output pulse following **Z2**, using the same specification given above for **P1**.

R is a positive integer (range 1-10e07) specifying the number of times the waveform segment defined by [**Z1 P1 Z2 P2**] is to be repeated. Setting **R** to zero terminates the waveform output with [**Z1 P1 Z2 P2**] for that row being discarded.

Note that any **Z** or **P** value can be set to zero, resulting in a zero duration for that element of the waveform. If the **R** value is set to zero, that signifies the end of the waveform and all subsequent entries in the `TW.PulseCode` array will be ignored. If the **R** value in the first row of the array is zero, then a waveform that produces no output at all has been defined. In any row that terminates the waveform, the value of **R** must be zero. The **R** set to zero waveform termination row can be omitted if it would be the last row of the definition array. **P1** or **P2** values of zero produce no waveform output (useful for equalization pulse isolation and one of a kind waveform patterns within repeating segments).

3.2.1.4 Simulation Transmit Waveforms

When running a sequence using simulate mode (Resource.Parameters.simulateMode = 1), the additional waveform types of 'function' and 'sampled' are allowed in addition to the standard types of 'parametric', 'pulseCode', and 'envelope'. When running with the Verasonics hardware, the standard types are all specified by `TW.PulseCode`, which is added by the `computeTWWaveform` function (called by VSX) if not present in the user's TW definition. If a simulation waveform is specified, no PulseCode is generated, and an error of 'missing PulseCode' will be output if the user attempts to run their script on the hardware.

There are some limitations on the length of the transmit waveform in simulate mode, based on the number of cycles and the total duration. Typically, a simulation waveform should be limited to 25 cycles or less.

For the ***'function'*** waveform type, the waveform is a windowed sinusoid that can be specified by the frequency and the window descriptors. The following TW attributes should be defined before the `computeTWWaveform` utility function is called to generate the `TW.Wvfm2Wy` and `TW.peak` data:

```
TW.type = 'function';
TW.frequency = f;  % frequency in MHz
TW.Descriptors = [A,B,C,D]; % Window descriptors and polarity
TW.numsamples = n; % number of samples at 250MHz sample rate
```

The waveform is calculated from the statements below:

```
f = TW.frequency;
n = TW.numsamples;
T = (0:n-1)';
TW.Wvfm2Wy(1:n) = D*Window.*sin(2*pi*(T/250)*f);
TW.peak = (n/2 * 1/250)*Trans.frequency; % in wavelengths of Trans.freq
```

where `D` is the polarity descriptor (+1 or -1), and `Window` is a generalized cosine window defined by:

```
Window = A – B*cos(T*2*pi/n) + C*cos(2*T*2*pi/n);
```

The `TW.Wvfm2Wy` waveform is sampled at 250MHz, as required by the simulation code. In this case, the pulse length is set by the `TW.numsamples` attribute.

Popular window functions are Hamming (A=0.54, B=0.46, C=0), Hann (A=0.5, B=0.5, C=0), and Blackman (A=0.42, B=0.5, C=0.08). The amplitude value of 1.0 is somewhat arbitrary for the simulator, but if 16 bit RF data is used, the value should not be higher, due to the possibility of saturation.

For the 'function' waveform type used with simulation mode, only one set of descriptors is required, as the current version of the simulation doesn't support different frequency or burst length specifications for individual transmitters. The `TX.Apod` array in this case can provide an apodization function by using weighting values for individual transmitters ranging from 0 to 1.0.

Finally, the ***'sampled'*** waveform type can also be used by the Simulator. In this case, the following attributes should be defined by the user:

```
TW.type = 'sampled';
TW.frequency = f;  % frequency in MHz (used only for computing attenuation)
TW.numsamples = n; % number of samples at 250MHz sample rate
TW.Waveform = (waveform of n samples, sampled at 250MHz);
TW.peak = w; % wavelengths of Trans.frequency to peak of waveform
```

The above attributes must be computed by the user in the Setup script. The provided `TW.Waveform` should contain the round trip waveform, sampled at 250MHz as required by the simulation code. The amplitude of the provided waveform should be approximately 1.0 (+/- 1.0). The `TW.Wvfm2Wy` attribute, which is used by the simulation, can be provided by the user or by calling the `computeTWWaveform` utility, which will simply set this waveform equal to the user's `TW.Waveform`.

For all waveform types, the `TW.peak` attribute provides the distance in wavelengths of `Trans.frequency` from the start of the transmit waveform to the peak amplitude point. This is needed to specify the effective time that the transmit pulse leaves a transmitting element, which is given by adding `TW.peak` to the delay value (in wavelengths) for an element. For symmetrical transmit waveforms, `TW.peak` is generally equal to ½ the burst length, in units of wavelengths, as specified by `1/Trans.frequency`.

3.2.2 TX Object

The TX object contains all the information needed to define the transmit waveforms and timing for each of the active elements in the transducer aperture. In the non-multiplexed transducer case, it is assumed that the number of transmitters, M, is greater than or equal to the number of transducer elements, N, and that the transducer elements are connected to the transmitters on a one-to-one basis (element number one is connected to transmitter number one, etc.). For the case where the elements are not wired one-to-one with the connector I/O channels, the `Trans.Connector` array specifies the mapping, allowing the element numbers to be considered as mapped one-to-one with transmitter numbers. The transmit period begins at the start of an acquisition event, which represents time 0. The individual transmitters begin counting down their delay times, and at the end of the delay, the transmit waveform is generated.

The main attributes of the TX object are shown below. The highlighted attributes are required before calling the utility function, 'computeTXDelays' to calculate the transmit delay values.

```
TX =
      waveform:   double           number of a transmit waveform structure.
      Apod        [1xM double]     (see notes)
      aperture    double           (see notes - only needed for muxed xducers)
      Origin      [1x3 double]     effective origin of transmit beam at xducer
      focus       double           Focal distance in wavelengths (0=flat).
      Steer       [1x2 double]     Beam steering angles in radians (θ,α).
      FocalPt     [1x3 double]     optional method of defining focus(see text)
      FocalPtMm   [1x3 double]     FocalPt in mm instead of wavelengths
      Delay       [1xM double]     delay times for xmitters in wavelengths
      TXPD        [PData.Sizex3]   transmit pixel data
      peakCutOff  double           used to qualify pixel intensities dflt=1.25
      peakBLMax   double           used to qualify pixel burst lengths

      VDASApod    [1xN double]     translated TX.Apod array for hardware
      VDASDelay   [1xN double]     translated TX.Delay array for hardware
```

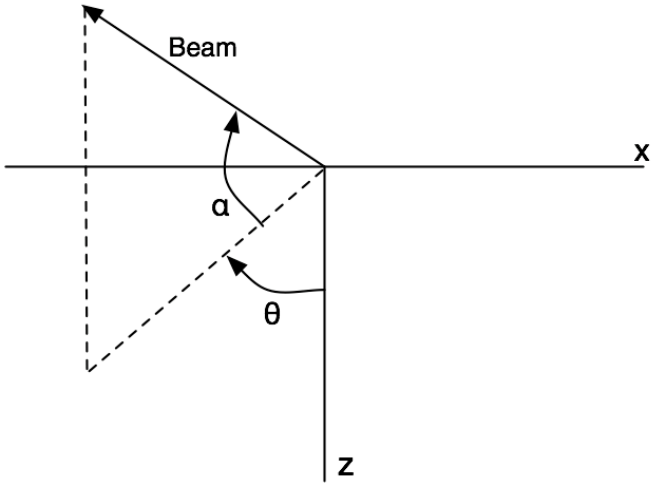
The `TX.waveform` attribute is the index into an array of TW objects (see previous section) that define the transmit waveforms to be generated by each of the active transmitters. The transmit waveform is typically a few cycles of the transducer element center frequency that are applied to the element after the delay period.

If not all transmitters are needed for a transmit event, the `TX.Apod` array can be used to select a subset of the available transmitters, by specifying a zero value for the transmitters not used. An apodization weighting function across the transmit aperture can also be specified, by using `TX.Apod` values that range from -1.0 to 1.0, with negative values inverting the waveform polarity. When using the VDAS hardware, the `TX.Apod` array is used both as an on/off indicator, with 0 values turning off a transmitter, and non-zero values enabling it, and for transmit apodization using transmit waveform duty cycle control. When a `TX.Apod` array has absolute values in between 0.0 and 1.0, the transmit waveform is modified using pulse width modulation to reduce the effect power transmitted. For example, with the parametric waveform, the `TW.Parameters` B values are modified when loaded into the hardware to set the nearest settable ½ cycle duty cycle. Since the minimum B value is 2 transmit clocks,

setting `TX.Apod` values smaller than 2/T, where T is the number of transmit clocks in the half cycle on time of the transmit waveform, will have no effect on the minimum apodization. If a TW structure defines an array of per-channel waveforms, then those waveforms will be used as-is and their relative pulse widths will not be scaled by `TX.Apod`. Instead, `TX.Apod` will be interpreted as a logical value only- zero will disable a channel's transmit output but any non-zero value will result in using that channels `TW.PulseCode` exactly as-is.

The `TX.aperture` attribute is only required for transducers that incorporate high voltage multiplexers in the probe body that are used to select a sub-aperture of the full array aperture. The value of `TX.aperture` specifies an aperture defined in the `Trans.HVMux.Aperture` array. Use of the `TX.aperture` attribute without a corresponding `Trans.HVMux.Aperture` definition will result in an error when the script is loaded by the loader program, VSX. The value of `TX.aperture` refers to the second index of the `Trans.HVMux.Aperture` array. It should also be noted that when a sub-aperture is selected using the `TX.aperture` attribute, the `TX.Apod` values apply linearly to the elements of the sub-aperture, even though the actual transducer elements can change with each `TX.aperture` selection.

For typical transmit beams, the attributes `Origin`, `focus` and `Steer` are sufficient to define the beam characteristics. The Origin is required for focused beams (`TX.focus > 0`), and virtual apex beams (`TX.focus < 0`) and specifies the point on the transducer element surface that the beam appears to originate from. For linear arrays, this is typically where the center of the transmit beam intersects the x, z plane of the transducer elements, which is usually the center of the transmit aperture for a 1D array (except for the case where the transmit beam is near the outer edges of the transducer aperture). The focus distance is the distance (in wavelengths) from the Origin point on the transducer to where the beam comes to a focus. A positive focus value defines the point of convergence for a multi-element transmit beam. A focus value of 0 specifies a flat wavefront across the transmit aperture, while a negative focus value specifies a diverging beam whose effective source is along the –z axis at the negative focal value (for no steering). The Steering angle(s) specify the angle of the beam with the transducer plane at the point of origin. θ specifies the angle of the beam projection into the x,z plane from the positive z axis (azimuth), and α specifies the angle of the beam with respect to the x,z plane (elevation).



If `TX.Apod, TX.focus` and `TX.Steer` are given (and also `TX.Origin` for `TX.focus~=0`), the method '*computeTXDelays*' can be called to compute the individual transmit delays. This function also requires access to `Trans`, the global object specifying transducer geometry (defined earlier). The method computes `TX.Delay` values for all elements with a non-zero `TX.Apod` value.

To re-iterate, if `TX.focus` is set to 0, this implies a flat focus beam, where for a linear array and zero degrees steering, all transmitters would start at the same time. In this case, the origin of the beam can be set to the center of the aperture. A non-zero steering angle would add a linear slope to the transmit delays. The *'computeTXDelays'* method will also work with a virtual apex type of transmit function if the origin point on the transducer and steering angles are properly specified. Another type of transmit pattern is a diverging beam for flash transmit imaging, and this can be specified by using a negative value for `TX.focus`.

For some applications, it may be more convenient to define the transmit focus at a fixed location. In these cases, the TX.FocalPt attribute can be used instead of TX.Focus, TX.Steer, and TX.Origin when calling the `computeTXDelays` function. The TX.FocalPt attribute specifies a fixed coordinate, (x,y,z) where the transmit energy will be focused. If the TX.FocalPt attribute is missing or empty, the TX.focus attributes will be used, but if present, the TX.focus, TX.Steer and TX.Origin attributes will be ignored.

For all other transmit delay patterns, the delays must be computed by the user and written to the `TX.Delay` array in wavelengths. The maximum value of a `TX.Delay` entry, when converted to microseconds, is 45.5 usec. For example, a 5 MHz center frequency transducer has a wavelength period of 200 nsec. The maximum wavelength setting for TX.Delay in this case would be 45.5e-06/200e-09 = 227.5 wavelengths. In addition, the effective origin point of the transmit beam at the transducer face must also be computed. In the case of an arbitrary delay pattern where the *'computeTXDelays'* method is not used, the attributes of `TX.focus` and `TX.Steer` are ignored.

3.2.2.1 TX.TXPD Data

The TXPD attribute of the TX object holds information on the simulated transmit field at the pixel locations defined in a `PData` array. A function is provided for computing the field parameters called computeTXPD. Its usage is as follows:

```
TX.TXPD = computeTXPD(TX, PData);
```

The output is an array of uint16 values with three indices for 2D and four for 3D volumes. The first two or three indices are the same as the first two or three indices of PData.Size, while the last index can be either 1, 2 or 3, and specifies the parameter calculated. The parameters calculated are 1) the peak transmit intensity at the pixel location (in single channel transmit equivalents); 2) the time that the peak transmit occurred at pixel location (in equivalent wavelengths); and 3) the length of the transmit burst, as seen at the pixel location (in wavelengths).

If added to a TX structure, a reconstruction that references the TX structure (through the ReconInfo.txnum attribute) will use the TXPD data in processing the reconstruction, and perform the follow:

1) The reconstruction output intensity will be normalized at each pixel location according to the peak transmit intensity at the pixel.
2) The time of travel from the origin of the transmit burst to the time of peak intensity at the pixel location will use the TXPD calculation instead of an algorithmic approximation.
3) Only pixels with a peak intensity greater than `TX.peakCutOff` (in transmit equivalents) and a burst length at the pixel location less than `TX.peakBLMax` will be reconstructed. These new TX attributes default to 1.25 and (the minimum transmit burst length + 0.25 (wavelengths)), respectively, if not provided.

The TXPD attributes can be added to facilitate image reconstruction for certain types of scanning sequences, particularly those that involve multiple overlapping transmit beams. In these sequences, the pixels within the overlapping beams may be reconstructed multiple times, namely once for each transmit/receive event that insonifies the pixel. The multiple reconstructions can be combined, with each pixel reconstruction normalized by its transmit beam intensity, to produce a synthetic transmit focal point at each pixel. More information will be provided in section 3.4, covering image reconstruction.

'showTXPD' - This function will automatically add the TXPD attribute to the TX structures in the workplace, and bring up a window to visualize the fields. After executing an example script to calculate the script's structures in the Matlab workplace (such as the 'SetUpL7_4FlashAngles.m) script, one can then type

```
>>showTXPD
```

on the Matlab command line. The transmit fields will be computed and the figure window shown below will be created.

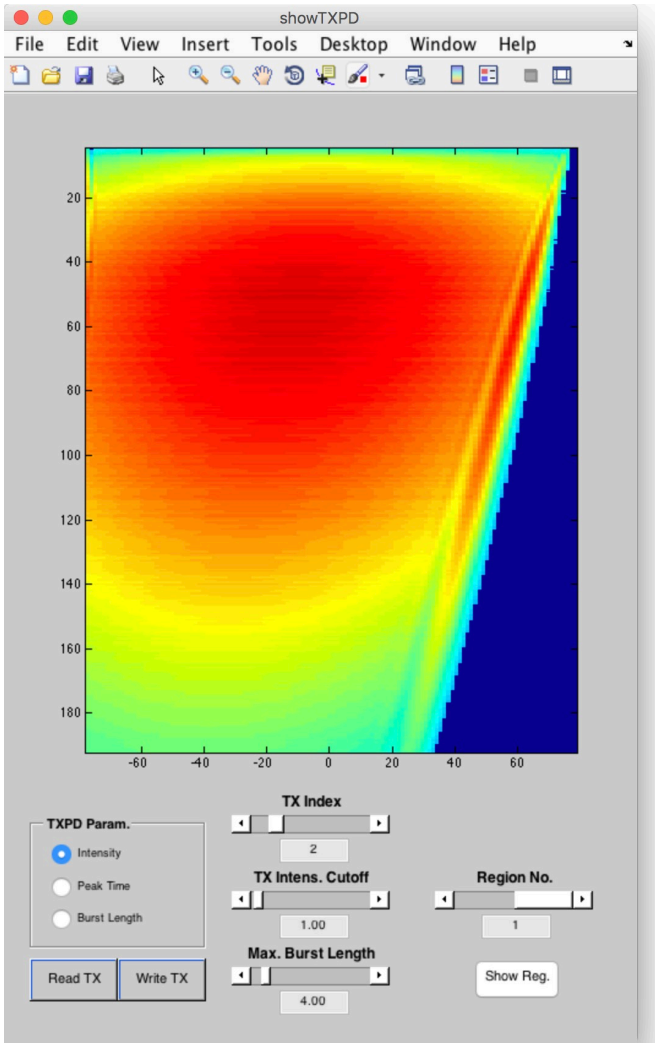


Fig. 3.2.2.1.1 Using showTXPD to visualize the transmit field for TX.

3.2.2.2 TX VDAS parameters

The `TX.VDASApod` and `TX.VDASDelay` attributes are set by the loader function, VSX, or the processing function, runAcq, and do not need to be set by the user. They are used to map the logical user TX attributes into the hardware specific parameters needed to program the VDAS system. For example, the `TX.Apod` array values apply to the linear range of elements in the transducer aperture, regardless of how that aperture is mapped to connector I/O channels. The `TX.VDASApod` array values are derived from the `TX.Apod` values, but permuted to have a direct one-to-one relationship with the I/O channels.

The VDAS parameters are made visible in the structure definitions as an aid in debugging and for use in custom configurations of the hardware.

3.2.3 Transmit Power Controller (TPC)

The Transmit Power Controller, or TPC provides a high voltage supply for the transmitters on the VDAS modules. Its range is 1 - 96 volts, which is settable using the High Voltage slider object on the VSX GUI window. Since the transmitters are bi-polar output devices, the actual transmitted peak-to-peak voltage is twice the level set. The high voltage level can also be set by typing a value in the text box below the slider. For safety reasons, the high voltage level is always set to a minimum when a new sequence script is loaded, and must be adjusted during run time to the level desired.

The TPC provides for rapid changes in high voltage output, allowing the use of different power levels for different modes of acquisition. Each power level can have its own set of unique attributes, including specification of transmit drive voltage level, and maximum power output. The collection of attributes for a given power level is referred to as a profile, and up to four profiles are supported in the current TPC firmware, with a fifth available as a “high power” option.

Profiles are specified using the TPC structure, with the structure index corresponding to the profile number. Note that all TPC attributes for profiles one to four are optional, and if left unset, are filled in with default values or set to empty. If no TPC structure definitions are provided, default definitions are provide for all profile numbers used. This means that for most usage models, the TPC structures need not be defined at all, and one can simply request a different profile number be used at the appropriate time in the Event list. The TPC structure has the following definition:

```
TPC =
     name           string  (optional) name identifier for profile
     maxHighVoltage double  (optional) max high voltage for this profile
     highVoltageLimit double (optional) high voltage limit based on use model
     xmitDuration   double  (optional) longest transmit duration (usec)
     hv             double  (optional) initial High Voltage value at startup
```

`TPC.name` is an optional string that can be used to identify the profile. For example, TPC(1) might be used for 2D imaging and TPC(2) might be used for Doppler. In this case, we could set `TPC(1).name = '2D'` and `TPC(2).name = 'Doppler'` for identification purposes. By default, this field is set to the empty array.

The `TPC.maxHighVoltage` value sets the maximum high voltage level that can be set for the profile, and can be used to set a lower level than the level set in `Trans.maxHighVoltage`. If the `TPC.maxHighVoltage` level is higher than the `Trans.maxHighVoltage` level, the `Trans.maxHighVoltage` level takes precedence. When using TPC profile five, the TPC structure must be provided with a user set `TPC.maxHighVoltage` level. The `TPC.maxHighVoltage` typically is set to a fixed absolute maximum allowed voltage, based on the capabilities of the system HW and transducer or perhaps a maximum acoustic output intensity limit for the transducer, and doesn't change during the running of a sequence.

The `TPC.highVoltageLimit` parameter also restricts the maximum high voltage setting that can be used, but it is intended to serve a slightly different purpose than `TPC.maxHighVoltage`. `TPC.highVoltageLimit` is intended to support a more dynamic limit that may change as a function of the system operating state, and may be more restrictive than `TPC.maxHighVoltage`. For example, in a PW Doppler script that gave the user control of the Doppler PRF and of the transmit burst duration,

`TPC.highVoltageLimit` could be set by an acoustic output control function to maintain a constant output power limit for any combination of burst duration and PRF. In some cases this state-dependent limit might be more restrictive than `TPC.maxhighVoltage` while in other cases it might not. The lower-level SW functions that actually set the transmit voltage in the system will always enforce the most restrictive value from `Trans.maxHighVoltage`, `TPC.maxHighVoltage`, and `TPC.highVoltageLimit`.

The `TPC.xmitDuration` attribute indicates the duration of the transmit period, and should be set to include the longest `TX.Delay` value plus the duration of the transmit burst. This indicates to the controller when it is safe to start the transition to the next profile, which can be taking place during the receive period of an acquisition event. If not provided, this attribute defaults to the entire length of the acquisition period.

Starting with the 3.3.0 software release, if you specify a value for the `TPC(n).hv` attribute for TPC profile n in your SetUp script, the system will automatically start up with that transmit voltage setting for that profile when the script is executed through VSX. If `TPC(n).hv` is not specified, the system will assign a default value of 1.6 Volts and thus the original system behavior of starting up at that minimum voltage setting will be preserved. If the initial voltage setting from `TPC(n).hv` exceeds the maximum allowed voltage as set by `TPC(n).maxHighVoltage` and `TPC(n).highVoltageLimit`, an error will be reported preventing the script from running. This new approach is much simpler than the scheme that was used in earlier releases to initialize the transmit voltage through "UI.Statement" commands, but the UI.Statement capability has been preserved for full backward compatibility with existing scripts using that feature.

3.2.3.1 Managing TPC Profiles in a Sequence

By default, if no TPC profiles are specified in a user sequence that performs typical transmits, a profile 1 will be created, along with its high voltage slider. Multiple profiles can be defined in a setup script, but only those that are actually used in the sequence will have high voltage sliders created. When two or more TPC profiles are defined and used in a sequence, two high voltage sliders are created in the VSX GUI window for controlling the profiles. If profile 5, with the high power option, is one of the additional profiles, then the second slider will control its level; otherwise, the second slider will control the first additional profile specified after the first. If the user wants to have slider control of a third profile level, they will have to provide their own GUI control elements.

Switching from one profile to another is controlled in the Event sequence, by use of a SeqControl command (SeqControl objects are described in section 3.6). The user is responsible for indicating when a new profile is to take effect, and providing adequate time for the transition from the current profile to the new. In addition, the user must take into account the time to replenish the power expended during a transmit event, so that the transmit bus for the specified profile has fully recovered before the next transmit event.

The user can specify a change from one profile to another by means of placing a SeqControl command of `'setTPCProfile'` in an acquisition event (an event that has both a transmit and receive specification or a transmit only specification), with a condition of `'next'` specified with the profile number in the argument. At the end of the `TPC(n).xmitDuration` period, if specified, or if not, the end of the acquisition period, the TPC will begin changing to the new profile's voltage level.

It can take from 800 usecs to as long as 6 to 8 milliseconds to switch from one profile to another, depending on how far away the high voltage level of the new profile is from the level of the old. This time period should be accounted for in the time between acquisition events that use different profiles (see usage for the `'timeToNextAcq'` SeqControl command). If the next transmit event in a sequence is programmed to occur before the TPC completes its transition from one profile to the next, the event will be delayed until the transition completes. If the next event was scheduled using a `'timeToNextAcq'` SeqControl command, a warning that the `'timeToNextAcq'` period was missed will be issued, indicating that insufficient time was allotted for the transition. This warning can be used to fine tune a sequence to allow just enough time for the transition to take place.

Another type of error that can occur with some sequences is a failure of the TPC to reach the set level of a profile. This can occur in certain high PRF sequences where the power drawn from the TPC is greater than the capability of the TPC to restore the set level. In this case, a TPC error is generated that stops the sequence.

In some instances, one would like the TPC profile to take effect immediately at some point in the sequence, and for this action, one can use the `'immediate'` condition of the `'setTPCProfile'` command. This option should be used in a non-acquisition event, meaning that `Event.tx` and `Event.rcv` are both 0. In this case, the transition to the profile specified in the argument field will begin immediately. Again the user is responsible for allowing enough time for the profile transition before the next transmit event. (In this case, a `'noop'` command would have to be used, as the `'timeToNextAcq'` command can only be used from within an acquisition event.) Using the `'setTPCProfile'` command in a non-acquisition event with the `'next'` condition specified will delay the start of the profile transition until after the next acquisition event in the sequence.

3.2.4 High Power Transmit (HIFU)

Two optional configurations of the VDAS system are available to allow use of the system at very high transmit power levels in support of both therapeutic applications using long transmit sequences ("HIFU"), and also short-duration "Push" transmit events used to stimulate the tissue for other purposes. With either of these options, a dedicated separate power supply is used for the high power transmit. This power supply is selected by selecting TPC profile 5 while profiles 1 through 4 always use the 'imaging' transmit power supply. Both options are available with either the two-board (128 transmit channel) or four-board (256 transmit channel) configurations. The two options are described separately below.

The **"Extended Transmit Option"** utilizes a dedicated internal power supply with 48 Watt capacity to power the “Push” transmit bursts. A large energy storage capacitor (referred to as the "push capacitor") provided within the system allows transmit for short bursts at much higher power levels, provided that enough time is allowed between bursts to recharge the capacitor from the internal supply. The amount of transmit power available is highly dependent on the transmit burst duration, the number of active transmit channels, and the load impedance presented by the transducer to the system. Typically transmit power levels from several hundred up to two thousand Watts can be achieved for burst durations of up to a few milliseconds.

The **"HIFU Option"** adds an external HIFU transmit power supply connected to the system. This option supports transmit output power levels of up to 1200 Watts for any desired combination of burst durations and repetition intervals, at any transmit frequency in the 1 to 5 MHz range. With this option installed and a typical transducer load impedance of 50 Ohms, each transmit channel can provide continuous output power levels of up to 8 Watts within the 1 - 5 MHz frequency range.

Imaging modes can be interleaved as desired in between HIFU transmit sequences, utilizing the full range of features and performance provided by the VDAS system for imaging and Doppler modalities. For both the Extended Burst and External HIFU options, TPC Profile 5 is utilized to select the high-power transmit supply while TPC profiles 1 through 4 can still be used in their normal fashion for imaging. This allows the transmit voltage levels for HIFU bursts and imaging to be controlled completely independent of each other. Transmit burst frequency, duration, active aperture, focus, etc. can also be programmed independently for each transmit burst.

3.2.4.1 Controlling the External Power Supply

NOTE: This section only applies to systems configured with the HIFU Option.

As part of the HIFU Option, the system is provided with an external OEM power supply with 1200 Watts capacity. The external power supply connects to the VDAS system transmit circuitry through a connector added to the rear panel of the system. In addition, a SW remote control link connects to the power supply from the host computer for the VDAS system. Through this remote control link and supporting system SW, the user can control the external power supply from the Profile 5 High Voltage slider GUI in exactly the same fashion as would be the case for a system with the Extended Burst Option using the internal auxiliary supply for profile 5.

Listed below are several items the user should be aware of with regard to setting up and using the external power supply:

> **Connecting to the system:** The power supply actually provides two completely independent 600 Watt outputs. For our purposes, we connect these two outputs in parallel and operate the supply in a "voltage tracking- current sharing" mode specifically intended for parallel connection. The cable assembly from the system to the power supply has four wires, two red and two black. These should be directly connected to the four output terminals at the back of the supply, matching red to red and black to black (it does not matter which of the two terminals the two wires of the same color go to). The direct parallel connection of the two outputs occurs inside the VDAS system, not at the power supply. This ensures more uniform current sharing through the cabling and rear panel connector pins. Make sure the power supply connections are tight (as tight as you can make them by hand).
>
> **`Resource.HIFU.extPwrComPortID:`** SW control of the external power supply used with the External HIFU option is implemented through a "virtual serial port" that is assigned by the Windows OS when the driver for the power supply is installed on the user's computer; the physical interface is through a

USB cable provided with the system. Once the power supply driver has been installed (refer to the instruction booklet and CD that came with the power supply), it will be identified on the computer with a port ID such as 'COM3'. In the user's setup script this string value must be assigned to the variable `"Resource.HIFU.extPwrComPortID"`. When VSX executes the script, the value from this variable will be passed to the routine that actually communicates with the power supply.

System Startup: Before attempting to run a script that will use profile 5, make sure the power supply is connected and turned on. If the power supply is not actually connected, or if the system cannot successfully communicate with it for any reason, execution of the script will be terminated and a fatal error will be reported. If this occurs, check all connections and try again. You can also use 'device manager' on the host computer to make sure it recognizes the power supply as a connected device, and that the com port ID number matches the one you have assigned in the Matlab script as explained above. Note that if you are going to be running an imaging script that does not invoke profile 5 at all, then the power supply does not have to be turned on or connected.

Front Panel Controls: If the remote control interface is initialized and functioning properly, the front panel controls on the power supply will be disabled. Do not override this and attempt to control the supply manually from its front panel controls; to do so could easily result in putting the system in a self-destructive state.

3.2.4.2 Using the High Power TPC Profile

For the most part, programming the system for 'HIFU' transmit events using the Extended Burst or External HIFU options is no different than programming for imaging acquisition, as explained throughout the rest of this document. The same ‘setTPCProfile’ SeqControl command is used to select the high power profile from within an Event sequence, with the argument set to 5. When the high power profile is used in conjunction with imaging profile numbers 1 - 4 in a user script, the second high voltage slider that is opened in the GUI controls the high power voltage level. In this case, for the profile 5 slider, an additional text box is created to the right of the normal text box that indicates the setting of the high voltage level. This additional text box reports the average measured voltage level on the energy storage 'push' capacitor. If the sequence being run is draining the storage capacitor more than the charge is being replenished, the measured voltage will drop. When the voltage drops to 80% of the set level, the text background changes to blue, to draw attention to the problem. If this occurs, the sequence must be modified to transmit less power, and/or allow longer durations for re-charging the capacitor.

The are several other system constraints and control parameters that the user must be aware of when utilizing profile 5 for high-power transmit, as outlined in the following paragraphs:

`P5ena` This parameter should not be specified in a user's SetUp script; it is defined and set in the Matlab Workspace automatically by VSX whenever a script

is executed, to indicate the actual HW configuration and operating state of the system. After running VSX, the user can check the value of P5ena in the workspace to confirm the actual operating state of the system:

- If the value is 0, profile 5 is not being invoked by the user's setup script.
- If the value is 1, the Extended Transmit Option is installed and being used on the system.
- If the value is 2, the HIFU Option is installed and being used.

If a script attempts to select TPC profile 5 but VSX determines that the HW configuration does not support it, P5ena will be set to 0, operation of the script will be terminated and a fatal error will be reported.

`Resource.HIFU.externalHifuPwr` This parameter must be specified and set to the value one in a user script that intends to use the External HIFU Option. If the system is configured for the External HIFU Option and the script invokes profile 5, operation will be terminated with a fatal error message if '`Resource.HIFU.externalHifuPwr`' is missing or is not set to 1. For imaging scripts that do not use profile 5 as well as for any script using the Extended Burst Option, this parameter does not need to be specified at all. The only purpose of this parameter is to help ensure that a user is consciously aware their script is going to exercise the 1200 Watt External HIFU power supply. If a script using profile 5 is executed in simulation mode with no HW present, the state of this parameter will be used by VSX to set `profile5ena` to either 1 or 2 (see '`profile5ena`' above) and thus during simulation the limit check algorithms will still know whether the script intends to use the Extended Burst or External HIFU option.

`timeToNextEB:` This is a specialized sequence control (SeqControl) command that can only be used with transmit events utilizing profile 5 with extended burst length. This allows greater flexibility in defining an event sequence that interleaves profile 5 "push" transmit events with imaging data acquisition events. It can also be used to enforce a maximum limit on the duty cycle for profile 5 transmit, thereby allowing operation at higher levels during the burst. The `'timeToNextEB'` command is specified and used in the same way as the `'timeToNextAcq'` command, except that `'timeToNextEB'` can only be used in a profile 5 Extended Burst event, and it sets the delay time to the next profile 5 Extended Burst event, ignoring any imaging bursts that come in between. For example, suppose that you had a single Extended Burst transmit event in your script which was to be followed by a sequence of imaging events and then an idle period before jumping back to repeat the sequence. You could place both a `timeToNextAcq` and a `timeToNextEB` command in the event with the profile 5 Extended Burst length transmit. The `timeToNextAcq` would set the delay to the following imaging event, while the `timeToNextEB` would set the overall delay to the repeat of the profile 5 transmit (i.e. it would be set to the sum of the imaging acquisition time plus the desired additional delay before repeating the sequence).

`Resource.HIFU.voltageTrackP5:` The system allows the use of normal

short-duration imaging transmit bursts from the Profile 5 power supply, which in some cases is convenient since it avoids the overhead time needed by the system to execute a profile transition between one transmit event and the next. But for optimal imaging performance (especially for applications such as Doppler that require very low noise and very good stability from the transmit power supply), the user is advised to switch back to an imaging profile. As a comprise between the conflicting goals of optimizing imaging performance while minimizing profile transition times, the imaging profile voltage can be kept the same as that being used for profile 5. This reduces the profile switch time to less than 500 usec. To facilitate user implementation of this concept, the optional Resource.HIFU.voltageTrackP5 parameter has been defined. If this parameter is specified in the user's setup script and set to the index value for an imaging profile (i.e. a value of 1, 2, 3, or 4), it will disable the GUI control for that profile and instead it will be automatically controlled by the GUI for profile 5. Thus the two profiles will always have the same voltage setting, minimizing profile transition times. If this parameter is not specified or if it is set to the value zero, the tracking feature will be disabled.

Transitions into profile 5: As explained in section 3.2.3.1, it is the user's responsibility to ensure that sufficient time is allowed within the event sequence for a profile transition to be completed, typically through the inclusion of `'noop'`, `'timeToNextAcq'` or `'timeToNextEB'` sequence control commands. Transitions into profile 5 are slightly more complex than other profile transitions, since in this case the HW must physically close a switch that connects the dedicated transmit power supply to the transmit circuitry. It could be self-destructive for the system to execute a transmit event while this switch was still in transition. Because of this, the system HW will actually delay the start of transmit if it occurs before a profile 5 transition is complete. No error will be reported, but if the delay of transmit results in a missed `'timeToNextAcq'`, a warning to that effect will be issued. (If transmit activity starts before the completion of a transition into any profile other than 5, a warning message is displayed at the Matlab command prompt with system execution allowed to continue).

TPC profile initialization: When profile 5 is going to be active in a script, the system also requires that a 'setTPCProfile' command be executed before the first transmit event, to avoid any possible ambiguity regarding the system state. To meet this requirement, it is usually convenient to start a sequence with a non-acquisition event and the 'setTPCProfile' command, with the argument set to the desired initial profile and the 'immediate' condition, followed by a long 'noop' (several milliseconds) to allow the high voltage level to be reached. The normal operation of the sequence can then be entered, with a jump at the end of the sequence back to this later starting point.

3.2.4.3 Transmit Circuit Limit Checking

At the high power levels associated with long transmit burst durations and/or high transmit duty cycles the power dissipation capacity of transmit components within the system can in some cases be exceeded, resulting in permanent damage to the system. To avoid this a limit-checking function is provided in the system. This function is automatically invoked every time a user setup script is executed, and also every time a

transmit parameter is modified while the script is running. If an operating state is detected that could be self-destructive to the system, a warning is displayed to the user and either the transmit power is reduced to a safe level or operation of the script is terminated.

A Word of Caution: The sole purpose of the limit checking algorithms and features described in this section is to protect the VDAS system itself from component damage. The user must be aware that an operating state that is 'safe' for the system may still be putting out enough power to be destructive to their transducer! It is the user's responsibility to define and impose additional restrictions as needed to protect the transducer from damage, and also to apply any acoustic output power limits that may be needed for the intended application. Some of the features and control parameters described in this section can be used to facilitate implementation of any additional output controls that may be needed. Another factor that the user must be aware of is that the Vantage system transmitters function essentially as a voltage source with a very low source impedance; therefore the actual output power and transmit current are highly dependent on the actual load impedance presented to the system by the transducer. This value must be specified by the user in the parameter `Trans.impedance`. An incorrect value for `Trans.impedance` can expose the system and transducer to potentially severe damage.

The limit-checking feature is provided through a separate Matlab function, `'TXEventCheck.m'`. This function is located in the Vantage release directory, in the "HIFU" subfolder. Comments throughout the `TXEventCheck.m` file explain each of the individual limits and how they are evaluated.

There are numerous system parameters used by `TXEventCheck` to communicate with the rest of the system. After executing a script utilizing profile 5, all of these parameters will remain in the Matlab Workspace where they can be reviewed by the user to help understand the limits and how to tailor the transmit operating state for optimal performance while remaining within those limits. There are additional parameters that can be set by the user to control some aspects of system operation. Some of these parameters and the associated features are defined below:

`TPC(5).highVoltageLimit`
`TPC(5).currentLimit:` These two parameters are used by `TXEventCheck` to communicate the transmit output limits that must be imposed to maintain a safe operating state for the system HW. The functions that actually control the profile 5 transmit power supply read these values from the TPC structure and enforce them whenever a change to the transmit voltage setting is initiated. The current limit value `TPC(5).currentLimit` only applies to the external power supply used with the External HIFU Option; for the Extended Burst Option the internal auxiliary power supply self-limits at a maximum output current of approximately 0.5 Amp, independent of `TXEventCheck`.

`Resource.HIFU.verbose:` While it is active, the limit check algorithm displays a large number of status and warning messages through the matlab command window. These are very valuable for the user while developing and testing a setup script, but they can be very annoying and disruptive when exercising a script that has already been thoroughly tested. When desired, the

user can assign a value of zero to `Resource.HIFU.verbose` in their setup script to suppress the warning messages. (Error message reports will always be displayed, however).

`TXEvent and TXthermal:` These are data structures created by `TXEventCheck` and written to the Matlab Workspace. They contain an entry for each profile 5 extended burst length transmit event that was identified in the event sequence. For each of those events, it lists the values of all operating-state parameters that have been identified and evaluated by the limit check algorithms. Review of these structures can help a user gain insight into which parameters are most restrictive for the operating states they have defined in their script. Refer to the code and comments in the `TXEventCheck.m` file for definitions and units of the parameters listed in the `TXEvent` structure.

`Resource.HIFU.TXEventCheckFunction:` To tailor the system for optimal operation with a particular application and transducer, the user may in some rare cases find a need to modify or enhance some aspect of the limit check function '`TXEventCHeck.m`'. To facilitate this, the system provides a means to substitute a different file name in place of the `TXEventCheck.m` function provided as part of the Verasonics SW release. To utilize this capability, make a copy of `TXEventCheck` and give it a different name. Then edit the desired changes into the renamed copy. Finally, assign the new name as a string to the variable '`Resource.HIFU.TXEventCheckFunction'` in the setup script that needs to use the modified function. VSX will use the value assigned to this variable as the name of the function to call when invoking the limit check. Note, however, that any changes or additions to `TXEventCheck` should only be made after consulting with Verasonics. If a user damages their system and it is determined that unauthorized modifications to the limit algorithms had been incorporated, Verasonics will not be responsible for the cost of any repairs even if the system is still under warranty or a service contract.

3.2.4.4 Paralleling transmit channels for higher output

With the 3.4.0 software release or higher, it is now possible to parallel channels for applications that require a higher transmit voltage level than the 190 Volt pk-pk available from a single channel of the Vantage system. This feature can be used with some commercially available HIFU transducers supported by newer software releases, through the “HIFUPlex” pairings 1 through 3, using annular array HIFU transducers available from Sonic Concepts.

Since there is very good channel-to-channel matching of all transmit channels within a Vantage system in terms of source impedance and rise/fall times for the tri-level transmit output, and since all channels share the same transmit voltage supply, up to four Vantage channels can be connected directly in parallel to drive a single transducer element with confidence that the channels will share the load current equally. As a result, for N channels in parallel the available transmit output current (and thus output power) will increase by a factor of N, and the effective transmit source impedance will decrease by a factor of N for the paralleled channels. Direct parallel connections should not be used over more than four channels because the uniformity of load current sharing will deteriorate with higher channel counts. Combining larger numbers of channels can be achieved by providing a “combining network” (typically a multi-winding

transformer or something functionally equivalent to that), such that all of the combiner input ports connected to the Vantage transmit channels present a closely matched load impedance. The HIFUPlex annular arrays use this approach, to combine up to 16 Vantage channels through a combiner with four inputs, and four Vantage transmit channels connected directly in parallel to each input. The combiner network can also be designed to provide an impedance transformation, to produce an output voltage to the transducer element that is a multiple of the voltage coming from the Vantage transmit channels.

Regardless of how the channel combining is done, through either direct paralleling of up to four channels or a combiner network with multiple input ports or a combination of both, the net effect is the same as long as one key requirement is met: *all of the transmit channels being combined to drive a single element must be programmed with the exact same transmit waveform and transmit delay!!* If there is ever any difference in the tri-state output levels of the channels to be combined, even for only a fraction of a microsecond, the net result could be instantly destructive to the transmit channels when operating at high transmit power levels!!

To avoid this potentially catastrophic mistake when paralleling transmit channels, it is very important to verify that the per-channel programming with the script actually matches the wiring of channels to elements within the probe, for each group of channels that are to be combined to drive a single element. The Vantage 3.4.0 release includes new software features to simplify script development for probes using paralleled channels, and also a new test utility to provide a semi-automated, foolproof way of verifying the actual parallel connections that are present in the probe.

As an example, suppose you have a 3-element transducer with each element wired to four Vantage transmit channels connected in parallel. This means the programming of the Vantage system must actually enable 12 individual transmit channels. In software releases prior to 3.4.0, the system had no way of associating more than one transmit channel with a single element. Instead you had to trick the system into thinking it was driving a probe with 12 elements, by creating a Trans.Connector array with 12 entries, each identifying one of the transmit connector channels that are going to be used. The TX.Apod and TX.Delay arrays will have 12 entries, one for each ‘element’ and thus the associated channel identified by Trans.Connector. You would have to manually ensure that each group of four channels that will be connected in parallel to drive a single element have identical entries in TX.Apod and TX.Delay (and also in the TW waveform definition if you were providing a unique waveform definition for each element). This approach was confusing and cumbersome, since you had to map the three elements you are actually trying to program to their respective groups of four channels, while making sure you make no mistakes in the mapping of each group of four to be connected in parallel. And the stakes are high- any innocent typo in that mapping logic could destroy your system!

The concept for the Vantage 3.4.0 release is that we have a new mechanism in the system software to identify the channels that will be connected in parallel, with the result that in the Trans and TX structures you now can define the transducer as it actually exists, with only 3 elements for the probe in our example. Thus TX.Apod and TX.Delay will have only three entries, one for each element. If you want to specify unique per-element waveforms in a TW structure, you will also only specify three waveforms there

instead of 12. When VSX, runAcq, and SequenceLoad process your script to run it on the hardware system, they will detect the mapping you have specified from each element to the group of four channels connected in parallel that will be driving that element. Since the mapping from elements to channels is being done automatically by the low-level software utilities, you no longer have to worry about a programming error or typo in your SetUp script corrupting that mapping. However, you do still have to ensure that the element-to-channel mapping specified for the probe in the Trans structure actually matches the way the probe is built. But once you have a Trans structure that accurately identifies the mapping, you can re-use that structure in multiple scripts rather than having to re-create the mapping in any new script, as was required in the pre-3.4.0 software releases.

The mechanism used in the 3.4.0 software release is a simple extension of the `Trans.Connector` array that has always been required in every Vantage SetUp script for probes that do not use HVMux switching. In Vantage software releases prior to 3.4.0, `Trans.Connector` is a column vector of length equal to the number of transducer elements. Each entry in `Trans.Connector` specifies the connector channel number of the system that is connected to that element of the probe. If a probe has elements that are not used by the system at all, a zero value in `Trans.Connector` for those elements is used to signify that no system connector channel is connected. (The `Trans.Connector` array is optional, but only for probes with the same number of elements as the system has connector channels and with a one-to-one mapping from element numbers to connector channel numbers. If those conditions are met, the `Trans.Connector` array can be left unspecified and the system will create it by default with the one-to-one mapping.)

Starting with the Vantage 3.4.0 release, you can specify that more than one channel will be connected in parallel to a transducer element by simply adding multiple columns to the `Trans.Connector` array. In our example of a three-element transducer with each element being driven by four channels connected in parallel, `Trans.Connector` will be specified as an array of three rows by four columns. The four values in each row will identify the four connector channels that the transducer will connect in parallel to that element. Thus the `TX.Apod` and `TX.Delay` arrays will only have three entries for our example, one for each element in the transducer and the associated row of `Trans.Connector`. At script initialization, the system software will automatically map the Apod, Delay, and TW waveform values for each element to all of the channels identified in the corresponding row of `Trans.Connector` (this is identical to what the system has always done with a single column `Trans.Connector array`).

The "ParallelDetect" Test Utility - As noted previously, the new `Trans.Connector` scheme will simplify the creation of user scripts for probes that connect elements in parallel, but it is critical that the Trans.Connector array be an accurate representation of the actual parallel-channel wiring within the probe since an error could lead to self-destructive operating states if a script is run at voltage settings above the 1.6 Volt minimum. The `ParallelDetect` utility can be run with a probe connected to determine which system channels are actually connected in parallel; the results can be used to verify whether a previously defined multi-column `Trans.Connector` array is accurate, or to generate the Trans.Connector array for a new probe.

ParallelDetect is essentially a crosstalk-measuring acquisition script. It executes a transmit-receive acquisition event for each individual channel of the system, by

transmitting a short low amplitude burst on one element at a time and then examining the electrical crosstalk from that transmit channel that appears in all other receive channels. Acoustic coupling paths are excluded, by ignoring any signal content in the receive data beyond the actual duration of the transmit burst. Since the worst-case channel to channel crosstalk levels in a typical probe are at least 20 dB down (even for channels on adjacent pins of the connector), it is easy to differentiate those from channels that are directly connected together, where the measured crosstalk level on the parallel-connected receive channels will only be a few dB down from the transmit output level. These measurements are conducted at the minimum transmit voltage setting of 1.6 Volts. At this voltage, the transmit current even for a direct short to ground will be low enough that it will not be harmful to the transmitters, and the burst duration is also kept very short with a very long repetition interval to further reduce the risk of damage. The script creates a list of the receive channels that appear to be connected in parallel with each individual transmit channel; then it compares those lists for all transmit channels within the group. If all lists are identical, that group of channels is listed by the script as a parallel group and this process is repeated for all parallel groups found across all channels in the system. The list of channels in each group should then be identical to the channels listed in one row of the `Trans.Connector` array. If they do not match exactly, there is either a problem in the way the Trans.Connector array was generated or in the wiring of the probe that was tested. In either case, do not attempt to use the probe or associated script until you have found and corrected the discrepancy so a repeat of the ParallelDetect script leads to an exact match.

Parallel Channel Programming Constraints and Guidelines -

- If the probe is using a combining network to drive an element with several groups of paralleled channels, all channels from all of those groups should be listed in one row of the `Trans.Connector` array since all should be using the identical transmit waveform and delay. The ParallelDetect script should be able to detect all of the channels that are coupled through the combining network, not just the group that is directly connected in parallel (but this may require fine-tuning of the parallel detection threshold in the script).
- Obviously, the same channel number must not appear in more than one row of the `Trans.Connector` array. Any channels that are not listed in the `Trans.Connector` array at all will be automatically disabled by the system (just as has always been the case with a single-column `Trans.Connector` array).
- If the transducer includes elements that will not be used by the system at all, zeroes must be placed in the entire element row of the `Trans.Connector` array.
- The number of columns in the `Trans.Connector` array must correspond to the largest group of paralleled channels. If some elements of the transducer will have fewer channels in parallel, then simply repeat one of the channel numbers as needed to fully populate the entire row. One example of this situation is a script driving two probes, one of which is for HIFU transmit with paralleled channels and the other an imaging probe with only one channel per element. In this situation, the script will require a composite Trans.Connector array that includes rows for all elements of both probes. The number of channels being paralleled for the HIFU array will set the number of columns in the `Trans.Connector` array; for the imaging probe simply repeat the channel number for an element in all columns of `Trans.Connector`.

- Paralleling of channels to drive a single element cannot be done with an HVMux probe, since paralleling of channels to drive one element can only be done on a system that has more channels than the probe has elements. HVMux probes do not have a `Trans.Connector` array - its role is replaced by the `Trans.HVMux.Aperture` array (which specifies the unique element-to-channel mapping that can be selected by each setting of the HVMux switches). Note that if the HVMux wiring allows it, you can define an HVMux.Aperture array that connects multiple elements to the same channel but you cannot connect one element to multiple channels. Note also that since HVMux switches have less current-carrying capacity than the transmit channel output, it would be pointless to connect transmit channels in parallel through an HVMux array.

3.3 Receive Objects

Receive objects define all of the characteristics of the receive phase of an acquisition event. In both the Vantage hardware and the simulation software, the TX and Receive periods start at the same point in time, or time 0. This means that as the transmitters are counting down their delay periods, the receive channels are already sampling the receive signals, since receive sampling also starts at time 0.

There are many parameters to specify for a receive period, including A/D converter sample rate, post-A/D processing functions, and storage locations in local memory on the Vantage modules and host memory. In most cases, the system is able to automatically select default operating parameters from a minimal set of user provided parameters, freeing the user from having to add a multitude of parameters to their setup scripts. The VSX loader program will automatically add the parameters to the user structures, which the user can examine after running a script.

The objectives of the receive processing in the Vantage hardware are:

1) Provide adequate A/D sampling of the receive spectrum, including the noise spectrum of the front end, where possible. Set anti-alias lowpass filter cutoff prior to the A/D appropriately for the selected sample rate.
2) Lowpass filter and decimate the A/D sample rate, if necessary, to provide a sample rate of approximately four times the center frequency of the transducer spectrum. The 4xFc sample rate improves the efficiency of the downstream filtering and any further sample rate reductions for narrow band signals. This rate is also required for any image reconstruction processing.
3) Bandpass filter the 4xFc sampled signal to match the transducer bandwidth, or to narrow the bandwidth of the signal for Doppler processing.
4) Decimate the 4xFc sample rate to the rate needed for adequately sampling the bandwidth of the filtered received signal. Here the decimation is performed on sample pairs, which for the 4xFc sample rate represent non-concurrent samples of the baseband I and Q signals. For 200% bandwidth, all of the 4xFc samples are kept; for 100% bandwidth, every other sample pair is discarded; for 50% bandwidth, one sample pair out of four is kept. The original filtered receive RF signal can be reconstructed precisely from these decimated sample pairs, provided the decimated sample rate is adequate to represent the bandwidth of the filtered RF signal.
5) Scale each receive channel's output independently. The scaling is used for receive apodization, or can be used to compensate for gain variations in the transducer elements. It can also be used for normalizing accumulated acquisitions.
6) Allow averaging of newly acquired receive data with data from previous acquisitions, if desired.

To help understand the signal processing that takes place in the hardware, one can refer to figure 3.3.1 below. The figure shows the signal path for a single receive channel in the Vantage system. In the figure, the attributes in red are Vantage hardware related attributes that are typically programmed automatically by the system.

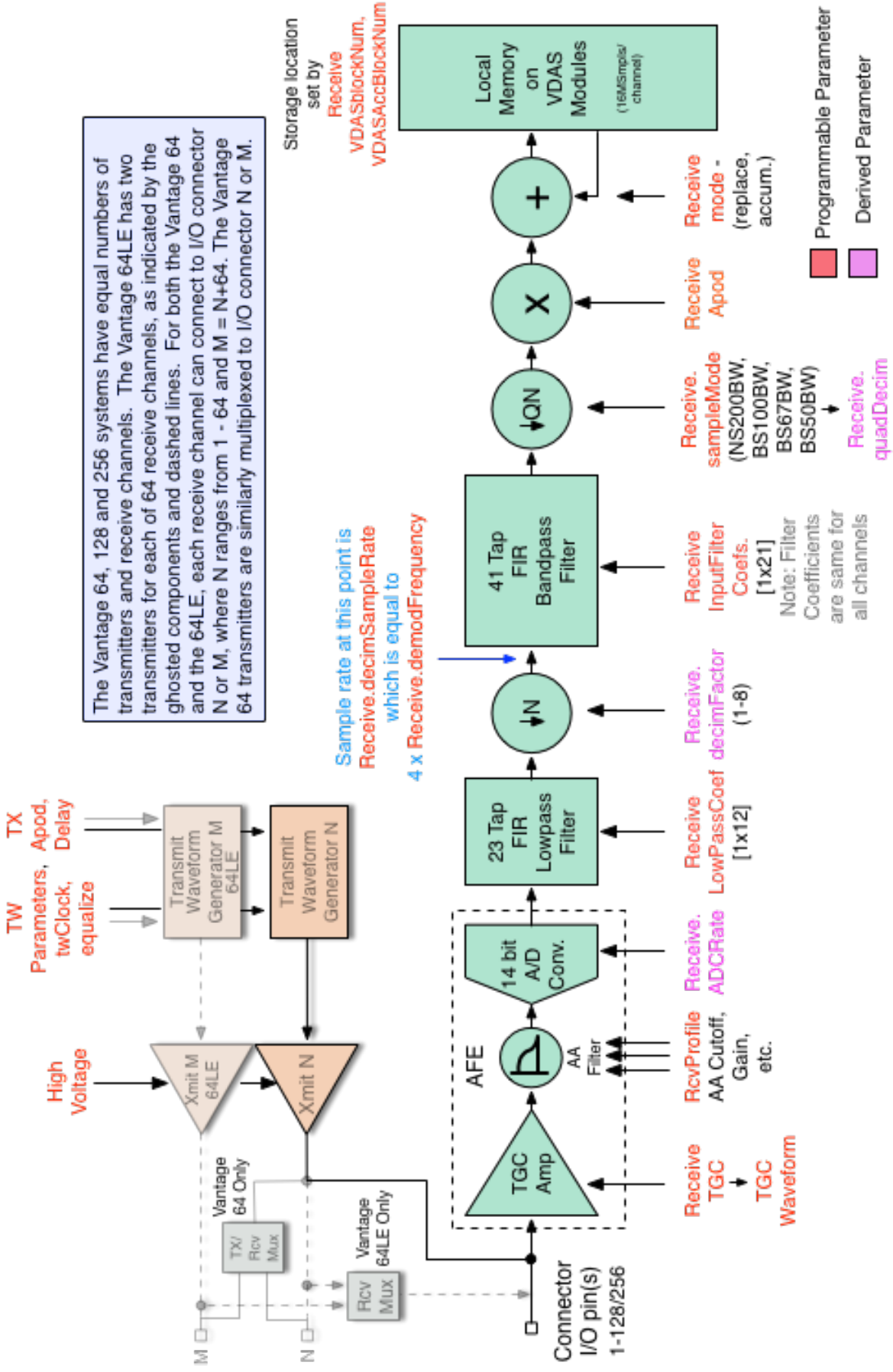


Fig. 3.3.1 Vantage hardware module signal processing path for single receive channel.

Signal Path Notes (for the technically inclined) -

- The Time Gain Control Amplifier (TGC) gain is programmed with a waveform specified in the referenced TGC structure. The gain is typically programmed to increase with increasing depth of the acquisition to compensate for signal attenuation. The TGC gain control range is 40 dB. There is a fixed gain low-noise preamp ahead of the TGC gain stage, and another fixed-gain buffer after it, before the A/D. Several gain settings are available for each of these fixed gain stages, to allow user adjustment of the overall gain from the transducer input to the A/D. These gain settings can be programmed using the RcvProfile structure (see section 3.3.3).
- Prior to the A/D there is a programmable anti-aliasing lowpass filter, with the cutoff frequency automatically selected by the system based on the specified transducer frequency and bandwidth. The user can override this automatic selection using the RcvProfile structure.
- The A/D converter is a 14 bit converter that can sample at rates from 10 MHz to 62.5 MHz (sample rate must be an integer submultiple of the 250 MHz system clock rate).
- Lowpass filter: In the digital signal path after the A/D, this symmetric 23 tap filter is programmed with twelve 16 bit coefficients with range of +/- 1.0. The first entry in the coefficient array is the coefficient used with the outermost pair of taps in the 23-tap FIR structure; the twelfth entry is the coefficient used for the center tap. 19 bits are carried at the output of each multiplier and through the summation (one extra MSB to prevent overflow with full-scale input data and coefficients, and four additional LSB's). Overflow detection and clipping are performed at the output, and the 2 lsbs are dropped to give a 16 bit output word width.
- Decimator: The decimator can be programmed to skip from 0 to 7 samples when passing lowpass filter output samples to the bandpass filter. The idea is to allow the A/D to run at high sample rates to sample the full noise spectrum, but deliver samples at four times the center frequency to the bandpass filter.
- Bandpass filter: This symmetric 41 tap filter is programmed with twenty-one 16 bit coefficients with a range of +/- 1.0. The first entry in the coefficient array is the coefficient used for the outermost pair of taps in the 41 tap FIR structure (taps 1 and 41); the second coefficient is used for taps 2 and 40 and so on; the twenty-first coefficient is used for tap 21. Internally, the filter summation provides two additional LSB's and one additional MSB, for a total of 19 bits. As with the lowpass filter, overflow detection and clipping logic at the output allow the MSB to be dropped, and the two extra LSB's are truncated to produce the 16-bit output width.
- Quadrature decimator: This decimator is for reducing the number of output samples stored in memory for a narrow band signal. The decimator options are: a) passes all the output samples (200% bandwidth), b) passes 2 samples, skips two samples (100% bandwidth), c) passes 2 samples, skips 6 samples (50% bandwidth).
- Scaling multiplier: A 16 bit coefficient width is provided for this 16x16 bit multiplier, scaled to allow a maximum coefficient value of +/- 4.0. Allowing coefficients with absolute values greater than one allows gain compensation for channel-to-channel gain errors centered on a nominal value of 1.0. Another overflow detection and clipping circuit is needed at the output to preserve the output word width of 16 bits.
- Accumulator: Allows adding incoming acquisition data to previously acquired data. Overflow detection and clipping are performed on values that exceed 16 bits.

3.3.1 The Receive Object

The primary object for setting the receive acquisition characteristics is the Receive object. The attributes of the Receive object are detailed below. The required attributes are highlighted in yellow. There are also some dependent attributes that are derived from others and added by the runtime software for reference. These are highlighted in blue.

```
Receive =
     mode          double  0=replace data, 1=accumulate
     Apod          [1xnele double] Apodization value (see below) for rcv aper.
     aperture      double  only used with HVMux probes to specify sub-aperture.
     startDepth    double  starting depth of acquisition in wavelengths
     endDepth      double  ending depth of acquisition in wavelengths.
     TGC           double  number of TGC waveform object to use.
     bufnum        double  number of rcv buffer to use (defined in Resource)
     framenum      double  number of frame in rcv buffer to use.
     acqNum        double  no. of acquisition in frame sequence.
     sampleMode    string  'NS200BW(I)','BS100BW','BS67BW','BS50BW','custom'
     decimSampleRate double sample rate after decimation in MHz.
     demodFrequency double frequency that will be translated to 0.
     decimFactor double  decimation factor after Low Pass Filter
     ADCRate       double  A/D sample rate in MHz
     quadDecim     double  decimation factor after input filter.
     samplesPerWave double samples/wavelength of Trans.frequency
     startSample double  start row for this acqNum (added by VSX).
     endSample     double  ending row for this acqNum (added by VSX).
     LowPassCoef [1x12]double sym. coefs for 23 tap FIR following A/D
     InputFilter [1x21]double sym. coefs of 41 tap input fltr (4*Fc smpls)
     callMediaFunc double 1=call Media Function (Simulator only)
```

The `Receive.mode` attribute is a number that specifies the method of storing acquisition data. For `Receive.mode = 0`, the newly acquired data is written to the storage memory, replacing any previous data. For `Receive.mode = 1`, the newly acquired data is accumulated into the storage memory, by first reading the data stored in memory at the address specified by `VDASblockNum`, adding it to the input data, and writing the sum back to `VDASblockNum`. (see section 6.1 for a more thorough explanation of how to implement accumulation.)

The `Receive.Apod` array has a one-to-one correspondence with the transducer elements in the active aperture. A participating receive element is indicated by non-zero value, while a zero value turns off the receive channel associated with the element and eliminates the element from any reconstruction processing. The `Receive.Apod` values can range from -4.0 to 3.9997 and are used to set the gain of the multiplier stage at the end of the signal processing chain. When negative values are used, the receive samples are sign inverted as well as scaled. The nominal `Receive.Apod` value of 1.0 scales the filtered 14 bit A/D data to one half of the 16 bit Receive Data output sample values. In other words the -8192/+8191 count output of the 14 bit A/D becomes -16384/+16382. Thus an extra least significant bit of dynamic range generated by the input filters is preserved. This scaling to less than full scale in the 16 bit memory word was chosen to leave 1 bit of word growth for gain compensation or signal averaging. For example, the user may want to scale the receive channel outputs higher or lower for purposes of compensating for receiver gain variations or variations in transducer

element sensitivity.

For a Vantage 64 or 64LE system, the `Receive.Apod` array can have no more than 64 non-zero values. Furthermore, the receive channel multiplexing imposes the added condition that array indices N and N+64 cannot both be non-zero, where N ranges from 1 - 64. VSX will check for these conditions and report an error if they are not satisfied.

The `Receive.aperture` attribute is used only for transducers with multiplexers in the probe handle or body. The value of Receive.aperture specifies an aperture defined in the `Trans.HVMux.Aperture` array. Use of the `Receive.aperture` attribute without a corresponding `Trans.HVMux.Aperture` definition will result in an error when the script is loaded by the loader program, VSX. The value of `Receive.aperture` refers to the second index of the `Trans.HVMux.Aperture` array and selects one of the defined sub-aperture positions across the full array. When a sub-aperture is selected using the `Receive.aperture` attribute, the `Receive.Apod` values then apply to the linear non-zero elements of the sub-aperture. The column in `Trans.HVMux.Aperture` (as selected by `Receive.aperture`) specifies the mapping from transducer elements to connector channels for the selected transducer connector on the system. [For transducers that do not have multiplexers (and thus no `Receive.aperture` value in the Receive structure), the `Trans.Connector` array provides the same element-to-connector channel mapping function.] If the `Receive.aperture` value differs from the `TX.aperture` value in the same Event, the high-voltage multiplexers in the transducer or transducer adapter must wait to switch until after the last transmitter has transmitted and the time for switching may amount to several microseconds. This may delay the reception of valid receive data from all channels in the RcvBuffer.

The `Receive.startDepth` attribute specifies the depth to start acquiring samples in wavelengths of the center frequency of the receive spectrum. The `Receive.endDepth` attribute specifies the depth at which to stop. If an image reconstruction is to be performed, the maximum range in wavelengths supported is 1024. When running with the Vantage hardware, the `Receive.endDepth` parameter controls how long the acquisition period runs. There is no fixed maximum limit on the value of Receive.endDepth or on the maximum number of samples that can be required in a single acquisition. Extremely long acquisition events will, however, eventually run into some other system constraint (such as the amount of memory available in the HW system, or the maximum DMA transfer size, or the memory available in the host computer for the matlab workspace, etc.) Since the Vantage hardware stores input samples in 128 sample blocks, the depth in wavelengths will get converted by the loader program, VSX, to samples (using the `Receive.samplesPerWave` attribute to determine the samples per wavelength) and rounded up to the next 128 sample boundary. In other words, if `Receive.endDepth - Receive.startDepth` is not a multiple of 128 samples, the `Receive.endDepth` will be automatically extended to acquire a multiple of 128 samples.

The `Receive.TGC` attribute contains the index into a collection of TGC waveform structures, which are defined below. The receive simulator supports attenuation in the media models, so a TGC waveform can be used when running in simulate mode. The Vantage hardware can store a number of pre-defined waveforms in local memory on the acquisition modules, and VSX currently allocates memory for 8 curves. The number of the waveform in local memory corresponds to the number of the TGC waveform structure index. When a Receive object is loaded into the Vantage hardware, the

`Receive.TGC` number is used to determine the location in the local memory area from which to load the waveform.

The `Receive.bufNum` attribute is an integer that defines the receive buffer number in CPU memory that the acquisition data should be transferred to. The characteristics of this buffer are defined in the Resource object defined above. Each buffer is generally defined to accept multiple frames of acquisition data where each frame may consist of multiple acquisitions.

The `Receive.frameNum` attribute is an integer that defines the frame number of the `RcvBuffer` (specified earlier) in CPU memory that the acquisition data should be transferred to. The `RcvBuffer` frames are required to be equal in size and structure, and each frame should be defined by a similar set of `Receive` structures, with the exception of the frame number.

The `Receive.acqNum` attribute is an integer that defines the acquisition number of a multiple acquisition frame of receive data. When a `RcvBuffer` frame contains data from multiple acquisitions, the separate acquisition event samples for a given channel are stored sequentially in memory. [For Matlab arrays, data is stored in sequential memory down columns.] For example, a `RcvBuffer` frame that contains five acquisition events would have data stored sequentially in memory as follows:

Column 1	Column 2	Column 3
Ch.1-Acq1, Acq2, Acq3, Acq4, Acq5, ...	Ch.2-Acq1, Acq2, Acq3, Acq4, Acq5, ...	Ch.3-Acq1, …

The size of each acquisition segment in samples is given by `Receive(j)`:

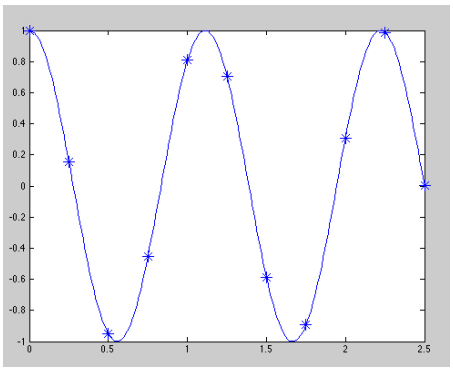
```
n = 2*(Receive(j).endDepth-Receive(j).startDepth)*Receive(j).samplesPerWave
```

with the result, n, rounded up to the next 128 sample boundary. When defining the Receive structures for a frame of acquisitions, the `Receive.acqNum` values should be defined in ascending order, starting with one as the first acquisition for the frame. The loader program, VSX, will then assign sequential storage locations for the acquisitions in local memory on the Vantage acquisition modules, so that all the acquisitions can be transferred to host memory in a single DMA operation. In certain frame acquisition sequences, such as when acquiring ray lines with multiple transmits using different focal zones (see section 6.3 on coding for multiple transmit zones), it may be desirable to partially overwrite previously acquired data in local memory. This can be achieved by reusing a previously specified `acqNum` in the sequence of `Receive` structures defined for the frame. When a `Receive.acqNum` is specified that is equal to or less than the highest `Receive.acqNum` previously defined, VSX tries to find the associated previously defined Receive structure and assigns the new Receive structure the same storage location in local memory. An error occurs if the storage required for the new Receive is greater than that used for the previously defined structure.

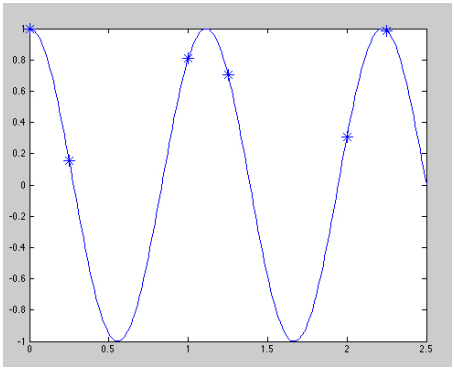
The `Receive.sampleMode` attribute is used to specify the method of sampling for receive data stored in the `RcvBuffer`. The choices for `Receive.sampleMode` are:

```
'NS200BW'    Nyquist sampling (200% bandwidth) of demodFrequency.
'NS200BWI'   2-1 interleaved sampling (requires 2 acquisitions).
'BS100BW'    100% bandwidth sampling of demodFrequency.
'BS67BW'     67% bandwidth sampling of demodFrequency.
'BS50BW'     50% bandwidth sampling of demodFrequency.
'custom'     sample rate set by decimSampleRate
```

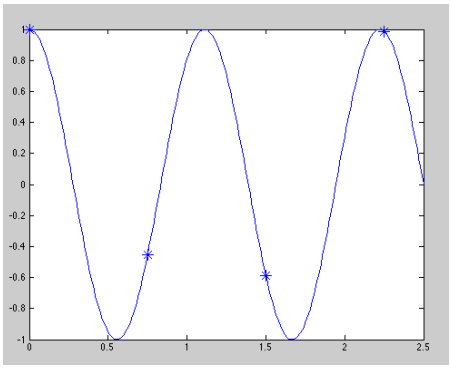
To illustrate how these different sampling schemes are implemented, the samples for a cosine wave at 90% of the realizable center frequency (effective center frequency or `demodFrequency`) are shown in the figures below.



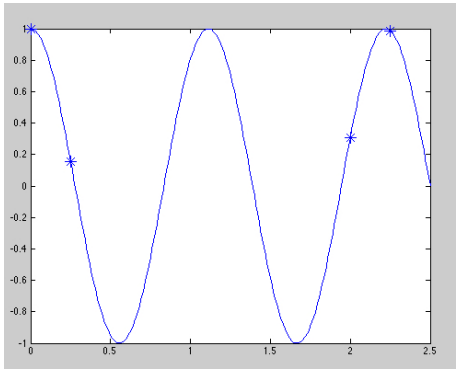
`'NS200BW'` 4 samples per period of the effective center frequency



`'BS100BW'` 2 samples at 1/4th of the effective center frequency period taken every period.



`'BS67BW'` 1 sample every 3/4 period of the effective center frequency.



`'BS50BW'` 2 samples at 1/4th of the effective center frequency period taken every 2 periods.

Fig. 3.3.1.1 Sample points for various SampleModes

Provided that the receive signal meets the bandwidth requirements, each of these `sampleMode` methods allows reconstructing precisely the original receive signal at any sample rate desired. The Verasonics image reconstruction software can work with any of the provided `sampleModes`. Using the lower sampling modes for narrow band signals has the advantage of requiring less memory for data storage, and reduces time and bus bandwidth for data transfers to the host computer memory. Most ultrasound transducers have typical -6dB bandwidths in the range of 60 - 100%, and Doppler measurements typically are made with transmit bandwidths of around 20 - 30%.

When defining `Receives` in a setup script, the user should choose a `Receive.sampleMode` according to their sampling and bandwidth requirements. If no `sampleMode` is defined, the system software sets the default attribute of `'NS200BW'`. With the first five `sampleMode` choices, the system software then will select an appropriate `ADCRate, decimSampleRate, decimFactor, demodFrequency` and `quadDecim` factor, using the provided values of `Trans.frequency` and

Receive.sampleMode. For example, for a given Trans.frequency and the 'NS200BW' sampleMode, the software will pick the highest ADCRate that can be decimated after the low pass filter to yield a 4*F rate where F is as close as possible to the Trans.frequency value, based on Vantage's available sample rates. The decimation factor after the low pass filter will be set in decimFactor and the resulting sample rate in decimSampleRate. The demodFrequency will be set to 1/4th of the decimSampleRate and the quadDecim factor will be set to 1 (for 200% bandwidth). The sample rate at which samples are stored in local memory is always the decimSampleRate divided by the quadDecim factor. The samplesPerWave attribute will then be set to specify the actual number of samples in the Receive Buffer per wavelength of Trans.frequency. If the Trans.frequency is precisely 1/4 of the selected sample rate, the Receive.samplesPerWave will equal 4.0, but if this is not the case, the Receive.samplesPerWave might be more or less than 4.0, depending on where the closest realizable 4*F sample rate falls. For example, if Trans.frequency = 5.0, and the sampleMode is 'NS200BW', the closest 4*5.0 sample rate that can be realized is 19.2308 MHz (250/13), yielding a Receive.samplesPerWave value of 3.8462. In this case, the Receive.demodFrequency will be equal to 19.2308 MHz/4 or 4.8077 and will not match the Trans.frequency value. If the user is planning on doing Doppler processing, they will want to match an Event's transmit frequency (defined in TX(Event.tx)) with the Event's Receive.demodFrequency (defined in Receive(Event.rcv)) to insure baseband I and Q samples. No warning will be issued for a derived demodFrequency that isn't aligned with the Trans.frequency value.

Typically, the Receive.sampleMode, along with the Trans.frequency are all that are needed to specify a receive sample rate. If the user wishes to set a sample rate independent of that derived from Trans.frequency, it is permitted to specify a Receive.sampleMode and at the same time a Receive.decimSampleRate or Receive.demodFrequency (if both are specified, they must be consistent). In this case, the runtime software will not use the Trans.frequency value, but will instead try to find a realizable ADCRate that after decimation by up to a factor of 8 yields the user supplied decimSampleRate or demodFrequency. The software will then set the other dependent attributes, including the Receive.samplesPerWave attribute. For example, if the user sets sampleMode to 'NS200BW' and decimSampleRate to 20.8MHz, the runtime software will then choose an ADCRate of 62.5 and a decimFactor of 3. The decimSampleRate will be modified to 20.8333 MHz, and with a 5MHz Trans.frequency, the Receive.samplesPerWave will be set to 4.1667. The Receive.demodFrequency will then compute to 20.8333/4 or 5.2083 MHz. Again, for Doppler acquisitions, the user would want to transmit at 5.2083 MHz to insure baseband I and Q samples.

If the user chooses the 'custom' choice for sampleMode, it is allowed to set a value for the decimSampleRate or ADCRate directly. If they do so, the value will be adjusted to the nearest realizable rate supported by the hardware. If no decimSampleRate is provided, it is set to the same value as ADCRate, programming decimFactor to 1. If the user provides a decimSampleRate value without specifying ADCRate, the software will select the nearest realizable ADRate with the highest decimFactor that yields the user's decimSampleRate value. If both ADCRate and decimSampleRate are provided, they must be consistent with integer decimFactors of 1 - 8. In all cases of a 'custom' sampleMode, the demodFrequency is set by the software to 1/4th of the decimSampleRate, and the quadDecim factor set to 1. A Receive.sampleMode =

'custom' cannot be used in a Receive that is provided to a reconstruction action and will result in an error.

Note that the Receive.sampleMode can be set independently in any individual Receive event. This allows specifying different acquisition sample rates within the same RcvBuffer frame, as may be desirable when using mixed mode acquisitions, such as 2D and Doppler, where the 2D acquisitions require 200% bandwidth and the Doppler acquisitions can use 50% bandwidth. It is always desirable to use the lowest bandwidth that is suitable for an acquisition to reduce the DMA transfer bandwidth required to move the acquisition data to the host.

The Receive.interleave attribute has been eliminated and interleaving is now indicated by the 'NS200BWI' sampleMode attribute. If the runtime software computes an ADCRate above 62.5MHz for a decimSampleRate of 4*Trans.frequency, the ADCRate is simply cut in halve on the assumption that 2-1 interleaved acquisitions will be used. The center frequencies where interleaved sampling is supported are:

CenterFreq.	4x SampleRate	VDASADClk	VDASADRate	VDASFilter
17.85	71.429	35.714	7	1
20.8333 MHz	83.3333MHz	41.6667MHz	6	1
25.0	100.0	50.0	5	1
31.25	125.0	62.5	4	1

With 200% bandwidth sampling, the Trans.frequency value need not be one of the precise frequencies listed above (the Trans.frequency should be within 10-20% of a realizable frequency for adequate sampling), but one should be aware that the demodFrequency must always be one of the center frequencies listed. The user is responsible for programming the appropriate dual acquisitions with the proper TX delay in the two interleaved acquisitions. The TX of the first acquisition of the interleave pair should have the additional TX.Delay time. The reconstruction will automatically call the ReconInfo.Pre = 'interleaveRF' function for Receives that have the 'NS200BWI' sampleMode attribute. This function will take two consecutive acquisitions and interleave them. The decimSampleRate and demodFrequency attributes of the Receive are programmed to match the interleaved data sample rate. For more information on interleaved sampling, see section 7.2.2.2.

The Receive.samplesPerWave attribute is now a dependent attribute derived from other attributes. For backwards compatibility, the Receive.samplesPerWave attribute will be allowed to be specified by the user, as long as no sampleMode attribute is provided. In this case, the old restrictions on samplesPerWave will apply, and values of 4, 2, 4/3, and 1 will get translated into the corresponding sampleMode attributes, and the Trans.frequency value will be required to be an 'allowed' value based on realizable Vantage sample rates. For no sampleMode and a samplesPerWave other than 4, 2, 4/3, or 1, the software will set the 'custom' value for the sampleMode attribute, and compute the ADCRate accordingly. If the sampleMode attribute is provided, the samplesPerWave attribute becomes a derived parameter and its calculated value will overwrite any user provided value.

The Receive.startSample and Receive.endSample values are computed by the runtime software (when a sequence setup script is executed) from the Receive.startDepth and Receive.endDepth values, using the Receive.samplesPerWave and the Receive.acqNum values. These attributes are computed as a convenience for the user to use in accessing RF acquisition data from a

`RcvBuffer`. For example, after executing a setup script, one could plot the RF data for channel 32 for the third acquisition (`Receive(3).acqNum = 3`) in RcvBuffer one, frame one, using the Matlab statement:

```
> plot(RcvData{1}(Receive(3).startSample:Receive(3).endSample,32,1))
```

The `Receive.LowPassCoef` attribute is a row vector of twelve values that represent the symmetric coefficients of a 23 tap FIR filter that is applied directly after the A/D converter for each receive channel. At this point, the sample rate is the same as the A/D sample rate. The first eleven values (C1 - C11) are the symmetric coefficients of the first and last eleven filter taps, while the last value (C12) represents the coefficient for the center tap of the filter. The coefficients have a range of -1.0 to +1.0. The setting of the `Receive.LowPassCoef` attribute is optional, and if not defined by the user, a suitable set of default values are provided by VSX when the user script is executed. The intent of this filter stage is to remove frequencies from the receive input spectrum that are greater than twice the center frequency of the transducer, allowing decimation of the sample rate (if necessary) to 4*Fc. This decimation is needed for low frequency transducers where the A/D rate cannot be set to 4*Fc (the minimum A/D rate is 10 MSPS). The 4*Fc sample rate is required for correct operation of the Input Filter stage that follows, and for the image reconstruction routines. If the A/D sample rate does not need to be decimated (it is already at 4*Fc), the Low Pass FIR filter can be used to perform other filter operations, and can be programmed as a high pass, bandpass, or even a bandstop filter.

The `Receive.InputFilter` attribute is a row vector of 21 values that represent the non-zero coefficients of a 41 tap FIR filter that is applied to each receive channel following the decimator at the output of the Low Pass Filter. At this point the sample rate is assumed to be approximately four times the receive center frequency. The first 20 values (C1 – C20) are the symmetric coefficients of the 41 tap bandpass FIR filter, while the last value (C21) is the coefficient for the center tap. The coefficients can have a range of -1.0 to +1.0. The intent of the Input Filter is to further shape the receiver input spectrum to improve signal-to-noise and eliminate DC offsets in the receiver and A/D converter outputs. For typical acquisitions, one would preferably use a bandpass filter that matches the transmitted spectrum of the transducer being used. If the quadrature decimation filter that follows the Input Filter is used to further reduce the sample rate, the Input Filter bandwidth should be set accordingly to prevent aliasing. If the `InputFilter` attribute is missing or set to empty ([ ]), VSX will provide a default set of coefficients for a band pass filter consistent with the `samplesPerWave` attribute setting.

The `Receive.callMediaFunc` is used only by the Simulator. Setting `Receive.callMediaFunc` to true (1.0) calls the Matlab function specified in `Media.function` before simulating the receive data.

The following table lists the center frequencies (demodFrequency) derived from the 250MHz master clock that are consistent with 4xFc sampling.

demodFrequency	decimSampleRate	ADCRate	250MHz/N	decimFactor
15.625 MHz	62.5MHz	62.5MHz	4	1
12.5	50	50	5	1
10.4167	41.667	41.6667	6	1
8.9286	35.7143	35.7143	7	1
7.8125	31.25	62.5	4	2
6.9444	27.7778	27.7778	9	1
6.2500	25.0000	50	5	2
5.6818	22.7273	22.7273	11	1
5.2083	20.8333	62.5	4	3
4.8077	19.2308	19.2308	13	1
4.4643	17.8571	35.7143	7	2
4.1667	16.6667	50	5	3
3.9062	15.6250	62.5	4	4
3.6765	14.7059	14.7059	17	1
3.4722	13.8889	41.6667	6	3
3.2895	13.1579	13.1579	19	1
3.1250	12.5000	62.5	4	5
2.9762	11.9048	35.7143	7	3
2.8409	11.3636	22.7273	11	2
2.7174	10.8696	10.8696	23	1
2.6042	10.4167	62.5	4	6
2.5000	10.0000	50	5	5
2.4038	9.6154	19.2308	13	2
2.3148	9.2593	27.7778	9	3
2.2321	8.9286	62.5	4	7
2.0833	8.3333	50	5	6
1.9531	7.8125	62.5	4	8
1.8939	7.5758	22,7273	11	3
1.8382	7.3529	14.7059	17	2
1.7857	7.1429	50	5	7
1.7361	6.9444	41.6667	6	6
1.6447	6.5789	13.1579	19	2
1.6026	6.4103	19.2308	13	3
1.5625	6.2500	50	5	8
1.4881	5.9524	41.6667	6	7
1.4205	5.6818	22.7273	11	4
1.3889	5.5556	27.7778	9	5
1.3587	5.4348	10.8696	23	2
1.3021	5.2083	41.6667	6	8
1.2755	5.1020	35.7143	7	7
1.2500	5.0000	25	10	5
1.2255	4.9020	14.7059	17	3
1.2019	4.8077	19.2308	13	4
1.1574	4.6296	27.7778	9	6
1.1364	4.5455	22.7273	11	5
1.1161	4.4643	35.7143	7	8
1.0965	4.3860	13.1579	19	3
1.0417	4.1667	25	10	6
0.9921	3.9683	27.7778	9	7
0.9766	3.9062	31.25	8	8
0.9615	3.8462	19.2308	13	5
0.9470	3.7879	22.7778	11	6

demodFrequency	decimSampleRate	ADCRate	250MHz/N	decimFactor
0.9191	3.6765	14.7059	17	4
0.9058	3.6232	10.8696	23	3
0.8929	3.5714	25	10	7
0.8681	3.4722	27.7778	9	8
0.8333	3.3333	16.6667	15	5
0.8224	3.2895	13.1579	19	4
0.8117	3.2468	22.7273	11	7
0.8013	3.2051	19.2308	13	6
0.7812	3.1250	25	10	8
0.7440	2.9762	20.8333	12	7
0.7353	2.9412	14.7059	17	5
0.7102	2.8409	22.7273	11	8
0.6944	2.7778	16.6667	15	6
0.6868	2.7473	19.2308	13	7
0.6793	2.7174	10.8696	23	4
0.6579	2.6316	13.1579	19	5
0.6510	2.6042	20.8333	12	8
0.6378	2.5510	17.8571	14	7
0.6250	2.5000	12.5	20	5

Table 3.3.1.1 Supported demodFrequencies and Sample Rates

3.3.2 Receive Profile Control Structures

The "RcvProfile" data structure gives the user a mechanism to control various settings associated with the TGC preamp and A/D converter in the receive signal path for each channel. Variables (listed in section 3.3.3.1 below) can be set as desired by the user in RcvProfile structures defined in the setup script. Any RcvProfile variable not explicitly set by the user will be assigned a default value by VSX.

There are three ways in which a user can employ the RcvProfile structure, to achieve the desired receive signal path settings for each acquisition event within their event sequence:

1. Use the default settings as assigned by VSX

If the user sees no need to change any of the default values that will be assigned by VSX, then a RcvProfile structure does not need to be defined in the user script at all. When VSX loads the script and sees there is no RcvProfile structure, it will create one and populate it with the default values assigned to each RcvProfile variable. When the HAL loads the script onto the HW system for execution, all receive channels will be programmed with the values from this default RcvProfile structure. Then every Receive Event in the acquisition sequence will use these same values.

2. Apply user-specified settings, with the same values for all Receive Events

If the user wants to change one or more of the RcvProfile variables from the default value that would be defined by VSX, simply define a RcvProfile structure in the SetUp script, and assign the values as desired. When VSX sees this user-defined RcvProflile structure it will simply use it as-is for sending the desired values to the HAL to program the HW. Any variables not specified by the user will be added by VSX, and assigned the associated default value. When the HAL loads the script onto the HW system for execution, all receive channels will be programmed with the values from this user-created RcvProfile structure. Then every Receive Event in the acquisition sequence will use these same values.

3. Set different user-defined values for different Receive Events within the Event Sequence

If the user wants to apply different receive path settings to different Receive Events in the Event sequence (for example, a script that interleaves imaging acquisition with color Doppler may want to use different settings for the Doppler and imaging events), then an array of multiple RcvProfile structures can be defined, with different values specified within each structure. Within each structure, VSX will automatically fill in default values for any unspecified variables. Then within the event sequence, the user can add "setRcvProfile" sequence control commands wherever desired. The argument for the setRcvProfile command will identify one of the RcvProfile structures, and the values from the selected structure will be loaded into the HW receive channels at that point in the event sequence. These values will then remain in effect for all subsequent Receive events, until another setRcvProfile command is encountered in the event sequence.

3.3.2.1 RcvProfile Variable Definitions

Descriptions are given below for each variable that is user-programmable within the RcvProfile data structure

- `RcvProfile.DCsubtract` is a string variable, which must be set to either 'on' or 'off'. It controls the DC offset subtraction feature of the A/D, to allow subtraction of a per-channel constant (stored in another per-channel HW register) from the receive A/D data samples of each channel. At system power-up, an automatic calibration sequence is used to determine the actual DC offset value produced by each channel's A/D converter. This value is saved in the per-channel register, so when DCsubtract is enabled the DC offset from each channel will be nulled to zero. If not specified by the user, VSX will set a default value of 'on' for DCsubtract.

- `RcvProfile.AntiAliasCutoff` is a double for which the allowed values are 5, 10, 15, 20, and 30, representing the cutoff frequency in MHz for the lowpass filter ahead of the A/D converter (5 is only available on systems built after approx. September 2015). The Vantage High Frequency System has the additional cutoff frequency values of 35 and 50 MHz. If not specified by the user, VSX will automatically set a default value which is the lowest of the available choices which is above the transducer upper bandwidth limit, Trans.Bandwidth(2). Note that if Trans.Bandwidth has not been set by the user, VSX will assign default values based on 60% of Trans.frequency.

- `RcvProfile.PgaGain` is a double to set the overall gain in dB of the preamp output buffer prior to the A/D. The only allowed values are 24 and 30 dB. If not set by the user, VSX will set a default value of 24 dB.

- `RcvProfile.PgaHPF` is a double that controls the integrator feedback path for the PGA, used to provide a high-pass frequency response and null the DC offset that would otherwise be present from the DC-coupled signal path. The only allowed values for this field are 80 and 0. `RcvProfile.PgaHPF` = 80 enables the integrator feedback path, resulting in a first order high-pass frequency response for the PGA with a breakpoint at approximately 80 KHz. This is the default system setting if this attribute is not set by user. `RcvProfile.PgaHPF = 0` disables the integrator, allowing the PGA to have a flat frequency response all the way down to DC. This may result in an unpredictable DC offset added to the signal.

- `RcvProfile.LnaGain` is a double to set the overall gain in dB of the fixed-gain low noise input amplifier stage. The only allowed values are 15, 18, and 24 dB. (The lowest setting is 12 dB instead of 15 on systems built before approx. September 2015.) If not set by the user, VSX will set a default value of 18 dB.

- `RcvProfile.LnaHPF` is a double controls the integrator feedback path for the LNA, used to provide a programmable high-pass frequency response and null the DC offset that would otherwise be present from the DC-coupled signal path. The only allowed values for this field are 200, 150, 100, 50 and 0. A value set to 200, 150, 100 and 50 enables the integrator feedback path, resulting in a first order high-pass frequency response for the LNA with a breakpoint at approximately 200 KHz, 150 KHz, 100 KHz and 50 KHz respectively. `RcvProfile.LnaHPF = 0` disables the integrator, allowing the LNA to have a flat frequency response all the way down to DC. This may result in an unpredictable DC offset added to the signal.

- `RcvProfile.LnaZinSel` is a double to select the feedback resistors used to enable the 'active termination' feature controlling the input impedance of the LNA input amplifier, as presented to the transducer connector. This variable must be set to an integer value in the range from 0 to 31. The value itself has no direct meaning (it represents a 5-bit binary value, where each bit switches one of 5 different resistor values in and out of the circuit). A value of 0 gives the lowest input impedance setting, of 115 Ohms. Input impedance increases with increasing values, up to the highest impedance with a value of 31 (the "high Z" state with no active feedback). For a given feedback resistance, the resulting input impedance will also be a function of the gain setting of the LNA, controlled by the LnaGain value described above. Note that the "high Z" input impedance is approximately 8,000 Ohms in parallel with 20 pf, when LnaZinSel is set to 31. If LnaZinSel is not specified by the user, the default value is 0. With the default LNA Gain of 18 dB, this results in an input resistance of 115 Ohms.

For Vantage system with the high frequency configuration, the HW design of the input feedback network has been modified to introduce a high-pass frequency response at the preamp input. If "4/3 sampling" (see section 7.2.3) is adopted for probes whose bandwidth extends above 25 MHz, the high pass filter must be introduced in to reject unwanted signals and noise from DC to Fs/2 before A/D conversion (because these frequencies would alias with the Nyquist band from Fs/2 to Fs that will be processed by the system when using 4/3 sampling). This highpass response is provided by the input impedance network, and is programmable through RcvProfile.LnaZinSel. A programmed value of zero results in a highpass cutoff at approximately 20 MHz, and the cutoff frequency moves progressively lower with increasing values of RcvProfile.LnaZinSel. A value of 31 disables the feedback network and thus also disables the highpass filter resulting in a flat frequency response at low frequencies with a 6000 Ohm input impedance (same as for the standard frequency configuration described above). If RcvProfile.LnaZinSel is not specified in the setup script, the system will assign a default value of 31 to disable the highpass filter. This is the preferred setting for lower frequency probes not using the 4/3 sampling feature.

3.3.3 TGC Waveform Objects

The TGC object defines the time-gain-compensation curve for the receive portion of the acquisition event. The structure for the object is as follows:

```
TGC =
      CntrlPts    [1x8 double]  Control points for specifying curve.
      rangeMax    double        No. of wavelengths to max range.
      Waveform    [1x512 double] 512 gain values for TGC curve.
```

To define a TGC waveform, the user specifies the `TGC.CntrlPts` array and `TGC.rangeMax` in the SetUp script. The system SW then uses these values to synthesize the `TGC.Waveform` array; this array is used to create the actual time-varying TGC control signal during the Receive event (in releases prior to 3.08, the `TGC.Waveform` array uses the uint16 datatype).

The `TGC.CntrlPts` array specifies the TGC level at 8 equally spaced points over the course of the Receive acquisition interval, starting at zero and ending at a depth set by `TGC.rangeMax`, in units of wavelengths of `Trans.frequency`. Therefore `TGC.rangeMax` is typically set equal to `Receive.endDepth`. The values used for `TGC.CntrlPts` must be in the range from 0 to 1023, where 0 represents minimum TGC gain and 1023 represents maximum TGC gain. The 40 dB TGC range of the Vantage system is roughly log-linear over this 0 to 1023 range of the control points. Each control point value is used to set the initial position of one of the TGC control sliders in the matlab GUI controls set up be VSX.

If desired, more than one TGC waveform can be created by defining an array of TGC structures. Then the `Receive.TGC` variable can be used to select the TGC waveform to be used for each individual Receive structure. When multiple TGC structures are defined, the GUI TGC control slidepots will default to TGC(1). With the 3.07 software release, other waveforms can be selected with the GUI control above the slidepots.

The media simulation software from the 3.0 release supports TGC for receive acquisitions, and applies the TGC gain curve to the simulated RF receive data.

3.4 Reconstruction Objects

Reconstruction objects are used only by the software processing, and are excluded from the Sequence Object structures loaded into the Vantage hardware. They describe the reconstruction methods required to form pixel information from the acquired receive data, and together with the `ReconInfo` objects, typically specify all the attributes needed to reconstruct a *full PData frame*.

3.4.1 Recon

The Recon structure provides the general attributes of the reconstruction, including the source and destination buffers to use. The attributes of the `Recon` object are given below:

```
Recon =
   senscutoff  [double]     Cutoff value for excluding channels from sum.
   pdatanum    [double]     Number of PData structure to use.
   rcvBufFrame [double]     If provided, overrides the frame no. in ReconInfo
   newFrameTimeout [double] (dflt 1000 msec) Time to wait for new frm
   IntBufDest  [1x2 double] [InterBuffer number, frame number]
   ImgBufDest  [1x2 double] [ImageBuffer number, frame number]
   RINums      [n double]   Row vector of n ReconInfo structure nos.
   rcvLUT      [uint16]     User provided receive Look Up Table (see text)
```

`Recon.senscutoff` is a value from 0.0 to 1.0 that sets the threshold of sensitivity below which an element is excluded from the reconstruction summation for a given pixel. A typical value would be 0.6, corresponding to a 4.44 dB loss in signal. Lowering the senscutoff value increases the effective number of elements in the receive aperture for the near field of the transducer, increasing lateral resolution. However, since elements are most sensitive to echoes from the normal direction to their surface, signals from large echoes at directions away from the reconstruction point will increase and contribute to clutter.

`Recon.pdatanum` specifies the number of the `PData` structure that defines the pixel locations of the reconstructed data.

Typically, the `RcvBuffer` containing the source RF data to be reconstructed is provided in the `Receive` objects referenced in the `ReconInfo` objects (described in the next section). `Recon.rcvBufFrame` is an optional attribute that when provided, over-rides the frame number specified by the `Receive` in the `ReconInfo` structures. If set to a non-zero, positive value, the value is the frame number that will be used for reconstruction. If set to -1, and running with the Vantage hardware, the processing determines the last acquisition frame transferred into the `RcvBuffer` used by this `Recon` structure, and processes this frame. This allows the same `ReconInfo` object(s) to be used for all the frames in the `RcvBuffer` processed by the running sequence. Obviously, in this case, all the frames processed must be able to use the same set of Receives. If the last frame transferred happens to be the same as the one used in the last reconstruction performed, the processing will wait up to a 1000 milliseconds for a new frame to be transferred, and if no new frame shows up, the old frame will be processed again. The `Recon.newFrameTimeout` attribute can be used to change this default timeout if needed. Using the -1 setting allows a sequence to implement asynchronous acquisition and processing, where the hardware acquires frames at one rate, and the processing

continually processes the most recently acquired frame at a different rate.

When not running with the Vantage hardware (simulate mode 1 or 2), the `rcvBufFrame = -1` setting will cause the reconstruction processing to always use the next frame from the previous frame reconstructed. When starting a sequence, exiting the freeze state, or transitioning from simulate mode 0 (normal acquisition) to simulate mode 2 (receive data loop), the first frame reconstructed in a Recon with `Recon.rcvBufFrame = -1` will be the next frame from `Resource.RcvBuffer.lastFrame`, which is typically the oldest frame in the buffer. This behavior allows the playback of receive data loops starting with the oldest frame in the buffer.

`Recon.IntBufDest` and `Recon.ImgBufDest` each specify the destination buffer and frame that will receive the reconstructed output, using a two element array. Depending on the reconstruction mode specified in the `ReconInfo` object, one or both buffer destinations may be required. For the frame number value in `Recon.IntBufDest(1)` or `Recon.ImgBufDest(1),` a value of -1 can be used, indicating that the next available frame number should be used for output. This can be used to fill a multi-frame buffer and provide a history of previous reconstructions that, for example, could be used for custom processing. When the end of the buffer is reached, the destination frame wraps to the beginning of the buffer. The `Resource.InterBuffer.lastFrame` and `Resource.ImageBuffer.lastFrame` attributes can be accessed after running a script to determine where the last frame processed resides in the IData/QData and ImgData buffers.

The `Recon.RINums` attribute is a row vector that specifies the `ReconInfo` structure indices associated with this reconstruction. Each `ReconInfo` object contains information on how to reconstruct pixels for a specific pixel data region. The individual pixel regions are typically defined by the ‘computeRegions’ function. Starting with the 3.0 software release, the pixels for each `Region` are automatically divided up into subsets for processing by the multiple processing threads of the reconstruction routine.

3.4.2 ReconInfo Objects

For each Recon object there is an associated set of ReconInfo objects. The set of ReconInfo objects should be unique for each Recon, even though in some cases, it would appear that the structures are identical. (While the visible attributes may be identical, there are attributes added during run time that depend on the parent Recon structure.) Each of the numbers in the `Recon.RINums` list described above is the number of a specific `ReconInfo` object. These objects provide the detailed information on how to process a Region of the pixel data, and so there are at least as many `ReconInfo` structures as there are `PData.Regions`. In situations where multiple reconstructions are combined to produce a result, there could be multiple `ReconInfo` structures for each region. The attributes of a `ReconInfo` object are given below.

```
ReconInfo =
     mode        [string]    Numeric or string representation
     txnum       [double]    Number of transmit object used for acquisition.
     rcvnum      [double]    Number of receive object used for acquisition.
     regionnum   [double]    Number of Region structure for pixel data.
     pagenum     [double]    Indicates page no. for multi-page InterBuffer
     normPower   [double]    Optional normalization power (see text).
     normWeight  [double]    Optional normalization weight (see text).
     scaleFactor [double]    Optional scale factor for all modes.
     Pre         [string]    'clearInterBuf','clearImageBuf','interleaveRF'
     Post        [string]    'IQ2IntensityImageBuf','…Add','…Mul'
     threadSync  [double]    default=0, 1=>sync after recon w this RI
     LUT         [int32]     user provided region Look Up Table (see text)
```

`ReconInfo.mode` specifies the mode of the reconstruction. It is typically specified with a string, but for backwards compatibility with pre 3.0 releases, it can also be specified with a number. In the following descriptions, the `InterBuffer` and `ImageBuffer` are those specified in the parent `Recon` structure. The currently supported options are:

mode		Destination Buffer(s)
0	'replaceIntensity'	ImageBuffer, (opt. InterBuffer)
1	'addIntensity'	ImageBuffer
2	'multiplyIntensity'	ImageBuffer
3	'replaceIQ'	InterBuffer
4	'accumIQ'	InterBuffer
5	'accumIQ_replaceIntensity'	InterBuffer, ImageBuffer
6	'accumIQ_addIntensity'	InterBuffer, ImageBuffer
7	'accumIQ_multiplyIntensity'	InterBuffer, ImageBuffer
8	'replaceIQ_normalize'	InterBuffer
9	'accumIQ_normalize'	InterBuffer
10	'replaceIQ_pages'	InterBuffer w no. of chnls pages
11	'accumIQ_pages'	InterBuffer w no. of chnls pages

Using modes `'replaceIQ'` and `'accumIQ'`, it is possible to accumulate reconstructed I,Q data in an InterBuffer for multiple acquisition events, and compute the magnitude of the sum on the last reconstruction, replacing, adding or multiplying with data in the output ImageBuffer (modes 5, 6 or 7). For normalizing the reconstructed data, based on the number of channels contributing to the sum, an internal secondary channel count

buffer with the same size as the InterBuffer or ImageBuffer is used to stored the accumulated channel count over multiple reconstructions. Modes 5, 6, 7, 8 and 9 use this accumulated channel count to normalize the magnitude (5, 6, 7) or IQ data (8, 9).

The last two reconstruction modes, '`replaceIQ_pages`' and '`accumIQpages`' allow the user to experiment with different methods of combining the delayed individual channel data. Each channel, representing a transducer element signal, is delayed and its signal value written to a separate page in the InterBuffer. The InterBuffer must be defined with the same number of pages as there are channel. An external function can then be used to combined the delayed signals using different methods. Simply summing the I and Q signals for all pages for a pixel will produce the same result as modes 3 or 4 ('`replaceIQ`' or '`accumIQ`').

`ReconInfo.txnum` specifies the index of the `TX` object used for transmit. This object is needed to determine the origin and characteristics of the transmit beam.

`ReconInfo.rcvnum` specifies the index of the `Receive` object used for acquisition. The `Receive` object contains the information on how the RF data was acquired, as well as the storage location of the data. The reconstruction software is limited to processing a depth of 4096 samples, so (`Receive.endSample - Receive.startSample`) must be less than 4096 (These attributes are computed when a setup script is run with VSX).

`ReconInfo.regionnum` – The number of the region that is to be reconstructed with this `ReconInfo` object. Each `ReconInfo` object specifies the reconstruction parameters for one and only one region of the `PData` structure specified in `Recon.pdatanum`. The number 0, which in software releases prior to 3.0 meant 'all Regions', is not permitted, as additional `ReconInfo` structures are no longer automatically created.

The regionnum attribute references a `PData.Region` structure that defines all of the pixels in the region. If there is no `TX.TXPD` data in the `TX` structure referenced by the `ReconInfo.txnum` specification, all of the `Region` pixels will be reconstructed. If `TXPD` data exists, it will be used to qualify which pixels in the `PData.Region` specification will be reconstructed. In this case, only the qualified pixels in the `Region` will receive reconstruction output. Non-qualified pixels will either not be touched or will have zeroes written to them, depending on the reconstruction mode.

`ReconInfo.pagenum` - This attribute is only needed when reconstructing to an InterBuffer that has multiple pages. A multi-page InterBuffer is typically used for color Doppler reconstructions, where an ensemble of acquisitions are needed to capture the Doppler information. Each acquisition can then be reconstructed to a separate page of the InterBuffer.

`ReconInfo.normPower` and `ReconInfo.normWeight` - These attributes, when provided, provide normalization constants used in reconstruction modes that write to an ImageBuffer. For the arithmetic mean, one would use '`replaceIntensity`', followed by multiple '`addIntensity`' reconstructions, or for synthetic apertures based on multiple acquisitions of I,Q data, '`accumIQ_replaceIntensity`', followed by multiple '`accumIQ_addIntensity`' reconstructions. Each reconstruction that adds to the sum can use `normWeight` to normalize its contribution. The result of three reconstructions where one wants to take a weighted arithmetic mean is then:

```
a * ReconInfo(1).normWeight + b * ReconInfo(2).normWeight + ...
     c * ReconInfo(3).normWeight
```

where `ReconInfo(1).mode = 'replaceIntensity'`
`ReconInfo(2).mode = 'addIntensity'`
`ReconInfo(3).mode = 'addIntensity'`

For equally weighted contributions, the `normWeight` values would each equal 1/3.

The `normPower` constant is used to normalize the products of a geometric mean, by taking the nth root of each product of n intensity values. (The geometric mean is generally preferred for averaging ultrasound data over the arithmetic mean, since it results in darker fluid filled areas, such as cysts.) The constant is expressed as a fractional exponent value. For the geometric mean, one would use mode `'replaceIntensity'`, followed by multiple mode `'multiplyIntensity'` reconstructions, or for synthetic apertures, mode `'accumIQ_replaceIntensity'`, followed by multiple mode `'accumIQ_multiplyIntensity'` reconstructions. For example, if the reconstruction consists of three intensity reconstructions multiplied together, the `ReconInfo` structures used in the set can set the `normPower` to 1/3 to apply the cube root to each of the multiplicands (a^0.333 * b^0.333 * c^0.333 = (a*b*c)^0.333).

These constants must be provided by the user in the setup script, so it is up to the user to set the appropriate values for the method of acquisition and reconstruction. If these attributes are not provided, default values of 1.0 for `normWeight` and `normPower` are applied. For an example script that uses multiplicative averaging for spatial compounding, see the SetUpL11_4vFlashAngleSC.m script.

`ReconInfo.scaleFactor` - This constant can be provided to scale the output of any reconstruction mode, since it is applied to the I,Q reconstruction output before any additional output processing (such as computing the magnitude or accumulating to an InterBuffer). It's main use is for weighting multiple synthetic aperture acquisitions which are to be combined in an InterBuffer. For example, with a 7 angle plane wave transmit reconstruction, one could weight steered angles higher than the straight ahead direction, since the magnitude of the transmit wavefront decreases with increasing steering angle, due to the element directivity function. The resulting 7 angle reconstruction accumulation can then be combined with other reconstructions outputs by weighting using `ReconInfo.normWeight` or `ReconInfo.normPower` as it is written to an ImageBuffer. The default value of `ReconInfo.scaleFactor`, if not provided, is 1.0.

The `Pre` attribute is used to specify a function that executes prior to the reconstruction action, while the `Post` attribute specifies a function that executes afterwards. The `'clearInterBuf'` and `'clearImageBuf' Pre` functions clear the entire `PData` pixel region for the destination frame. These functions are required if the reconstruction output goes only to a `Region` that is a subset of the pixels in the entire `PData` array, since pixels not written to may contain data from previous reconstructions. If the reconstruction output goes to the entire `PData` array, the `'replaceIQ'` or `'replaceIntensity'` reconstruction modes can be used instead.

The `Post` functions, `'IQ2IntensityImageBuf'`, `'IQ2IntensityImageBufAdd'` and `'IQ2IntensityImageBufMul'` are used when multiple IQ reconstructions over partial `Regions` of the `PData` array are combined in the InterBuffer, as in the case of multiple overlapping wide beams. In this case, the summed IQ data is accumulated in the `InterBuffer` using reconstruction mode `'accumIQ'` and after the last IQ reconstruction, the entire IQ frame is detected to intensity data in the destination

ImageBuffer using the Post function. Use of a 'accumIQ_replaceIntensity' mode reconstruction (reconstruct IQ, accumulate and detect) for the last reconstruction in the frame is not equivalent to using the 'IQ2IntensityImageBuf' Post function if the region of the last reconstruction is a subset of the PData array, since only the subset of pixels would be processed. 'IQ2IntensityImageBufAdd' and 'IQ2IntensityImageBufMul' (analogous to mode 'accumIQ_addIntensity' and mode 'accumIQ_multiplyIntensity') can be used to add or multiply the intensity data with data already in the output buffer. In these Post routines, the ReconInfo attributes of 'normWeight' or 'normPower' can be used to normalize the output values. If no Pre or Post action is needed, these attributes can be missing or empty.

The 'threadSync' attribute, when set to 1 (true), specifies that the reconstruction threads should be synchronized after the reconstruction of the region specified in the ReconInfo. Thread synchronization is required when a reconstruction has multiple ReconInfo structures with unique regionnums to be processed and the pixels in the ReconInfo region overlap with pixels in one or more ReconInfo regions yet to be processed. Without thread synchronization, different threads might be processing different regions with shared pixels at the same time, writing to the same pixels in a race condition. The last ReconInfo in a multiple ReconInfo reconstruction always performs thread synchronization, even if the threadSync attribute is 0. Thread synchronization slows the reconstruction processing and should be used only when needed, which is mainly when processing wide beam scans with overlapping regions. Thread synchronization is not needed for line mode reconstructions, where there are a large number of non-overlapping Regions. It is also not needed when multiple reconstructions are performed on all the pixels of the same Region or the entire PData space, as in the flash and flashAngles scripts. If the user is uncertain as to whether threadsync is required, a check can be made of reconstruction of a static image in simulate mode. If no pixels exhibit flashing intensity behavior, then threadsync is not likely to be required.

3.4.3 User Provided Reconstruction Look-Up-Tables

Normally, the reconstruction software automatically calculates Look-Up-Tables to specify the reconstruction processing. These tables are calculated on the first call to the reconstruct function and are hidden from the user. Starting with software version 4.2, the user can specify custom LUTs for reconstruction in their setup script by providing the tables in the Recon.rcvLUT and ReconInfo.LUT structure attributes. If these attributes are missing or empty, the LUTs will be computed by the processing software. The tables, if provided, are read into the software during initialization, and the internal table computation is disabled. If the tables need to be modified during run-time, an external function can be provided that recalculates the tables and updates the Recon structures. A script that provides custom LUTs and modifies either the PData definition or changes the transmit parameters must also modify the LUTs accordingly. Only the size and datatype of the user provided LUTs are checked, and an error is generated if these are incorrect.

The receive LUT for a specific reconstruction is computed in `Recon.rcvLUT`. The rcvLUT has an entry for each pixel in the PData array, in order of the linear address of the pixel/voxel, where each entry has `Trans.numelement` unsigned words (uint16). There is an unsigned word for each transducer element that specifies the path length in `Trans.frequency` wavelengths from the pixel to each transducer element. The path length should include the `Trans.lensCorrection` path length. A zero path length means that the element is not used in the reconstruction. The path length is specified as an unsigned word value with 12 bit integer and 4 bit fraction parts, as shown below.

pathLength = uint16(round((double)PathLength) * 16))

The size of the `Recon.rcvLUT` in unsigned words is then given by the number of pixels/voxels in the PData array times the number of transducer elements.

The `ReconInfo.LUT` has three int32 values for each pixel to reconstruct, which includes all the pixels in the Region specified by the regionnum attribute of the ReconInfo. The first value is an int32 with the linear address (from zero) of the pixel to be reconstructed in the PData array. The second value is an int32 that specifies the weight for the pixel, using a 24 bit integer part and an 8 bit fractional part. The weight can be positive or negative, and defined as follows:

weight = int32(round((double)Weight * 256))

The third value is the path length in `Trans.frequency` wavelengths from the origin of the beam on the transducer to the pixel location, expressed as an int32, computed as follows:

pathLength = int32(round((double)PathLength) * 16))

In the case of an unfocused beam, the transmit path length is typically the shortest distance from the pixel to the transmitting aperture of the transducer. The transmit path length actually represents the time of arrival of the transmit pulse at the pixel location, and so it should include the transmit delay time as well, expressed as a length in wavelengths. In addition, the arrival time is assumed to be the arrival time of the peak of the transmit waveform, so the path length should also included the value of `TW.peak`. (`TW.peak` is computed by the computeTWWaveform utility function.) The size of the `ReconInfo.LUT` array in int32 values is then given by the number of pixels in the PData.Region (specified by `ReconInfo.regionnum`) times 3.

3.5 Process Objects

`Process` objects are used to describe the type of processing to apply to acquired or reconstructed data. Process objects are used only by the system software - the Vantage hardware sequencer ignores these items in the `Event` list. The `Process` object is a structure that specifies a processing task to be executed at a specific point in the event list.

```
Process =
     classname   [string]   Class of processing objects.
     method      [string]   Method to use for class.
     Parameters  [string,value,string,value,…] An array of property, value
                            pairs that define the various attributes of the
                            processing object. In Matlab, this construct
                            is a cell array, consisting of strings and
                            values that possibly have different datatypes.
```

3.5.1 Image Display Object

A common processing task is displaying an image. While the output of the reconstruction processing can be pixel intensity data, which in itself defines an image that could be viewed on the display, there is typically additional processing that needs to be performed before rendering the image data to the display. For example, the intensity data is usually compressed in dynamic range and scaled to the proper maximum numeric values for the output display. In addition, persistence processing is often used to average the new frame with previous frames of data. In cases where both intensity data and Doppler velocity data are to be presented on the same display, the Doppler data must either replace or be combined with the intensity data, and a suitable color map chosen to display the information. The processing object associated with these actions is an `Image` Object, which is defined by the following structure:

```
Image =
     imgbufnum   [double]  ImageBuffer number to use for source.
     framenum    [double]  Source frame number (use -1 for last)
     pdatanum    [double]  No. of PData structure
     srcData     [string]  'intensity'(default),'signedColor','unsignedColor'
     grainRemoval [string] 'none'(default),'low','medium','high'
     persistMethod [string] 'none'(default), 'simple', 'dynamic'
     persistLevel [double] Amount of persistence, range 0 – 99
     interpMethod [string] Interpolation method ('4pt' interp(default)).
     processMethod [string] 'none'(default),'reduceSpeckle1','reduceSpeckle2'
     averageMethod [string] 'none'(default),'runAverage2','runAverage3'
     pgain       [double]  Processing gain.
     reject      [double]  Low level reject (0-100)
     compressMethod [string] 'power' or 'log'
     compressFactor [double] (0-100)
     mappingMethod [string]  'full'(default),'lowerHalf','upperHalf'.
     threshold   [double]  0-255. Compare against pixel value
                              to determine whether to overwrite.
     gamma       [double]  compression applied to gray scale colormap
     blacklevel  [double]  reject applied to gray scale colormap
     display     [double]  0-no display, 1-display.
     displayWindow [double] Dest. display window (default is 1).
```

When the class name specified in the `Process` object is 'Image', an `Image` object is created during VSX initialization to control the processing. In Matlab, the Image object's attributes would be given in a cell array of property/value pairs using the `Properties.Parameters` structure. Below is an example of the Matlab code needed to define a set of Image properties.

```
Process(1).classname = 'Image';
Process(1).method = 'imageDisplay';
Process(1).Parameters = {'imgbufnum',1,… % ImageBuffer to process.
                         'framenum',1,…
                         'pdatanum',1,…
                         'compressMethod','power',…   % power comp.
                         'compressFactor', 40,…
                         'pgain',1.0,…   % image processing gain
                         'reject',3,…    % reduce intens. below this index
                         'mappingMode','full',…
                         'displayWindow',1,…    % Dest. display
                         'display',1};          % display image
```

As shown above, not all processing attributes are required to be defined, as undefined attributes will default to appropriate values.

The rendering process consists of processing the source data in the specified `ImageBuffer`, according to the attributes of the Image structure. The resulting pixel data is written to a DisplayData array that is associated with a `DisplayWindow`. If the `Image.display` attribute is true (1), the DisplayData array is then copied to the display window's CData structure and the image is rendered to the screen. By setting the `Image.display` attribute to false (0), the DisplayData array is computed, but no image is rendered to the display. This feature is generally used when combining data from more than one `ImageBuffer`, as in color Doppler imaging. The first `ImageBuffer` is processed into the DisplayData array without rendering to the screen, and then the 2nd `ImageBuffer` is processed and combined with the same DisplayData array. After all the `ImageBuffers` are combined in the DisplayData array, the final image processing sets the `Image.Display` attribute to true (1), causing the image to display on the screen.

There are four temporary buffers for display data. Buffer ID0 acts as a temporary buffer for some of the processing routines, such as the output of the interpolation function. Buffers ID1 through ID3 act as a history buffer for processed image data frames, allowing frames processed serially to be combined for display.

The `'grainRemoval'` attribute for intensity data selects from three different 3x3 matrix filters to remove singular low and high intensity values that differ significantly from their surrounding neighbors. The filters are `'low'`, `'medium'`, and `'high'` and offer increasing levels of grain removal. These filters operate on the data in the ImageBuffers, and are thus sensitivity to the pixel density of these buffers. The higher the pixel density, the less effect these filters will have.

The `'persistMethod'` has three options - `'none'`, `'simple'` and `'dynamic'`. The `'dynamic'` option requires specifying a vector for the 'persistLevel' of four values. Each value sets the persistence according to the amount of change in the intensity value at a pixel from frame to frame. The first value sets the persistence for a change that is less than one quarter of the full scale intensity. The second value sets the persistence for a

change that is between one quarter and one halve of the full scale intensity, and so on. These values can be adjusted in real-time using the Image Processing tool (PTool).

The `'interpMethod'` is now specified as a string, with '4pt' being the currently supported method. The 4pt interpolation has been significantly improved over previous releases and now does a much better job with coarse pixel densities. The interpolated output for intensity data is sent to the current frame in the history buffer if there is no process method specified; otherwise, it is sent to the ID0 temporary buffer.

The `'processMethod'` is used to apply spatial filters on the interpolated ImageBuffer frame. These filters are 3x3 matrix filters that modify the central pixel according to functions applied to the surrounding neighbors. The current filters are `'reduceSpeckle1'` and `'reduceSpeckle2'`. These filters operate on the line pairs of the surrounding pixels and clamp the central value to the difference between a selected line pair. `'reduceSpeckle1'` finds the line pair with the smallest absolute difference and uses it to clamp the central value. `'reduceSpeckle2'` sorts the minima and maxima of all the line pairs and clamps the central value to be between the maximum of the minima and the minimum of the maxima. The `processMethod` takes as input the ID0 buffer and outputs to the current frame in the history buffers.

The `'averageMethod'` has three options: `'none'`, `runAverage2'` and `'runAverage3'`. The latter two compute the running average of the last two or three frames, respectively. The original frames in the history buffer are not modified, as the average is computed to DisplayData buffer.

The `'pgain'`, `'reject'` and `'compressMethod'` attributes are now applied to intensity data just before the data are written to the DisplayData buffer. First the `'pgain'` scale factor is applied, with the maximum data value limited to maxV, where maxV is a fixed value which is the largest value expected from the normalized reconstruction intensity output. The `'reject'` attribute, which can range from 0 to 100, sets minV as a percent of maxV/4. The value minV is then subtracted from the scaled data so that the data now ranges between 0 and maxV-minV.

The `'compressMethod'` now has two options - `'power'` and `'log'`. The `'power'` option computes the intensity raised to a fractional power, set by the 'compressFactor'. A `'compressFactor'` of 20 gives a power of 1.0 for a linear output, while a `'compressFactor'` of 40 gives a power of 0.5 for a square root compression. Higher `'compressFactor'` values provide smaller powers, resulting in more compression. The `'log'` option provides a logarithmic compression curve at different scale factors, depending on the `'compressFactor'` value. Higher values of the `'compressFactor'` result in a more rapid rise to the log curve, raising the brightness of low level level intensity values.

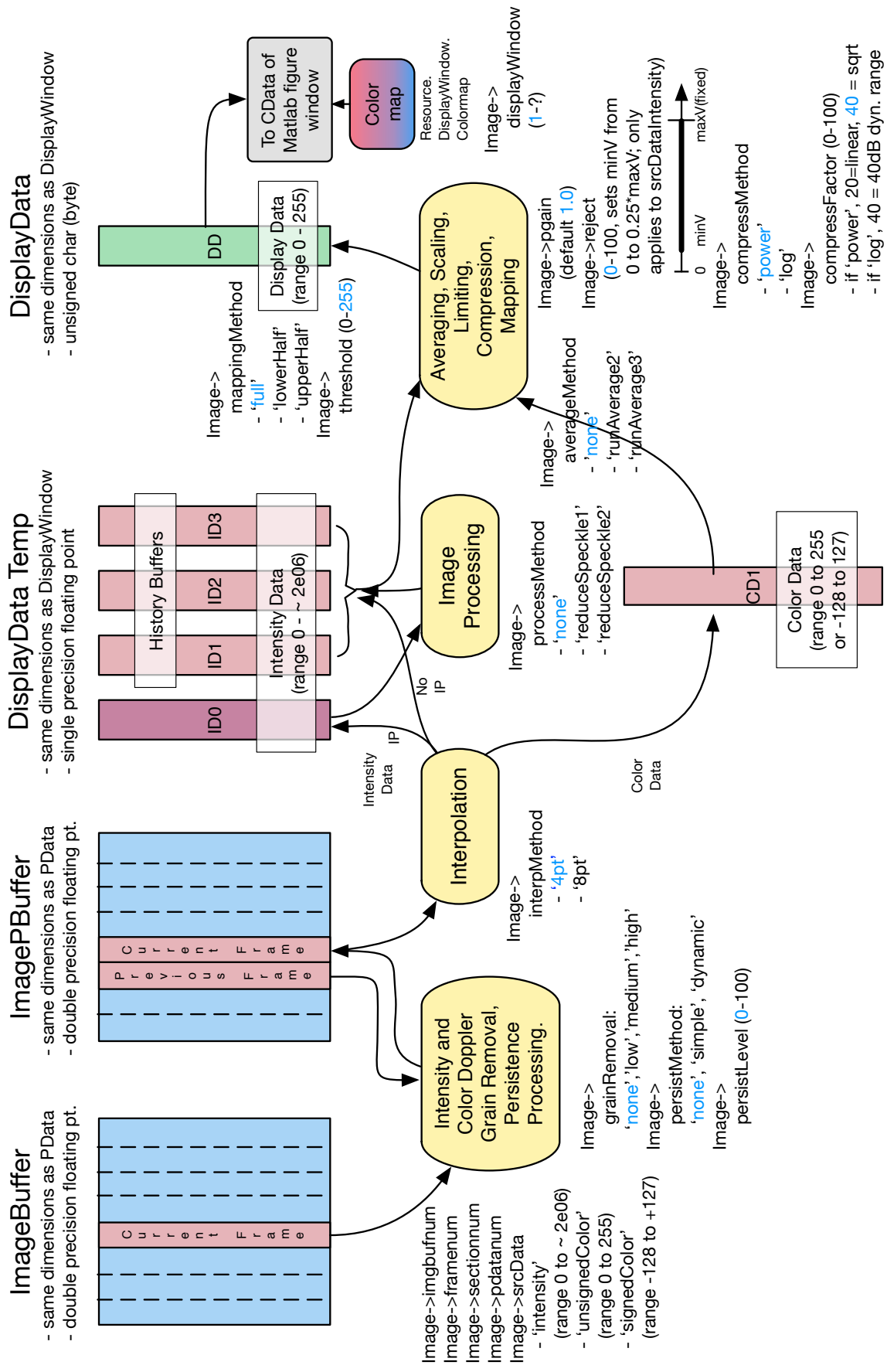
The greyscale curve (or colormap) of the Matlab figure window is no longer modified by compress and reject sliders in vsx_gui, as these functions are now specified by the `'compressFactor'` and `'reject'` attributes of the Image structure. If the figure window colormap requires modification, new maps can be loaded by setting the `'colormap'` attribute of the appropriate `Resource.DisplayWindow`, or by using the figure window's colormap editing tool in Matlab. A Verasonics colormap tool has been developed that a sonographer can use in the final optimization of an image.

There may be some applications where the user wants to use a custom display routine, rather than the Matlab figure window. Normally, the runAcq processing code calls the VsDisplay function in the Utilities directory when image display output is generated, passing the DisplayData array to be copied to the CData of the `DisplayWindow`. The user can add to or replace code in the VsDisplay.m function to perform their own display function.

The external display function could also be coded as a Matlab mex file if desired. When using an external display function, the attributes of the `Resource.DisplayWindow` referenced by the Image parameter, `'displayWindow'` must still be defined in the Resource definition, since the DisplayData delivered to the UDisplay function will be defined by these attributes.

The `imageDisplay` data flow and processing functions are shown graphically in Fig. 3.5.1.1 below.

Fig. 3.5.1.1. Post-Reconstruction Image Processing



3.5.2 Doppler Process Object

For Doppler processing of reconstructed IQ data, a `Doppler` Object must be defined, using the following structure.

```
Doppler =
     IntBufSrc      [1x2 double] InterBuffer number and frame
     SrcPages       [1x2 double] Start page num., number of pages.
     IntBufDest     [1x2 double] InterBuffer num. and frame (for complex data)
     ImgBufDest     [1x2 double] ImageBuffer number and frame
     pdatanum       [double]     Number of PData structure
     prf            [double]     PRF in Hz.
     WallFilter     [string]     Method of wall filtering.
     pwrThreshold   [double]     % of max power threshold
     maxPower       [double]     Max power level for calculations & display
     postFilter     [double]     Method of filtering estimate data (0=none)
```

When the class name specified in the `Process` object is 'Doppler', a `Doppler` object is created during run time to control the processing. As for the Image object, the Doppler object's attributes should be given in a cell array of property/value pairs using the `Properties.Parameters` structure parameter. Below is an example of the Matlab code needed to define a set of Doppler properties for Color Flow Imaging.

```
Process(2).classname = 'Doppler'; % process structure for Doppler ensemble
Process(2).method = 'computeCFIFreqEst'; % compute frequency estimates
Process(2).Parameters = {'IntBufSrc',[2,1],... % buf[num, frm] of interbuffer
                         'SrcPages',[3,ne-2],... % start page, num pages
                         'ImgBufDest',[2,-1],... % buf[num, frm] of dest.
                         'pdatanum',2,...        % number of PData structure
                         'prf',dopPRF,...        % Doppler PRF in Hz
                         'wallFilter','regression',... % quadratic regression
                         'pwrThreshold',0.9,... % reject est. below threshold
                         'maxPower',50,... % reject est. above maxPower
                         'postFilter',1};
```

For 2D image formats, the various frames of I,Q data included in an ensemble should be reconstructed into separate pages of an complex format Intermediate buffer. An example source buffer definition might be as follows:

```
Resource.InterBuffer(m).datatype = 'complex';
Resource.InterBuffer(m).numFrames = 1; % Only one frame needed.
Resource.InterBuffer(m).rowsPerFrame = 2048;
Resource.InterBuffer(m).colsPerFrame = 64;
Resource.InterBuffer(m).pagesPerFrame = numPRIs; % numPRIs pages per ensemble
```

In the 2D case, the Doppler processing starts with the `SrcPages(1)` page number in InterBuffer number `IntBufSrc(1)` at frame `IntBufSrc(2)` and increments by page number for `SrcPages(2)` pages.

Doppler processing now processes all the Regions of the PData object. If a Region is not specified in the PData definition, a single Region is created by the computeRegions function that contains all the pixels in PData. Each Region is processed by dividing its pixels by the number of processing threads, so that each thread processes an equal number of pixels. This occurs automatically (there is no dividing up of a Region by the computeRegions function).

When reconstructing the Regions of the Doppler PData structure, one should define n ReconInfo structures for each Region, where n is the number of acquisitions in the Doppler ensemble, with the ReconInfo.pagenum attribute specifying the InterBuffer page number that will receive the I,Q data. Doppler Regions shouldn't overlap, allowing processing of all Regions without the need for synchronizing threads.

For 3D PData formats, the Doppler.Srcsectionnums attribute is no longer required, as the sections to be processed can be defined in the PData.Region structure, which simply provides the linear addresses of the pixels to be processed. Use the Pyramid, Cone or Slice shape definitions to specify the region of pixels to be processed. Unfortunately, older scripts that use the Doppler.Srcsectionnums attribute are not supported, and an error message is generated.

The new attribute, `IntBufDest`, allows output of the Doppler I and Q autocorrelation estimates to a complex single InterBuffer frame that will allow more sophisticated post processing of Doppler velocity data. This destination is used when the attribute is defined, otherwise the `ImgBufDest` output destination is used.

3.5.2.1 Choosing a Wall Filter

The method of wall filtering for Doppler processing is specified as a string. The current choices include 'none' and 'regression', as well as settings for FIR filters and two sets of matrix filter designs.

'none' – no wall filtering. This setting could be used for tissue Doppler sensing.
'regression' – selects a 2nd order (quadratic basis) regression filter. This filter works best for short ensemble lengths (under 16 PRIs).

FIR Filter Settings - these select among 3 FIR filter choices. FIR filters work best for longer ensemble lengths (> 19 PRIs) since they reduce the number of filtered samples in the ensemble by the filter order. Thus a 12-tap 32 PRI ensemble will only have 21 PRIs after filtering. If a FIR filter is specified with less than 20 PRIs, excluding dummies, a run-time error is generated. The filter names and characteristics are:

- '**FIRLow**' - 13 taps, -60 dB below 0.003 PRF, -3dB @ 0.08 PRF
- '**FIRMedium**' - 12 taps, -63 dB below 0.017 PRF, -3dB@ 0.15 PRF
- '**FIRHigh**' - 13 taps, -65 dB below 0.035 PRF, -3dB @ 0.18 PRF

Matrix Filter Settings (General Flow Conditions) - Under a restricted set of ensemble lengths, six choices of matrix filter coefficients provide a range of high-pass attenuation and passband behavior. These implement time-varying impulse response filters. Each of the six filtering choices is available for even ensemble lengths over the range of [12,24] (net, not including precursor "dummy" pulses). The filter names are:

- '**TVFiltVLow**'
- '**TVFiltLow**'
- '**TVFiltMedLow**'
- '**TVFiltMed**'
- '**TVFiltMedHigh**'
- '**TVFiltHigh**'

The table below describes the characteristics corresponding to each filter name.

# PRIs	wallfilter =	cutoff freq.	reject BW	Description
12	'TVFiltVLow'	Low	narrow	quadratic regression
12	'TVFiltLow'	Low	wider	-70 dB equiripple stopband
12	'TVFiltMedLow'	Med	narrow	cubic regression
12	'TVFiltMed'	Med	wider	-70 dB equiripple stopband
12	'TVFiltMedHigh'	High	narrow	quartic regression
12	'TVFiltHigh'	High	wider	-70 dB equiripple stopband
14	'TVFiltVLow'	Lowest	narrow	quadratic regression
14	'TVFiltLow'	Low	wider	-70 dB equiripple stopband
14	'TVFiltMedLow'	Med. Low	...	-70 dB equiripple stopband
14	'TVFiltMed'	Medium	...	-70 dB equiripple stopband
14	'TVFiltMedHigh'	Med Hi	...	-70 dB equiripple stopband
14	'TVFiltHigh'	High	widest	-70 dB equiripple stopband
16	'TVFiltVLow'	Lowest	narrowest	-60 dB equiripple stopband
16	remainder	increasing	increasing	-70 dB equiripple stopband
18 - 24, even	all	increasing	increasing	-65 dB or better equiripple S.B.
odd	N/A	N/A	N/A	Invalid
< 12	N/A	N/A	N/A	Invalid
> 24	N/A	N/A	N/A	Invalid

Matrix Filter Settings (Weak/Low Flow Conditions) - Under weak Doppler signal flow conditions, a restricted set of ensemble lengths between 20 and 60 permits six choices of appropriate matrix filter coefficients. These have an equiripple stopband of -95 dB, and are optimized for detecting weak flow signals with ensemble lengths of { 20, 24, 26, 28, 30, 34, 36, 40, 44, 48, 52, 56, 60 }, not including precursor pulses. The Weak Flow / Low Flow filter names are:

- '**WeakFlowVLow**' (lowest stopband cutoff frequency)
- '**WeakFlowLow**'
- '**WeakFlowMedLow**'
- '**WeakFlowMed**'
- '**WeakFlowMedHigh**'
- '**WeakFlowHigh**' (highest stopband cutoff frequency)

Note that at higher ensemble lengths, the Weak-Flow Matrix Filter transition band becomes steep, while the stopband is wide enough to be effective even at low cutoff settings. This allows lower-cutoff filters to use a high PRF, which generally improves detection performance of the autocorrelation processing currently implemented in the Doppler Process Object. The Weak/Low Flow filter coefficient performance is documented in “CDIWeakFlowFilterResponses.pdf”, stored in the Documentation subfolder.

3.5.3 External Process Object

For situations where the user wants to provide their own processing routines, the run time software provides a mechanism to support this. At the appropriate point in the event list, the user places an event with a Process structure that identifies 1) the source buffer, frame number, section number and page number needed (the page number attribute only can be used with a `srcbuffer` of type `'inter'`); 2) the output buffer frame, section and page number to send the processed data to; 3) the name of the external processing routine, which should be defined as a normal Matlab function (or mex file). Below is the format for the Process structure:

```
Process(1).classname = 'External';   % Identifies the processing as external.
Process(1).method = 'processFunctionName';
Process(1).Parameters = {'srcbuffer','receive',…
                         'srcbufnum',1,…    % no. of buffer to process.
                         'srcframenum',1,…  % starting frame no.
                         'srcsectionnum',1,…
                         'srcpagenum',1,…
                         'dstbuffer','image',…
                         'dstbufnum',1,…
                         'dstframenum',1,…
                         'dstsectionnum',1,…
                         'dstpagenum',1};
```

The choices for `srcbuffer` and `dstbuffer` are `'receive'`, `'inter'`, `'image'` and `'imageP'` or `'none'`. The option of `'none'` is chosen for when no input and/or output buffer is needed. A missing specification or value of zero for either `srcframenum, srcsectionnum` or `srcpagenum` means that all frames, all sections or all pages, respectively, are to be transferred as input. Currently, only contiguous memory transfers are supported, so it is not allowed to transfer a single section or a single page from all frames. When `'receive', 'inter', 'image'` or `'imageP'` are specified for srcbuffer, a value of -1 is permitted for srcframenum, which acts to specify the last frame processed into the buffer.

For dstbuffer = 'receive', dstframenum can be -1, 0 or a specific frame number. A value of -1 writes the receive data to the last frame transferred to the buffer, which would be the same frame as provided as input with srcbuffer = 'receive' and srcframenum = -1. A dstframenum of zero with dstbuffer = 'receive' writes to the entire buffer.

Fro dstbuffer = 'inter', dstframenum can be -1 or a specific frame number. The -1 value writes the data to the next available frame in the buffer from the last frame written. Transferring the full buffer (dstframenum = 0) is currently not supported.

For dstbuffer = 'image' or 'imageP', only single frame transfers are allowed. In this case, dstframenum can be -A value of -1 or -2 is also permitted for dstframenum if `'image'` or `'imageP'` are specified for dstbuffer. Both options write to the last frame in the destination buffer, but -1 will increment the last frame pointer and the Receive.ImageBuffer.lastFrame attribute before writing. When `'image'` or `'imageP'` are specified for `dstbuffer`, the `dstpagenum` attribute is ignored.

When processing data externally, the buffer(s) used for input should be considered to be read only; modifying these data between successive calls to 'runAcq' may lead to a

software failure, since runAcq reads and stores the address of these buffers once at initialization time, and writing to a Matlab array usually results in Matlab making a copy of the original array, then modifying it and storing with a different address. In addition, if your external processing function is processing individual frames of a multi-frame buffer with vector instructions, the memory alignment of each frame may need to be considered, and defining the input buffer's frame size to be a multiple of 32 bytes may be necessary. For the output buffer, the original buffer should not be modified; instead the new values to be written should be returned as the output of the external function.

The format for the Matlab processing function should be an m file function with the first statement formatted as follows:

```
function Output = myProcessingFunc(Input)
```

Only one input and output parameter is allowed, and the format of the Output array should match the format of the target array. The output format can be determined at run time by using the Resource buffer definitions in the Matlab workspace. For example, a function that reads the receive RF data from the designated buffer, and returns image data, could have the following definition:

```
function ImageData = myProcessFunc(ReceiveData)
```

In the above function definition, both ReceiveData and ImageData are local variables to the function. If the ImageData returned is to replace a frame in an ImageBuffer, it should have dimensions (Resource.ImageBuffer(n).rowsPerFrame, Resource.ImageBuffer(n).colsPerFrame).

Another more complete example is the following external processing routine that plots the RF data for a specific channel after an acquired frame. The processing structure is defined as follows:

```
Process(2).classname = 'External';
Process(2).method = 'myFunction';
Process(2).Parameters = {'srcbuffer','receive',... % buffer to process.
                         'srcbufnum',1,...
                         'dstbuffer','none'};      % no output buffer
```

In the Event list, the external processing function can be referenced in any event without an existing processing reference. In the following case, the processing reference is included in a 'NOP' event.

```
Event(n).info = 'noop';  % noop between frames for frame rate control
Event(n).tx = 0;         % no transmit
Event(n).rcv = 0;        % no rcv
Event(n).recon = 0;      % no reconstruction
Event(n).process = 2;    % external processing function
Event(n).seqControl = 1; % reference for 'noop' command
```

The external processing function for plotting the RF data is defined in an m file as follows, and should be saved in a location in the Matlab path.

```
function myFunction(RData)
```

```
    persistent myHandle
    channel = 10;  % Channel no. to plot
    if isempty(myHandle)||~ishandle(myHandle)
        figure;
    myHandle = axes('XLim',[0,1500],'YLim',[-2048 2048], …
                    'NextPlot','replacechildren');
    end
    plot(myHandle,RData(:,channel));
    return
```

Defining multiple external functions in the default directory can be problematic - the files need to have unique names from those used in other scripts and when a script is moved to a different directory, one must remember to also move the associated external function files. To keep the external processing functions from getting separated from the Setup script, it is possible in recent software releases to define the external processing function m file in the Setup script as a cell array of program lines. The cell array must have the name EF(n).Function and the first member should be the prototype of the function. The program lines are defined as quoted strings. For example, the "myFunction.m" file listed above can be defined in the Setup script as follows:

```
EF(1).Function = {'myfunction()', …
    'persistent myHandle', …
    'channel = 10;  % Channel no. to plot', …
    'if isempty(myHandle)||~ishandle(myHandle)', …
    '    figure;', …
    'myHandle = axes(''XLim'',[0,1500],''YLim'',[-2048 2048],...' …
    '                ''NextPlot'',''replacechildren'');', …
    'end', …
    'plot(myHandle,RData(:,channel));', …
    'return', …
    };
```

Note the use of double quotes to include a quoted value in a character string. When an external function cell array definition is included in the Setup script, the loader function, VSX, will interpret the cell array into a file in the 'tmp' directory of Matlab, and add the 'tmp' directory to the path.

For long external functions, the definition of the function as quoted strings in a cell array is tedious and difficult to read. A helper function is provided that will translate text between two delimiters into cell array strings, called 'text2cell' whose calling inputs are the filename to read from (by default, the script that calls the function) and the delimiter string. This allows writing normal Matlab code at the end of the Setup file that can be automatically interpreted into `EF.Function` cell array. To prevent this code from being executed when the SetUp script is run, place a 'return' after all structures in the SetUp script have been defined, and then after the 'return', place the lines of code for the external function between two instances of a specified delimiter. Use only the function prototype on the first line, since including the key word 'function' will generate a Matlab error, as function definitions are not allowed in scripts. For example, we can specify the `EF(1).Function` above as follows:

```
return % Place this return at end of script to prevent executing code below
%EF#1
myfunction()
```

```
persistent myHandle
channel = 10;  % Channel no. to plot
if isempty(myHandle)||~ishandle(myHandle)
   figure;
myHandle = axes('XLim',[0,1500],'YLim',[-2048 2048],...' …
                'NextPlot','replacechildren');
end
plot(myHandle,RData(:,channel));
return
%EF#1
```

Then all that is needed to insert the function lines into `EF(1).Function` is the following:

```
> EF(1).Function = text2cell('%EF#1');
```

Be sure to use a unique identifier for defining multiple external functions, and be sure not to use this identifier string anywhere in the main body of your SetUp script, as it must uniquely bracket your function lines.

After running VSX, the file created in the tempdir directory can be viewed in the Matlab editor by typing (VSX adds the tempdir to the path):

```
>edit myFunction.m
```

For debugging, remember that each time VSX is executed, the function in the tmp directory is overwritten by the cell array definition, so changes to the code should be made in the cell array definition or the delimiter bracketed text, not in the function in the tmp directory. To debug the external function in Matlab, insert the line,

```
'keyboard', …
```

within the cell array definition, or just

```
keyboard
```

for the delimiter bracketed definition, after the function prototype line or at the point where you want to begin debugging. This will cause Matlab to break at this line and return to the command prompt. At this point, one can bring up the external function in the Matlab editor, using the edit command above, and single step through the code and observe the execution. If necessary, exit debug mode with the ‘dbquit’ command.

3.6 Sequence Control Objects

Sequence Control objects are used to associate special function actions with a sequence event. In addition, some commands relate specifically to flow control for the hardware and software sequencers. The `SeqControl` structure has the following form:

```
SeqControl =
    command       [string]   Command specifying seq. event action (see text).
    condition     [string]   Condition for enabling command.
    argument      [double]   Command argument
```

Multiple `SeqControl` commands can be specified in a single event by specifying a list of their index values in the `Event.seqControl` attribute. (For example, `Event.seqControl = [1,2,5,8]`). When a `SeqControl` index or list of indices is given in an `Event` object, the control command(s) listed will be generally be executed following any TX/Receive acquisition period in the order specified. (A TX/Receive acquisition period is indicated by the presence of a non-zero TX or Receive structure number in the `Event` structure.) However, some `SeqControl` commands need to execute at the start of an event and before any TX/Receive acquisition period. Regardless of the order specified in a multiple command SeqControl list, the following commands will execute at the start of an event.

1) 'triggerOut' - the trigger output pulse will be generated at the start of an `Event` and immediately before any TX/Receive interval. Refer to the 'triggerOut' definition text below for details.
2) 'pause' (for trigger in) or 'triggerIn' - with either of these, the HW sequencer will pause and wait for the trigger input signal immediately before the point at which it is ready to start a TX/Receive interval in the same event, and thus this interval will start immediately after the trigger signal.
3) 'setRcvProfile' - when placed in an `Event` with an active TX/Receive interval, the transition to the new RcvProfile state will be completed prior to the start of the active interval for that Event.
4) 'setTPCProfile' - when placed in an `Event` that includes TX/Receive activity, a TPC profile transition will be scheduled to take place after the end of transmit activity. The default 'next' condition is required in this case. If the 'setTPCProfile' is specified in a non-acquisition event, the 'immediate' condition can be used to start a transition immediately.

`SeqControl` commands that are currently recognized are listed below. If the command is recognized by both the hardware and software sequencers, it is listed in black. Commands that are recognized only by the hardware sequencer are colored red, and commands recognized by only the software sequencer in blue.

`'call'` Push current event pointer on stack and jump to event number specified in argument attribute.

`'cBranch'` Conditional branch to new event number specified in argument attribute. The condition field should be set to 'bFlag':

— bFlag is set using a Matlab function. If bFlag is 0, no branch occurs; if nz, branch. bFlag is automatically cleared after the branch is taken. (see text)

'DMA' Initiate a DMA, as described in the DMAControl object whose number is given in the argument attribute.

'jump' Unconditional branch to event number specified in argument attribute. Optional condition: exitAfterJump (see text)

'loopCnt' Set loop counter N (1-8) to value specified in argument attribute (maximum value 65536). This value will get decremented after testing (see following command). The counter to be set is specified by the text 'counterN' in the condition field.

'loopTst' Test loop counter N: if 0, continue; if nz, decrement loop count and jump to instruction specified in argument attribute. The counter to be tested is specified by the text 'counterN' in the condition field.

'markTransferProcessed' Mark a transfer as processed. The transfer to mark is referenced by providing the SeqControl number of a previous 'transferToHost' command in the argument field. This command is typically used to release the hardware sequencer from a 'waitForProcessing' condition associated with a DMA.

'multiSysSync' Used only with multiple Vantage systems that utilize the external clock synchronization module. Condition options are 'normal' (default), 'BNC_Rising' or 'BNC_Falling'. Argument specifies SW error timeout in msec (default 10000, or 10sec). (See the app notes for multi-system synchronization for usage information).

'noop' Execute NOOP ("NO OPeration) time delay of duration specified in argument attribute (value*200nsec; max. value is 2^25 - 1 for 6.7 sec). Whether the software sequencer implements the 'noop' in simulate mode 0 (running with hardware) can be set by specifying a condition attribute. 'Hw&Sim' (default if attribute not specified) means the software only executes a 'noop' in simulate mode 1 and 2, and ignores a 'noop' in simulate mode 0, that is, when running in hardware. 'Hw&Sw' specifies that the software sequencer also executes the 'noop' in simulate mode 0.

'pause' Pause for condition (VDAS) — Conditions currently supported: 'extTrigger' — Wait for external trigger. Trigger options are specified with an argument:

```
1 - Trigger input 1 enable with falling edge.
2 - Trigger input 2 enable with falling edge.
3 - Both triggers enabled (falling edges must
    occur simultaneously).
17 - Trigger input 1 enable with rising edge.
18 - Trigger input 2 enable with rising edge.
19 - Both triggers enabled (rising edges must
     occur simultaneously).
```

Adding 256 to the above argument values specifies a wait for external trigger with a 2.8usec delay. This is used to lock out DMA memory accesses to insure consistent acquisition times.

'returnToMatlab' When this command is encountered, the processing function, runAcq, suspends execution and returns to the Matlab environment. The next time runAcq is called (with no

startEvent specified), execution resumes at the next Event in the sequence.

'rtn' Pop event number pointer from stack and start executing instructions at next event from pointer.

'setTPCProfile' Specifies a transition to a TPC high voltage profile number, which is given in the argument field. Valid profile numbers are 1 to 4, with profile 5 available with the high power option of the TPC. Conditions supported:

'immediate' - start immediately the transition to the new profile.

'next' - (default) start the transition to the new profile at end of the TPC.xmitDuration period if set, or the end of the current acquisition period.

'setRcvProfile' Specifies a transition to a RcvProfile, whose index is provide in the argument field. The new profile will stay in effect until the next setRcvProfile command is given.

'stop' Stop executing instructions & clear instruction pointer.

'sync' Synchronize hardware and software sequencers. A timeout for how long the software sequencer will wait for the hardware sequencer can be set in the argument attribute (up to 2.14e09 usec). (see text)

'timeToNextAcq' Specifies the time (in microseconds) to the next acquisition event. The time is specified in the argument attribute, which can range from 10 — 4190000 usec. The condition field can be used to indicate whether that a missed timeToNextAcq period should not be reported as a warning message ('ignore'). The software sequencer only recognizes a 'timeToNextAcq' in simulate mode 1 and 2, and ignores it when running in hardware (simulate mode 0). (see text)

'timeToNextEB' Specifies the time (in microseconds) to the next extended burst transmit event. The time is specified in the argument attribute, which can range from 10 — 4190000 usec. The condition field can be used to indicate whether that a missed timeToNextAcq period should not be reported as a warning message ('ignore'). (See Section 3.2.4.2 - Using the High Power TPC Profile.)

'transferToHost' This command is used to have VSX create a 'DMA' command and automatically insert it into the sequence. A DMAControl object is also created to specify the DMA transfer. An optional condition attribute can be specified with this command named 'waitForProcessing' which then requires an argument value. Under this condition, the DMA transfer will be executed, but if the data transferred by the SeqControl structure specified in the argument has not been marked processed or reconstructed (by a Recon object that references the RcvBuffer frame transferred to), the hardware sequencer will pause before the next DMA transfer. Note: the SeqControl structure referenced in the argument field must be another 'transferToHost' command. The timeout in msecs for the hardware sequencer wait is set by Resource.VDAS.dmaTimeout (default = 1000msec). (See Text.)

```
'triggerIn' Specifies the conditions for the TRIG IN 1&2 BNC connectors.
            (See Text.)

'triggerOut' Generates external 1 microsecond active low output on the
             TRIG OUT BNC connector. A delay can be set in the argument
             field. (See Text.)

'waitForTransferComplete'  Used to pause the software sequencer until a
             transfer specified in the argument field is complete.  The
             argument field should reference the SeqControl number of a
             previous 'transferToHost' command. This command pauses the
             software sequencer before any processing action and waits for
             a 'transferred-But-Not-Processed' flag in the DMAControl
             object to be set at the end of the referenced transfer.  The
             flag will stay set until a SeqControl command of
             'markTransferProcessed' is given.  There is a timeout on the
             length of the wait, which is set by Resource.VDAS.dmaTimeout,
             which defaults to 1000msec.
```

The '`jump`' command has some special characteristics. When the sequence is running, the software is processing the event list independently of the hardware sequencer. In software, the processing of a '`jump`' command back to the first event in the sequence implements a return to the Matlab environment, to allow checking for user interface events. If there are no user interface actions to process, Matlab again calls the runAcq processing routine and processing continues with the first event. If a '`jump`' command goes to some other event rather than the first event, no return to Matlab is implemented (this allows for loops in the sequence without the overhead of the return to Matlab). In some cases, however, one might desire a return to Matlab for a jump that does not go back to the first event. In these cases, specifying a condition of '`exitAfterJump`' with the '`jump`' command will force the return to Matlab. The next call to runAcq will resume event processing at the next event after the '`jump`' event. If one just wants to return to Matlab at some point in the event sequence, the 'returnToMatlab' command can be used. There should be at least one return to Matlab in any sequence that repeats, since without a return, the sequence cannot be exited.

The 'cBranch' (conditional branch) command has a Matlab function to set the 'bFlag' which causes the branch to be taken - '`setConditionalBranchFlag`'. This function is called from the Matlab workspace and can be used to cause the hardware sequencer to branch to a different sequence event based on a user input from a GUI control. To also cause the software sequencer to branch, set the `Control.Command` string to 'setBFlag' in the Matlab workspace. The next return to Matlab will then pick up this command and set the BFlag in the software sequencer. It should be noted that the hardware and software sequencers will not necessarily branch at the same time, which can lead to problems with synchronous sequences. If the hardware and software sequencers need to start from the branch point at the same time, the 'sync' command can be used in the first event of the branch with no other actions.

The 'triggerIn' command specifies the conditions expected at the two trigger in BNCs on the rear panel. These conditions are provided in the `SeqControl.condition` attribute and are:

'Trigger_1_Falling'
'Trigger_2_Falling'

'Trigger_1_2_Falling'
'Trigger_1_Rising'
'Trigger_2_Rising'
'Trigger_1_2_Rising'

The system will wait for the specified condition to occur with a timeout provided in the SeqControl.argument attribute, which can range from 0 to 255, where each count represents a time increment of 250 msec. The default value is 0, which means that the system will wait forever for the trigger condition to occur.

The 'triggerOut' command, when programmed in a non-acquisition event, is sent immediately when the event is encountered in a sequence, subject to conditions specified below. When programmed in the same event as an acquisition (tx or tx/rcv), the trigger will be sent just before the start of transmit delay time period, which includes waiting for a previously programmed 'timeToNextAcq' to expire. There are several conditions that may be applied to the trigger out: 1) 'syncADC_CLK' synchronizes the trigger out to an edge of the specified A/D clock. 2) 'syncSYNC_CLK' synchronizes to the SYNC clock, which is 1/4 of the A/D clock. This option must be used if the system is triggering itself and another Vantage unit. 3) 'syncNone' (default) means the trigger out is generated as soon as possible after the trigger's scheduled time. If it is desired to trigger an external event some time in advance of the TX/Receive period, place the 'triggerOut' command in a separate event preceding the acquisition event, and follow it with a 'noop' command with the time period desired.

The trigger output pulse can also be delayed from the beginning of the associated transmit/receive event by an optional time delay specified in the argument field. The argument must an integer value from 0 to 1,048,575 (the equivalent of a 20 bit unsigned integer), representing counts of the 250 MHz clock period (4 ns) for a maximum delay of about 4 milliseconds. This programmable delay permits precise timing control over the trigger out pulse with respect to the start of acquisition, including delaying the trigger pulse enough to permit setting up a following wait for trigger-input event that will detect that trigger pulse and continue the sequence. If an argument is not specified, a default value of zero delay will be used.

The `'DMA'` command is meant to be set only by the runtime software, and will be substituted for any user specified `'transferToHost'` commands. The '`DMA`' command ('`DMA`' stands for Direct Memory Access) specifies the transfer of data in an address range of local memory on the acquisition modules to the memory of a `RcvBuffer` in the host computer. A DMAControl structure (also created by the runtime software) is referenced in the SeqControl.argument attribute that provides details of the transfer. The complexities of DMA programming have been abstracted from the user, who only needs to manage the times within their sequence when acquired data should be transferred to a `RcvBuffer`. This is accomplished using the '`transferToHost`' command.

'transferToHost' programming

At the events in the sequence list where one wants to transfer acquired data to the host `RcvBuffer` (typically at the end of a frame), the user specifies a `'transferToHost'` `SeqControl` command. Before loading a sequence, the runtime software goes through the `Event` list in order, and when it finds an event with a `SeqControl` command of `'transferToHost'`, it generates a `DMAControl` object for the acquisition data of the

previous acquired frame in the `Event` list. When the hardware sequencer gets to this event, it generates an interrupt that initiates the DMA transfer to the host `RcvBuffer`. Note that there must be a unique `SeqControl` structure for each `'transferToHost'` command in the sequence; in other words, you can't 're-use' the same `SeqControl(n).command='transferToHost'` at multiple points in your sequence, since each `'transferToHost'` is associated with a unique DMA action. Starting with the `Event` that contains the '`transferToHost`' command, the software searches backwards through previous `Events`, looking at the `Receive` structures reference by acquisition `Events`. The `RcvBuffer` number and frame number of the first found `Receive` are captured and compared with previous `Event Receives`. All `Receives` with the same `Receive.bufnum` and `Receive.framenum` number are then used to compose the `DMAControl` structure for the transfer. It is therefore important in a sequence not to mix acquisitions from different `RcvBuffers` or `RcvBuffer` frames prior to a '`transferToHost`' action. If one does, the data transferred will only be from the last `Events` that have the same `Receive.bufnum` and `Receive.framenum` attributes. When `'transferToHost'` DMAs are programmed into the hardware sequencer, two conditional stop commands are set before the each DMA of the sequence. The first is a conditional stop to 'wait for processing', and its function will be described in Section 3.6.1 below. The second is a conditional stop for a 'DMA in progress'. This stop is set when a `'transferToHost'` DMA is started, and cleared when the DMA completes. This prevents a new DMA from being initiated if the previous DMA has not yet completed. If DMA can't keep up with acquisition, this stop may lead to a missed `'timeToNextAcq'` message (described below).

The `'transferToHost'` command is typically used without a `'waitForProcessing'` condition. In this case, the hardware sequencer will perform the DMAs without setting a stop condition other than the 'DMA in progress' stop; if all `'transferToHost'` commands are set this way, the hardware sequencer will run asynchronously at its own rate, independent of the processing. In this mode of operation, reconstruction or external function processing can be programmed to process the most recently transferred frame (using `Recon.rcvBufFrame = -1`), which will process frames as fast as the processing will allow, up to the acquisition frame rate. Setup scripts using this method of programming are referred to as 'asynchronous' (described further in section 3.6.1 below). Asynchronous programming can be used for some interesting acquisition sequences, where acquisition can run at a very high frame rate, capturing frames into a multi-frame `RcvBuffer`, and during live acquisition, software is processing frames as fast as it can, skipping acquired frames as needed to generate a real-time display of frames. In RcvDataLoop mode, all frames acquired will be processed, so that the `RcvBuffer` frames play back at the processing frame rate, which might look like slow motion playback. See the SetUpL11_5vFlash script for an example of how to use this method of acquisition.

The `'sync'` command can be used to synchronize the hardware and software sequencers at a specific event in the sequence. This command sets a conditional pause for both sequencers, and whichever sequencer reaches the sync point first will pause waiting for the other sequencer to arrive. When the other sequencer arrives, the conditional pause is released and both sequencers continue to the next event in the sequence. The user can set a timeout for how long the software sequencer will wait for the hardware sequencer to arrive, by specifying the time in microseconds in the

argument field of the SeqControl with the 'sync' command. The default timeout is 500000 usec (0.5 seconds). The hardware sequencer doesn't have a timeout, and will wait for the software sequencer to catch up indefinitely. The 'sync' command is treated somewhat differently than other SeqControl commands in that it's order of execution in an event with multiple SeqControl commands is enforced independently from the order given in the list of SeqControl commands. The 'sync' conditional pause is always executed after any acquisition (Tx and Rcv), reconstruction or processing actions and before any branching commands. If a trigger in or out is included with a 'sync' command in an event that has no acquisition actions, the trigger action occurs after the 'sync' command and before any branching commands. If the event performs acquisition actions, the trigger in or out is placed before the acquisition actions and thus before the sync command as well.

The 'timeToNextAcq' command is used to set a precise time between acquisition events. It can also be used to set the duration between transmit only Events (Event.rcv = 0). The time duration is set in the argument field in microseconds (10 – 4190000). When the transmit/receive period starts for the current event, a counter is set for the duration of the timeToNextAcq, and this counter then runs in the background until the next Event with a TX reference occurs. Before the transmit/receive period is started for this next Event, the counter is checked, and if still running, the transmit/ receive period is delayed until the counter completes (note that this delay applies to the start of the transmit delay countdown, since the transmit delay counters are started at the start of the transmit/receive period). This provides a precise time difference between the successive acquisition events. If the new Event also has a timeToNextAcq command, the counter is set to run again with the value in the argument field. In the case that the timer has already completed when the next Event is initiated, a warning message is printed to notify the user that their requested time duration is not being met. This can occur for several reasons: 1) In an asynchronous script, the typical reason for not meeting timing is excessive DMA time. If a previous DMA is in progress when a new DMA is ready to be launched, the sequence must pause until the previous DMA completes. This can then lead to missed timeToNextAcq durations. The solutions here are a) reduce the amount of data transferred, possibly by using a lower signal bandwidth (Receive.sampleMode), b) group your acquisitions and transfer more data with each DMA, thus minimizing the overhead associated with launching new DMAs, or c) increase the timeToNextAcq duration. 2) In a synchronous script, the typical cause for missing timeToNextAcq durations is processing delays. If the processing takes longer than the time durations programmed into the acquisition sequencer, the timeToNextAcq durations will be missed. When the warning is output, it also displays a duration in microseconds that the timing missed by. This is the time after the counter completed to when the sequencer finally got to the next acquisition event. This missed time duration can be used as an aid in deciding how much to increase the 'timeToNextAcq' duration. (Note: In a synchronous script that is missing timing repeatedly due to processing delays, the missed time duration may be longer than actual, due to the added time to print the warning message.)

3.6.1 Methods for Programming Acquisition and Processing Sequences

As mentioned previously, the hardware and software sequencers run independently at their own rates, the hardware sequencer processing transmit/receive acquisitions and the software sequencer processing acquired data. In most cases, this is an advantage, allowing high frame rate acquisitions, while processing the most recently acquired frame at a slower frame rate. Switching to RF loop playback, one can then visualize all frames, which are played back at the processing frame rate. There are some times, however, when it is needed to synchronize actions by the two sequencers. For example, one might want to acquire frames with different acquisition methods, with unique processing for each frame. In this case, we would want the hardware sequencer to be synchronized with processing, so that we know that we are processing the correct set of acquisitions.

3.6.1.1 Asynchronous Acquisition and Processing

This method is the default method used by most example scripts. Please refer to Fig. 3.6.1.1 to help interpret the sequence of events and actions. Transmit/receive acquisition events are created for acquiring a full “frame” of RF data, and these events are then replicated for some number of frames. After each frame, a sequence control `transferToHost` is performed which transfers the frame data to the host computer in the RcvBuffer. The acquisition events execute completely independently from any software processing and can be precisely timed with `timeToNextAcq` commands. At the end of each `transferToHost`, an identifier for the frame just transferred is written to a hardware register which we can name ‘lastFrame’. For reconstruction processing of a frame with Recon, the Recon.rcvBufFrame attribute is set to -1 (for external processing functions with srcbuffernum = ‘receive’, set srcframenum = -1).

With these settings, the frame number to be used for processing is determined by a function which accesses the ‘lastFrame’ register. If the function accesses the ‘lastFrame’ register and finds either a ‘0’ (no last frame) or the same frame number as the previous access, it will poll the ‘lastFrame’ register for 1000 milliseconds (this time can be set differently by setting `Recon.newFrameTimeout` to the polling time in milliseconds) to try and get a new frame to process. If the function finds a new frame number, it returns immediately and the processing can start on the new frame. There is no need to wait on a transfer to complete, since only completed frame transfers are reported to the ‘lastFrame’ register. To insure that hardware acquisition doesn’t overwrite RF data in a frame being processed, it is necessary to provide enough frames in the RcvBuffer so that the acquisition time to wrap around the buffer and write to the frame just transferred is longer than the time to process a frame.

This method of acquisition and processing has several advantages. 1) Precise timing can be achieved for transmit/receive events - especially important for Doppler acquisitions. 2) Frame acquisitions can proceed at a much higher frame rate than processing of frames - often required for high frame rate acquisitions such as shear wave elastography. 3) Processing of frames at a lower rate than acquisition allows real-time visualization of transducer position for high frame rate applications. 4) Processing of frames occurs independently and concurrently with acquisition. A possible disadvantage is that the displayed frames in real-time may not occur at an even rate. At relatively modest frame rates (greater than ~15 fps), this effect is not typically noticeable. When switching to RcvDataLoop playback, all frames are processed

sequentially at the software processing rate, which then may appear as slow motion playback.

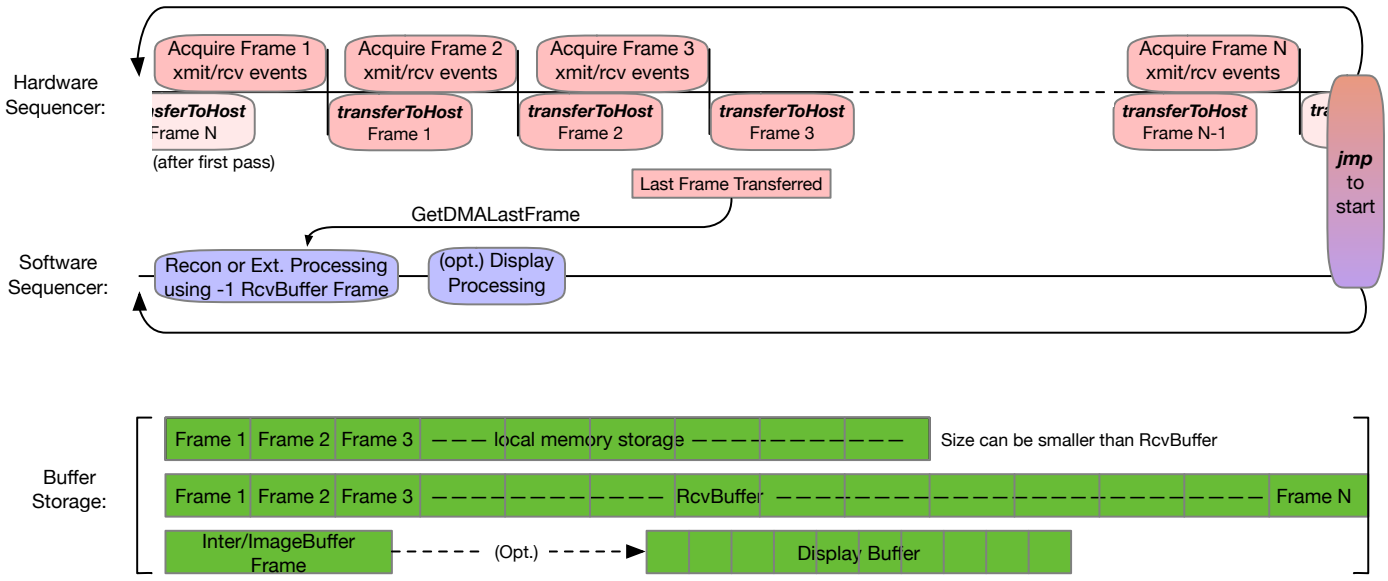


Fig. 3.6.1.1 Asynchronous Acquisition with Concurrent Processing.

3.6.1.2 Serial Acquisition and Processing

In this method, ultrasound acquisitions, data transfers and processing all occur in sequence. Please refer to Fig. 3.6.1.2 to help interpret the sequence of events and actions. Transmit/receive operations are performed by the hardware sequencer to acquire a 'frame' of ultrasound data. These data are stored in local memory on the hardware acquisition modules in real-time at the sample rate set by the analog to digital converters for each channel. At the completion of a frame, the `transferToHost` command is initiated and the data moved from local memory to a frame defined in the host RF memory buffer using Direct Memory Access (DMA). At completion of the transfer, an internal TransferComplete flag is set.

The software sequencer waits for the TransferComplete flag to become set using the `waitForTransferComplete` command, and then initiates the processing routine. The internal Recon processing automatically performs this wait, but if using an external processing routine, the `waitForTransferComplete` sequence control action is needed. At the completion of processing, a `markTransferProcessed` action should be performed to clear the TransferComplete flag. Again, the internal Recon processing automatically performs this action, but an external processing routine must explicitly use this command after processing completes. Finally, a `sync` control command is placed after all processing has completed. This aligns the hardware and software sequencers to start acquiring and processing a new frame of ultrasound data at the same time.

This serial acquisition method is relatively straight forward to program, and has a few advantages. 1) All frames acquired are processed and visualized in real-time. 2) The sequence uses a single RcvBuffer frame, which can be quite large using all the available memory (be careful to respect the 2 GByte `transferToHost` limit). 3) Similarly, only a single InterBuffer or ImageBuffer frame is needed. 4) Multiple unique acquisition and processing actions can be chained using this method to mix different scanning and processing methods. The main disadvantage is that the serial acquisition and processing may result in low frame rates, since processing has to wait for acquisition and transfer, and acquisition has to wait for processing.

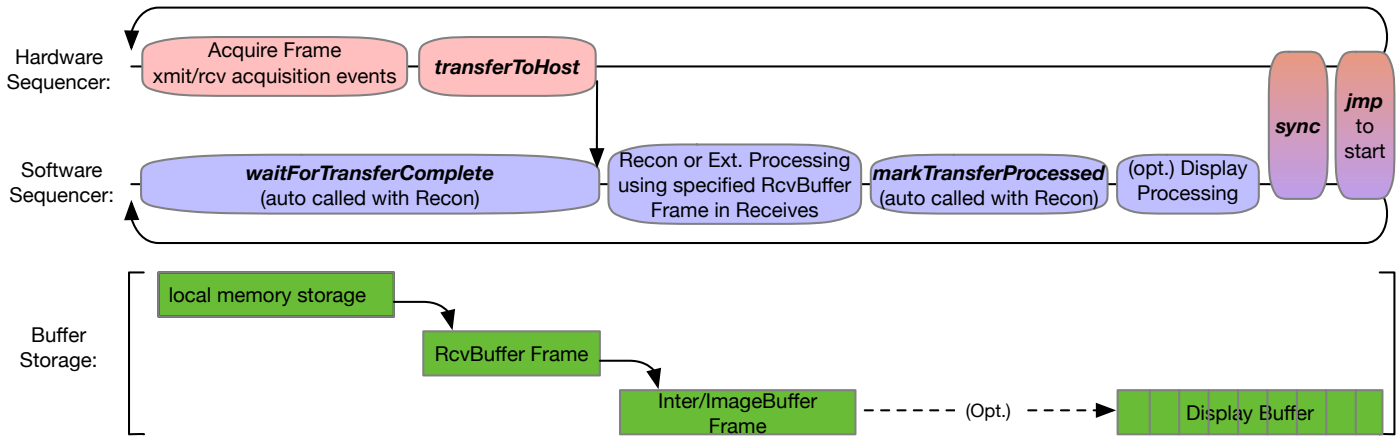


Fig. 3.6.1.2 Serial Acquisition and Processing

3.6.1.3 Synchronous Acquisition and Concurrent Processing

To improve the frame rate processing of the Serial Acquisition and Processing method, and to allow acquisition of multiple frames of RF data, a new method of synchronous acquisition and processing is now available with the 4.3 software release. This method is invoked by setting the attribute `Resource.Parameters.waitForProcessing = 1`. Referring to Fig. 3.6.1.3, we can see that this sequence looks very much like the Asynchronous Acquisition and Processing sequence defined earlier. In fact, the only difference is the setting of the `Resource.Parameters.waitForProcessing` attribute. When this attribute is set, sequence stop points are set in the hardware sequencer that prevent the hardware sequencer from getting more than one frame ahead of the software processing. At the end of a `transferToHost` action, the stop points are activated and the hardware sequencer must stop before the next `transferToHost` action. The stop is released when the software processing obtains a new frame to process from the `lastFrame` hardware register. The processing of the previous transferred frame is then overlapped with the acquisition of the next frame.

This method will then process and display frames in real-time at the software processing rate, which will depend on the processing capability of the host computer.

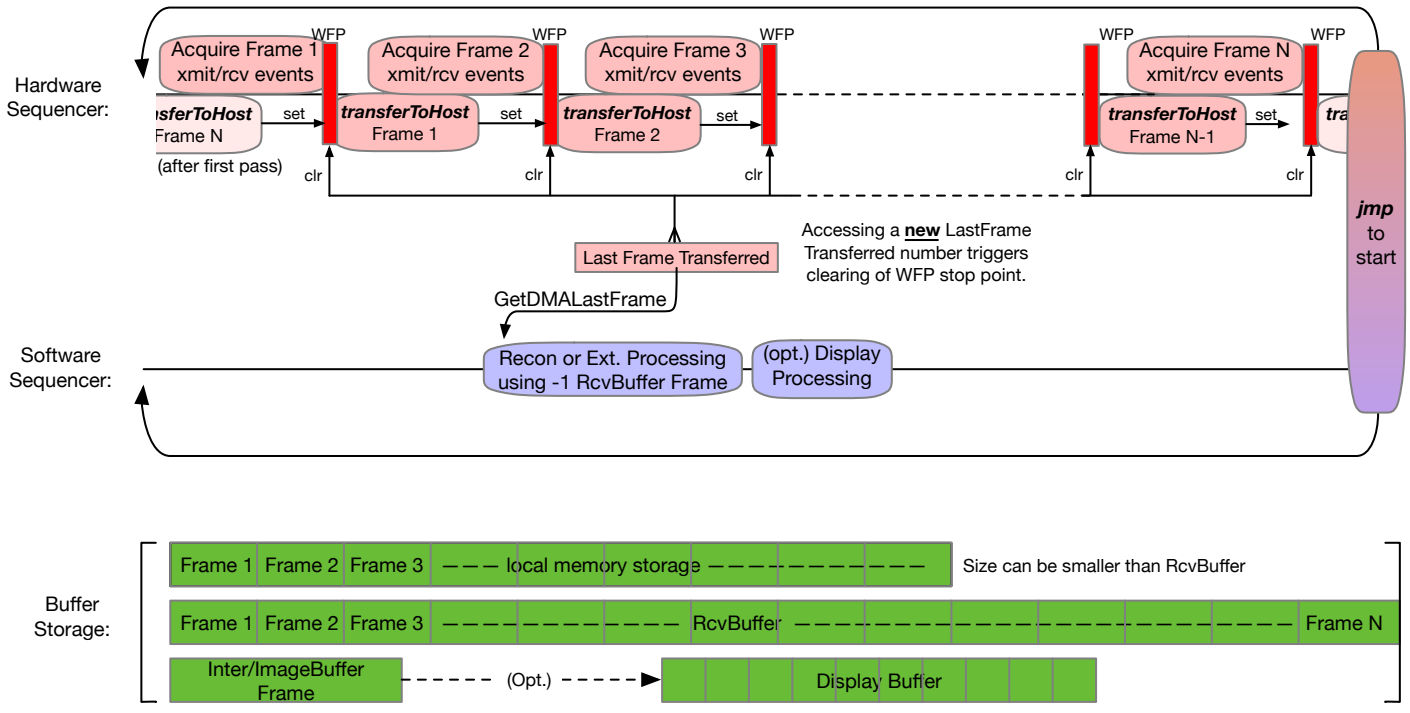


Fig. 3.6.1.3 Synchronous Acquisition with Concurrent Processing using Resource.Parameters.waitForProcessing = 1.

3.7 Event Objects (Continued)

While introduced at the start of this section, not all features of the Event object were covered. The structure is given again below:

```
Event =
      info:        string     'optional descriptive text for this event'
      tx:          double     # of TX structure array to use for transmit
      rcv:         double     # of Receive structure array to use for rcv.
      recon:       [up to 16 doubles] # of Recon structure array for recons
      process:     double     # of Process structure array to use.
      seqControl:  [up to 3 doubles] # of SeqControl structure array to use
```

In the above definition, note that the `recon` attribute can be either single valued, or an array of indices. The support for multiple reconstructions in the same event is needed for asynchronous sequences, where the `Recon` applies to the most recently acquired frame of `RcvBuffer` data. When multiple recons are encountered in the same event that all reference the most recently acquired frame (`Recon.rcvBufFrame = -1`), the determination of the most recent frame is performed only for the first Recon, and re-used for the additional Recons. This insures that all the Recons are using the same frame, which is essential for mixed mode acquisitions such as 2D and Doppler flow imaging.

The support for multiple `SeqControl` indices in the same event allows multiple control actions that need to be associated with a single event. For example, one might want to specify both a '`timeToNextAcq`' command and a '`transferToHost`' command for a particular event. When multiple `SeqControl` indices are given in the same event, they are typically executed in the order given, although often the order is not relevant to the actions being taken.

`Event` objects should be the last sequence objects defined in a script. This is so the indices of the objects referenced can be checked for existence when the objects are loaded into the system.

In the example scripts, it will be noticed that the `Event` indices are defined with a variable, n, which is incremented after each `Event`. This allows easily adding or deleting an `Event` in a sequence, without having to renumber following `Events`.

4. Verasonics Script Execution (VSX)

Once a Sequence Object has been created and bundled into a .mat file format, it can be loaded and executed using the 'VSX' Matlab program. This program performs a number of functions, including the following:

1) The Sequence Object is loaded into the Matlab environment, and the various structures are parsed, checking for missing required parameters and programming some default attributes. (Some attributes are not added until later in the initialization process, after calling the mex function 'runAcq'.)

2) The presence of the VDAS hardware is detected, and if found, the hardware is 'opened' and 'owned' by the VSX application. If no hardware is detected, the script will be executed in simulate mode.

3) For large chassis Verasonics systems (2 or 4 boards), the presence of a transducer connected to the scanhead connector specified by the script (default is 1 or left connector) is detected and its ID is read. (Most Verasonics and HDI1000 - 5000 probes contain a personality eprom in the probe handle that identifies the probe.) If a transducer is not plugged in, or the probe ID doesn't match the name of the probe specified in the user script, VSX will produce an error and exit. The user script can use a `'custom' Trans.name` entry for transducers that have no personality EPROM, in which case VSX will skip the probe ID check.

4) A graphical user interface window is opened, with some basic control functions. Any additional GUI controls defined in the setup script .mat file are added to the interface window.

5) One or more display windows are opened, according to the `Resource.DisplayWindow` definition(s) defined by the setup script.

6) An execution loop is entered, which will continue until the GUI window is closed, or the list of sequence events is completed. (Most sequences have a jump back to the start at the end of the sequence, so the sequence repeats indefinitely.) In this execution loop, a mex (Matlab external) function named 'runAcq' is called, which performs all the processing for the sequence events. 'runAcq' starts at a specified event in the sequence (`Resource.Parameters.startEvent`), and continues to execute events until one of the following conditions is met:

- The end of the sequence is reached, and there is no jump back to the start. In this case, the 'freeze' state is automatically entered in VSX. Un-freezing the system by pressing the 'freeze' GUI toggle button will run the sequence again.
- A 'jump' SeqControl command back to the start of the sequence is taken, in which case 'runAcq' returns control to the Matlab environment after taking the jump. This return to the Matlab environment is needed to process any GUI controls that may have been activated during event processing. If there is no action to be taken in the Matlab environment, 'runAcq' is called again and processing proceeds from the start of the sequence.
- A 'jump' SeqControl command occurs in the body of the event list that has the condition 'ExitAfterJump' set. In this case, runAcq will return to Matlab, and the event number that the jump was directed to is saved, so that the sequence will start

at this event the next time runAcq is executed. This modification of the jump command allows returning to Matlab while executing sub-loops in a sequence.

- A ‘returnToMatlab’ SeqControl command is encountered in the sequence. This command is used when the sequence is of a long duration in time, and waiting until the end to return to Matlab would result in too slow a response to GUI controls. The next time runAcq is called, execution will continue with the next event.
- A ‘stop’ SeqControl command is executed, in which case, ‘runAcq’ immediately returns to Matlab and the freeze state is entered.

After initialization, the typical run-time behavior of VSX and runAcq for a sequence that jumps back to its start event is for VSX to call runAcq repeatedly from its ‘while’ loop, with each call processing all events in the sequence. The runAcq function handles all event and data processing from the start event to the last event, where the jump back is found, which triggers a return to Matlab. While in Matlab, the only action is to check for any GUI events that require attention, and then to call runAcq again, with appropriate input arguments.

4.1 Defining GUI Controls

To change Sequence Object parameters during run-time, the correct method is to set up a GUI object that when activated executes a callback routine. The GUI object definition and the callback routine can usually be defined in the Setup script that builds the Sequence Object .mat bundle. The callback routine can set an action value in the Matlab environment that is read by VSX in between calls to runAcq, triggering some action in the Matlab environment. This method is used by some of the built-in GUI controls, but is not recommended for users, as it requires modifying VSX to incorporate your GUI action. Since VSX will be updated with new software releases, your GUI modification would have to be continually applied to the new code.

The recommended method for providing a run-time control without modifying VSX is to set up the GUI object and callback function in your Setup script, so that it gets incorporated into the GUI window by VSX when your script is loaded. The callback function, when activated, can modify the appropriate sequence structures specified in your script in the Matlab environment, but these modifications will not have any effect until reloaded into both the runAcq processing function and the hardware. To perform the reload, the GUI control needs to also set up a Control structure in the Matlab environment, which will be passed to runAcq on the next call from VSX, specifying the structures or attributes to be updated. The Control structure is automatically cleared after it is passed to runAcq, so that the control action only executes once.

When modifying runtime parameters, it is important to remember that VSX may have added attributes to the original script structures, in particular, VDAS attributes for programming the hardware. For structures that have added VDAS components, such as TW, TX, and Receive, the function ‘update.m’ will be called by runAcq when it detects a change to the structure to update the added VDAS components. In almost all cases, the user should not modify VDAS parameters directly.

To define a GUI control in one’s script, a structure named UI is provided. This structure has the following fields.

```
UI =
    Statement       string   Matlab statement executed at load time.
    Control         cell     Input parameters for Matlab uicontrol command.
    Callback        cell     Quoted statements for callback function.
    handle          handle   Handle for callback(s) created (set by VSX).
```

The `UI.Statement` attribute defines a string that will be executed as a Matlab statement at the time the UI control is created. This is useful for executing any commands needed to initialize the GUI or its components. The `UI.Statement` field can also be used to execute commands before the user's sequence is loaded and run. For example, one could set the initial high voltage for a specific transmit profile (which normally defaults to the minimum) with the following statements.

```
% Set TPCHighVoltage for profile one to 40V
UI(1).Statement = '[result,hv] = setTpcProfileHighVoltage(40,1);';
UI(2).Statement = 'hv1Sldr = findobj(''Tag'',''hv1Sldr'');';
UI(3).Statement = 'set(hv1Sldr,''Value'',hv);';
UI(4).Statement = 'hv1Value = findobj(''Tag'',''hv1Value'');';
UI(5).Statement = 'set(hv1Value,''String'',num2str(hv,''%.1f''));';
```

The above statements set profile one's high voltage to 40 volts, and also set the high voltage slider and text field to reflect the new value. Since the 3.3.0 software release, the simpler method of setting an initial high voltage is to set the hv attribute in a TPC profile structure.

```
TPC(i).hv = voltage;
```

The `UI.Control` attribute is a cell array of attribute/value pairs that are to be used in the Matlab uicontrol command, or with a custom Verasonics control (see next section below). For example, the attributes typically needed to define a slider control could be specified as follows:

```
UI(1).Control = {'Style','slider',...
                 'Position',[0.375,0.1,0.25,0.030],... % position on UI
                 'Max',10.0,'Min',1,'Value',5.0,...
                 'SliderStep',[0.025 0.1],...
                 'Tag','mySlider',...
                 'Callback',{@mySliderCallback}};
```

Note that the position units are defined in relative values, with respect to the full GUI window size. Fonts should also be defined in relative units, so that the text is sized to fill the defined control size. VSX will pass these attributes to the Matlab uicontrol function when the GUI is created, and set the handle to the control in the `UI.handle` attribute. The handle can be used to find the control by other functions or callbacks, or the control defined above could also be found by its tag, 'mySlider'.

Finally, the `UI.Callback` attribute provides a cell array definition of a callback function associated with the control. The first cell specifies the prototype of the callback function, whose name should match the name specified in the `UI.Control 'Callback'` attribute. The following cells specify the lines of the callback function, which VSX will create in the tmp directory of the system. (Matlab defines a temporary directory for files, accessed by the function 'tempdir'.) The callback function typically modifies a sequence object attribute, and passes it to the sequence execution program,

runAcq, using the Control structure (described in section 4.2). For example, if the slider defined above is used to control the processing gain (pgain) attribute of the second Process structure, the callback definition could be as follows:

```
UI(2).Callback = {'mySliderCallback(hObject,eventdata)',...
                  ' ',...
                  'pgn = get(hObject,''Value'');',...
                  'Control = evalin(''base'',''Control'');',...
                  'Control.Command = ''set&Run'';',...
                  'Control.Parameters = {''Process'',2,''pgain'',pgn};',...
                  'assignin(''base'',''Control'', Control);',...
                  'return'};
```

To make it easier to create and interpret long callback functions, a utility function is provided that will translate text between two delimiters into cell array strings, called 'text2cell' whose calling inputs are the filename to read from (optional; if missing defaults to calling script file) and the delimiter string. This allows writing normal Matlab code at the end of the Setup file that can be automatically interpreted into the `UI.Callback` cell array. To prevent this code from being executed when the SetUp script is run, place a 'return' after all structures in the SetUp script have been defined, and then after the 'return', place the lines of code for the callback function between two instances of a specified delimiter. Use only the function prototype on the first line of the text to be encoded, since including the key word 'function' will generate a Matlab error, as function definitions are not allowed in scripts. For example, the UI(2).Callback function above can be written as follows:

```
return % Place this return at end of script to prevent executing code below
%CB#2
mySliderCallback(hObject,eventdata)
pgn = get(hObject,'Value');
Control = evalin('base', 'Control');
Control.Command = 'set&Run';
Control.Parameters = {'Process',2,'pgain',pgn};
assignin('base', 'Control');
return
%CB#2
```

The delimiter should be placed on its own line, and start with a comment character, immediately followed by the delimiter string. To insert the above function lines into `UI(2).Callback` use the following statement:

```
> UI(2).Callback = text2cell('%CB#2');
```

The delimiter can be any string, but one should make sure that it is not used elsewhere in the script file, other than in the function call itself. (A delimiter surrounded by single quotes is ignored when searching for the start of text to encode.) For example, one might want to include the function name in the delimiter string, such as:

```
'%myCallback#2'
```

If one wants to encode text from a file other than the calling script, specify the filename (with its path, if not in the default path) as the first input parameter in text2cell, for example:

```
> UI(2).Callback = text2cell('myCodeFile.m','%CB#2');
```

4.1.1 Verasonics UI Controls

To facilitate creating user controls on the Verasonics GUI, newer software releases provide some built-in UI functions that can be positioned at certain predefined locations. The currently available VsXXX controls that can be selected are: VsSlider, VsPushButton, VsToggleButton, and VsButtonGroup. These controls are specified by a custom location and 'Style' in the UI.Control cell array, as follows:

```
UI(1).Control = {'UserXX', 'Style', 'VsXXX', … }
```

The locations currently available for user controls are shown in fig. 4.1.1 below. The label used for 'UserXX' in the statement above specifies the location of the control. For example, to create a control in the lowest location in the center column, use `'UserB1'` for the first entry in `UI.Control`. Note that some of the user control locations overlap with some of the default controls placed in the GUI by VSX. This is because the overlapped controls are conditional - the Speed of Sound control is added only if a Recon structure is present in the script, and the Zoom, Pan, Compression, Reject and Persistence controls are added only if an ImageBuffer is defined. Even in scripts where the default controls are added, a user control can be placed on top of a default control if desired.

Verasonics user controls are created at sequence load time by VSX, and are created with default tags and callbacks. The callbacks are basically code preambles that help manage the control operation and provide values for the user-provided code that follows. The user-provided code is placed in the `UI.Callback` cell array and added to the preamble code when the callback function is created. When creating the `UI.Callback` cell array, include only the code statements needed. VSX will name the function according to the UserXX location as `'UserXXCallback.m'`, and the function definition and calling parameters are automatically included in the preamble.

As described in the previous section, the user-provided code for the callback functions can be written between specified delimiters at the end of the SetUp script and placed in the UI.Callback cell array using the statement:

```
> UI(n).Callback = text2cell('%CB#n');
```

where the delimiter in this case is `'%CB#n'` and n is the number of the UI structure.

While it is possible to code callbacks and external functions as separate Matlab function files that are in the Matlab path at the time of running VSX, the coding of these functions within the SetUp file is recommended, as it keeps callback and external functions together with the SetUp script that requires them, and eliminates having to manage a proliferation of small function files that require different names in the main directory.

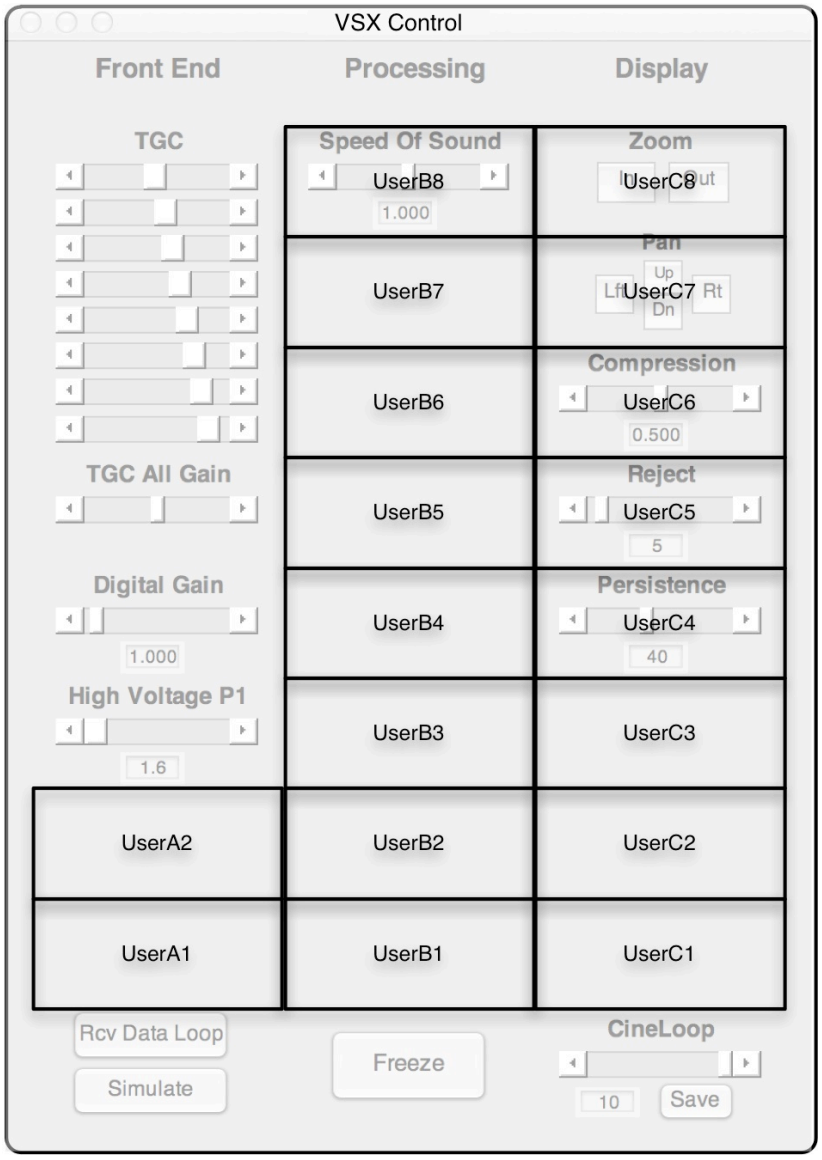


Fig. 4.1.1 Locations available for Verasonics custom controls.

VsSlider - When a UserXX location is specified with a 'Style' of 'VsSlider', VSX will create three UI controls - a text box, a slider, and an edit window linked to the slider operation. Unique attribute/value fields for the VsSlider can be provided that simplify creation. These attribute/value combinations are:

```
Label, Text string
SliderMinMaxVal, [Min,Max,initialVal]
SliderStep,[small,large]   default [0.01,0.1 ]
ValueFormat, '%3.0i'       default format for integer field
```

The handles for the text, slider and edit controls are placed in `UI.handle(1)`, `UI.handle(2)` and `UI.handle(3)`. Tags are also created for the slider ('UserXXSlider') and edit box ('UserXXEdit'). For the edit box created, the ValueFormat is copied to the UserData field of the UIControl, where it can be accessed from a callback that changes the value.

When the slider or edit box value changes, the `UI(n).Callback` function is executed, with automatic updating of the slider and or edit box values. The callback created by VSX is named UserXXCallback.m and has the preamble code below to synchronize the slider and edit values. The user callback code to be executed after the preamble is specified in `UI(n).Callback`, and can assess a variable named 'UIValue' to obtain the changed value from the slider or edit box.

```
function UserXXCallback(hObject, eventdata);

Cntrl = get(hObject, 'Style');
if (strcmp(Cntrl,'slider')
   UIValue = get(hObject,'Value');
   h = findobj('Tag', [UserXX 'Edit']);
   set(h,'String',num2str(UIValue,get(h,'UserData'));
else
   UIValue = str2num(get(hObject,'String'));
   h = findobj('Tag',[UserXX 'Slider']);
   max = get(h,'Max');
   min = get(h,'Min');
   if (UIValue > max)
      UIValue = max;
      set(hObject,'String',num2str(UIValue,get(hObject,'UserData'));
   end
   if (UIValue < min)
      UIValue = min;
      set(hObject,'String',num2str(UIValue,get(hObject,'UserData'));
   end
   set(h,'Value',UIValue);
end
< User provided statements >
```

For example, creating the UI controls for the channel selector slider described in the Sequence Programming Tutorial is simplified to the following:

```
UI(1).Control = {'UserB1','Style','VsSlider','SliderMinMaxVal',[1,nr,32]};
UI(1).Callback = {'assignin(''base'',''myPlotChnl'',UIValue)'};
```

VsPushButton - When a UserXX location is specified with a 'Style' of 'VsPushButton', the available attributes are:

Label, Text String	Label to be placed within the push button control.

The `UI(n).Callback` code is executed when the push button is released with the cursor in the frame of the button. In this case, the callback preamble is as follows:

```
function UserXXCallback(hObject, eventdata);
```

VsToggleButton - When a UserXX location is specified with a 'Style' of 'VsToggleButton', the available attributes are as follows:

Label, Text String	Label to be placed within the toggle button control.

The `UI(n).Callback` preamble for the toggle button provides the state of the toggle button in the UIState variable.

```
function UserXXCallback(hObject, eventdata);
```

VsButtonGroup - When a UserXX location is specified with a 'Style' of 'VsButtonGroup', the available attributes are:

Title, Text String	Title of the button group (placed at top of bounding box).
NumButtons, N	Number of buttons in the group (default = 2).
Label, [Text1,…TextN]	Labels for button N buttons.

For 2 buttons, (the default) the size of the button group is the same size as the location box. For more buttons, the button group grows downward and overlaps user locations below. The `UI(n).Callback` code is executed when a button is pressed. The variable UIState is set to the number of the button pressed. In this case, the callback preamble is as follows:

```
function UserXXCallback(hObject, eventdata);
S = get(eventdata.NewValue,'Tag');
UIState = str2double(S(18));
```

4.1.2 Debugging VSX Generated UI Controls

Debugging a VSX generated UI.Callback or EF.Function requires a few extra steps, as these functions don't exist until VSX is run and are overwritten with each new run of VSX. Consequently, a breakpoint can't be set prior to running VSX. The easiest method for debug is to insert a 'keyboard' statement in the callback or external function at the point where one wants to break the execution. When VSX is run and the callback or external function invoked, the keyboard command will return control to the Matlab command line. At that point, one can bring up the function in the Matlab editor using the command:

```
>edit UserXXCallback.m
```

for a Verasonics callback, or for an external function

```
>edit myExtFuncName.m
```

Matlab will show the point of execution at the 'keyboard' statement in the function. One can then use the Matlab debugging commands to step through the code and observe the execution. If changes are to be made to the code, one should of course make these changes in the function definition in the SetUp script, and recompile the script before running VSX again. When the VSX generated function is working properly, the 'keyboard' statement can then be removed.

4.2 Setting Control Parameters

The runAcq function has a single input parameter, which is a structure array called 'Control'. This structure has two fields:

Control.Command – a string that represents a command.

Control.Parameters – a cell array that contains various parameters needed for the command given.

The valid commands and their associated parameters are as follows:

Control.Command	Control.Parameters
[]	{}; No parameters needed.
'set'	{'Object', objectNumber, 'attribute', value, 'attribute', value, ...} * returns without running event sequence
'set&Run'	{'Object', objectNumber, 'attribute', value, 'attribute', value, ...}
update&Run'	One or more of the following parameters: {'Resource','ImageBuffer','DisplayWindow','Parameters', … 'Trans','Media','SFormat','PData','TW','TX','Receive', … 'TGC','Recon','Process','SeqControl','Event'}
'copyBuffers'	{}; No parameters needed.
'setBFlag'	{}; No parameters needed.
'imageDisplay'	{ 'Image attribute', value, 'attribute', value, ...}
'debug'	{'on'}; or {'off'};

Control.Command is set to [ ] (the empty state) when no action is required. This is also the condition that is set after a Control.Command is executed by runAcq, so that commands only execute once.

It is possible to execute more than one Control.Command for a single call to runAcq. This is accomplished by defining multiple Control structures with an index, such as

```
Control(1).Command = 'set&Run';
Control(1).Parameters = {'ObjectX', objectNumber, 'attribute', value};
Control(2).Command = 'set&Run';
Control(2).Parameters = {'ObjectY', objectNumber, 'attribute', value};
```

In this situation, each Control command is executed in series, starting from index 1. The sequence events (for commands that don't return immediately) are executed only after all Control commands have completed.

For setting multiple attributes of the same object, note that the 'set' command can take a variable number of attribute, value pairs. Also note that if you set an attribute with the 'set' command, that attribute will not be updated in the Matlab environment. This can be

used as a feature, allowing you to set various parameters in the runAcq environment, and later restore the original state by doing an 'update' of the same objects. It can also lead to unexpected behavior, if you set some parameters and later try an update on the same object. When changing an object's attribute, it is usually a good idea to change the attribute first in the Matlab environment, regardless of whether you intend to perform an 'update' or a 'set' operation.

At the current time, not all structure attributes can be changed with the 'set' or 'set&Run' commands and there are some structures, such as the Resource structure that can't be updated during run time. The table below lists the currently supported structures (structure names to use in Control.Parameters are in single quotes) that have settable attributes or that can be completely updated. (See Section 4.1 for examples).

Structure	Settable Attributes	Update-able
Resource.ImageBuffer - 'ImageBuffer'	lastFrame	No
Resource.DisplayWindow - 'DisplayWindow'	Position ReferencePt pdelta clrWindow figureHandle imageHandle colormap	Yes
Resource.Parameters - 'Parameters'	speedCorrectionFactor startEvent simulateMode	Yes
Trans - 'Trans'	None	Yes
Media - 'Media'	None	Yes
SFormat - 'SFormat'	None	Yes
PData - 'PData'	None	Yes
TW - 'TW'	None	Yes
TX - 'TX'	None	Yes
Receive - 'Receive'	None	Yes
TGC - 'TGC'	None	Yes
Recon - 'Recon'	None	Yes

Structure	Settable Attributes	Update-able
Process - ‘Process’	Image class: method imgbufnum framenum sectionnum srcData pdatanum norm pgain interp persistMethod persistLevel compression mappingMode threshold displayWindow display extDisplay Doppler class: method srcbufnum srcframenum Srcsectionnums srcpagenum pdatanum dstbufnum prf numPRIs wallFilter pwrThreshold maxPower powerCeilingFactor postFilter External class: method srcbuffer srcbufnum srcframenum srcsectionnum srcpagenum dstbuffer dstbufnum dstframenum dstsectionnum dstpagenum	Yes
SeqControl - ‘SeqControl’	None	Yes, but keep same no. of structures and don’t modify ‘transferToHost’ objects.
Event - ‘Event’	None	Yes, but changing no. of Events or acquisitions may confuse DMA transfers.

5. An Example Script – 'SetUpL11_4vFlash.m'

This script generates all the sequence objects needed to acquire and process ultrasound images produced with an unfocused transmit pulse from a typical linear array transducer, in this case, a Verasonics L11-4v transducer. Each element in the transducer transmits a burst of ultrasound at exactly the same time, producing a flat wavefront that travels out from the transducer. On receive, all 128 channels are used to acquire the echo data from the full field of view.

This sequence illustrates the use of asynchronous acquisition and processing, where the frame rate of acquisition doesn't necessarily match the frame rate of processing. The acquisition of frames into memory can be at a fairly high rate, while the rate of processing can be at something reasonable for real-time viewing, such as 30fps. With only a few minor changes, the sequence can be converted to provide synchronous acquisition and processing. A cineloop for both acquired RF data and processed image data will be created, either of which can be reviewed in a playback mode.

Since this is a fairly simple script, one could likely program the relevant structures without an outline, but in general it is a good idea to first describe the actions needed in the sequence.

Sequence actions for simple flash (plane wave) imaging:

1. Transmit a single pulse with all transmit delays set to 0, generating a plane wave that insonifies the entire image space. Transmit at the center frequency of the transducer.
2. Receive the data from all channels using 200% bandwidth Nyquist sampling and the default digital filters. Program the Receives to transfer data into a 100 frame RcvBuffer.
3. Program a time between acquisitions that sets acquisition frame rate at 100 fps.
4. Transfer the frame of acquisition data to the host after each frame is acquired.
5. Reconstruct the most recently acquired image.
6. Process the reconstructed image with mostly default parameters and display the resulting image.
7. Capture a 20 frame display cineloop for review.

We can now proceed to write our Setup script. The lines in Courier font are the Matlab code, while the lines in blue italics are for explanation of the programming steps.

5.1 Define system parameters.

First, be sure to start your scripts with a 'clear all' statement to eliminate variables that might be present in the Matlab workspace. This is important as the new variables and structures defined will be saved at the end of the SetUp script. Next, it is best to define some global and system parameters at the top of the file. The parameters are defined as follows:

Define some fundamental parameters of the script. Parameters that are prefaced with a P are saved along with other values when a Preset is saved.

```
clear all
P.startDepth = 5;   % define the acquisition range in wavelengths
```

```
P.endDepth = 192;

% Define system parameters.
```

Define the number of transmitters available in the system. For a 128 channel Vantage system, or a 256 channel system using a single connector, this number is 128.

```
Resource.Parameters.numTransmit = 128;  % number of transmit channels.
```

Define the number of receive channels available in the system. For a 128 Vantage system, or a 256 channel Vantage system using one connector, this number is 128.

```
Resource.Parameters.numRcvChannels = 128; % number of rcv channels.
```

Define the speed of sound in the media to be imaged in meters/sec. For medical imaging, the speed of sound is an average of the various tissues, typically 1540 m/s. This parameter, if defined differently from the default of 1540 m/s, should be defined before computing the transducer characteristics in Trans.

```
Resource.Parameters.speedOfSound = 1540; % speed of sound in m/sec
```

Define the level of error reporting. Level 2 will report most system errors.

```
Resource.Parameters.verbose = 2;
```

The simulateMode attribute specifies whether you want to run your script in simulate mode, or with acquired data from the Vantage hardware. A value of 0 means that you would like to use the Vantage hardware for acquisition. If the hardware is not detected, VSX will automatically switch to simulate mode, which is set with a value of 1. In simulate mode, you can choose whether to simulate the RF data from a Media model which you can define (simulateMode = 1), or whether to simply process whatever data is already in the RcvBuffer memory (simulateMode = 2). VSX creates a GUI toggle control to switch to simulateMode = 2 (named RcvDataLoop) from either simulateMode = 0 or 1.

```
Resource.Parameters.simulateMode = 0;
%  Resource.Parameters.simulateMode = 1 forces simulate mode, even if
%    hardware is present.
%  Resource.Parameters.simulateMode = 2 stops sequence and processes
%    RcvData continuously.
```

5.2 Define the Transducer characteristics.

The Trans structure array defines the characteristics of the transducer attached to the system or used for simulation.

```
% Specify Trans structure array.
```

Set the name of the transducer. In this case, the L11-4v is a known transducer type, and the rest of the transducer specification can be filled in by using the utility function – 'computeTrans'. The units can be specified in either 'wavelengths' or 'mm'. We can also set a safe limit for the High Voltage supply.

```
Trans.name = 'L11-4v';
Trans.units = 'wavelengths';
```

```
Trans = computeTrans(Trans);
Trans.maxHighVoltage = 50;
```

The additional Trans attributes computed by the utility function can be viewed after running the SetUpL11_4vFlash script by typing 'Trans' at the Matlab command prompt.

For better understanding, some of the added attributes are described below.

```
Trans.frequency = 6.25;
```

This attribute is the center frequency of the transducer, Fc. This value is used to determine the A/D sample rate for acquisition, which will be typically be set to the nearest realizable four times Fc rate.

Fc also specifies the conversion of wavelengths to time and or distance (given the speed of sound). Since time and distance attributes are specified in wavelengths, it is important to provide an appropriate center frequency. [Note: Specifying time and distance in wavelengths allows everything to scale appropriately when either center frequency or speed of sound are changed. For example, specifying a depth of acquisition in wavelengths allows the depth to scale up or down with a change to a different scanhead frequency, matching penetration, which scales inversely with frequency.] Important: If you wish to use a different Fc than the default specified by computeTrans, specify it before calling computeTrans so the function can compute wavelengths using your Fc value.

```
Trans.type = 0;
```

A transducer type of 0 specifies a linear array. Other types supported are 1 (curved linear), and 2 (2D array). A linear array has elements whose elements can be specified with a single x coordinate value.

```
Trans.numelements = 128;
```

The number of elements in the transducer.

```
Trans.elementWidth = 1.0958;
```

The element width. Note that this could be different than the element spacing below.

```
Trans.spacingMm = .300; % mm
Trans.spacing = 1.2175; % Wavelengths
```

The element spacing is sometimes known as pitch. The element spacing is provided by computeTrans.m in both mm and wavelengths.

```
Trans.ElementPos = zeros(Trans.numelements,4);
Trans.ElementPos(:,1) = Trans.spacing * …
        (-((Trans.numelements-1)/2):((Trans.numelements-1)/2));
```

The position of all transducer elements in the array. This array has a row for each element with four values: x, y, z, and alpha. The angle alpha is in radians, and specifies the angle of the normal of the element with the z axis. This parameter is typically only used for curved linear arrays. The x coordinate value is the only parameter we need to set for a linear array, and it is set to the position of the center of the element. The z axis passes through the center of the transducer array, between the central elements.

```
Theta = (-pi/2:pi/100:pi/2);
Theta(51) = 0.0000001;
eleWidthWl = Trans.elementWidth * Trans.frequency/speedOfSound;
```

```
Trans.ElementSens = abs(cos(Theta) * sin(eleWidthWl*pi*sin(Theta))./ …
        (eleWidthWl*pi*sin(Theta)));
```

The element sensitivity curve, sometimes known as a directivity pattern is computed using the above formula. This curve represents the fall off in intensity of an element's response to an echo coming in at an angle theta from the normal of the element (and also the fall off in intensity of the transmit intensity with angle). In this case, we are using a formula that calculates the curve for a narrow bar type element. If we knew the exact curve, as could be measured with a hydrophone in a water tank, we could enter these values directly.

```
Trans.impedance = [53x2 double];
```

For some transducers, the element impedance has been measured and provided by computeTrans. This attribute is used to compute power limits and to simulate the transmit waveform in simulate mode.

5.3 Define the Pixel Data region for reconstruction.

The Pixel Data structure array, PData, defines the region(s) of the transducer field of view that are to be reconstructed and possibly displayed. The overall PData pixel array is always rectangular in shape, and is typically sized to enclose the entire scan area of a linear array transducer. However, this rectangle can be sized and placed anywhere with respect to the transducer, and can be set to define a particular region of interest within the field of view. The pixel density of the PData array is defined independently of the DisplayWindow pixel density, and should be set to adequately sample the image resolution of the system. This resolution is determined by multiple factors, such as the transducer frequency, the size of the transducer aperture, and the length of the transmit burst. Typically, a pixel density between 0.4 and 1.0 wavelengths is adequate to represent most images. When specifying the location of a pixel, one should consider a pixel as an infinitesimal point.

```
% Specify PData structure array.
```

Set the PDelta(pdeltaX,0,pdeltaZ) attributes to a wavelength increment per pixel that is small enough to adequately sample the resolution of the image. We use separate pdeltaX and pdeltaZ values for the L11-4v linear array, since the depth resolution is typically much better than the lateral resolution. Note that the smaller the values, the longer the image or signal reconstruction will take for a given region.

```
PData.PDelta = [Trans.spacing, 0, 0.5];
```

Set the height of the PData region by specifying the number of rows.

```
PData.Size(1) = ceil((P.endDepth-P.startDepth) ...
                    /PData.PDelta(3));
```

Set the width of the PData region by specifying the number of columns. In this case we want the width to be equal to the width of the transducer.

```
PData.Size(2) = ceil((Trans.numelements * ...
         Trans.spacing)/PData.PDelta(1));
```

For 3D scans, the PData array would be 3 dimensional. In this case, it is only 2D, so we set the Size(3) parameter to 1 (single page).

```
PData.Size(3) = 1;
```

The PData.Origin is location of the upper left corner of the region. In this case we set the x dimension of the Origin to the center of the first transducer element. The z dimension is set to the start of the scan, as given by P.startDepth.

```
PData.Origin = [-Trans.spacing*(Trans.numelements-1)/2,0, ...
                P.startDepth];
```

The PData region definition above does not have a Region attribute. When loaded by VSX, the 'computeRegions' utility function will be called, which will create a single Region name 'PData' which will be the same size as the PData array.

5.4 Define a media model to use when in simulate mode.

Most scripts should define a media model to use for validating a script in simulate mode prior to running it with the VDAS hardware. The media model defines a collection of point targets, each with it's own reflectivity. It is also possible to provide a function that can be called during execution of the script that manipulates the point targets in some fashion. The following attributes are used in the L11_4vFlash script:

```
% Specify Media object.
```

In this script, we call a separate Matlab script to define the media model, named 'pt1'. This script defines the Model name, the location of the point targets along with their reflectivity, and the number of point targets.

```
pt1;
```

Specify a function to move the all the points in the x direction after each frame. When to call this function is specified in the Receive structures to be defined below.

```
Media.function = 'movePoints';
```

5.5 Define resources used by the sequence.

The Resource object defines memory resources needed by the system as well as some system parameters. See section 2.5 for a description of the various buffer types and how they are organized.

First we define a RcvBuffer for storing the acquired, individual channel RF data:

Set the datatype of the buffer to 16 bit signed integers. This is currently the only choice.

```
Resource.RcvBuffer(1).datatype = 'int16';
```

Set the rowsPerFrame attribute to something larger than the sum of all the channel acquisitions in the frame, since the acquisitions will be stacked down a column. In this case, there is only one acquisition per frame. Considering that a maximum depth acquisition would typically be less than 512 wavelengths (1024 round trip), or 4096 samples per acquisition (at 4 samples per wavelength), we can set a maximum rowsPerFrame of 4096 samples.

```
Resource.RcvBuffer(1).rowsPerFrame = 4096;
```

Set the colsPerFrame attribute to the number of channels available, in this case, 128.

```
Resource.RcvBuffer(1).colsPerFrame=Resource.Parameters.numRcvChannels;
```

Set the numFrames attribute to the number of frames to be captured in the RcvBuffer. The total number of frames should be greater than the number of frames that can be acquired during the processing time for a frame. In this way, we won't overwrite the data in a frame that is being processed. For example, if acquisition is set to acquire 100 frames per second, and we are able to process 50 frames per second, acquisition will acquire two frames while we are processing a frame. In this case, we should have at least two frames in our RcvBuffer (probably 3 or more for good measure, as some processing times may be longer due to the variabilities of CPU scheduling). For this script, we are setting a large number of frames (100) so as to provide a 1 second RF cineloop (at a 100 fps acquisition rate).

```
Resource.RcvBuffer(1).numFrames = 100;
```

There is no need for an InterBuffer for storing the complex signal pixel reconstruction data if we are not using synthetic aperture reconstructions. We can reconstruct directly to intensity data in an ImageBuffer. The 10 frame ImageBuffer will receive the output of the image reconstruction, but won't have any of the additional processing needed for display. If we want to also have access to the last frame of IQ reconstructed data, we can define a single frame InterBuffer. If we don't define rows and cols, the InterBuffer and ImageBuffer will be sized the same as PData.Size.

```
Resource.InterBuffer(1).numFrames = 1;
Resource.ImageBuffer(1).numFrames = 10;
```

The next Resource to define is a DisplayWindow for displaying the processed echo intensity image. This buffer will define the attributes of the Matlab display window used to show the image. The DisplayWindow typically has a higher pixel density than the ImageBuffer, as the ImageBuffer must only adequately sample the resolution of the ultrasound image, where the DisplayWindow must contain enough pixels to present a reasonable sized image on a high definition display. The pixels of the ImageBuffer will be interpolated up to the pixel density of the DisplayWindow.

The DisplayWindow is created as a bit-mapped window with 8 bit pixels. A colormap is defined to set the mapping of echo intensity to gray scale values.

Set the title that will appear in the frame of the display window.

```
Resource.DisplayWindow(1).Title = 'L11-4vFlash';
```

Set the pixel density of the display window. This will determine the scale of the image data within the DisplayWindow. The larger the pdelta value, the smaller the image that will be rendered within the DisplayWindow.

```
Resource.DisplayWindow(1).pdelta = 0.35;
```

Set the position of the Display window on the computer's display. To center the displayWindow vertically, we obtain the ScreenSize array, which provides the width and height of the computer display. The position parameters are [leftEdge, bottomEdge, width, height]. In this case, we set the size of the DisplayWindow (when converted to wavelengths) to match the size of the PData region (in wavelengths). We also set the DisplayWindow x and y reference point to match the PData.Origin x and z reference point. If we wanted a black border around our PData region on the display, we would

set the DisplayWindow size somewhat bigger and the reference point slightly above and to the left of the PData.Origin. (Note that the reference point is the location of the upper left corner of the display window in wavelength units, with respect to the transducer coordinate system.)

```
ScrnSize = get(0,'ScreenSize');
DwWidth = ceil(PData.Size(2)*PData.PDelta(1)/ ...
          Resource.DisplayWindow(1).pdelta);
DwHeight = ceil(PData.Size(1)*PData.PDelta(3)/ ...
          Resource.DisplayWindow(1).pdelta);
Resource.DisplayWindow(1).Position = [250, ... % left edge
          (ScrnSize(4)-(DwHeight+150))/2, ... % bottom
          DwWidth, DwHeight];
Resource.DisplayWindow(1).ReferencePt = [PData.Origin(1), ...
          0,PData.Origin(3)];   % 2D imaging is in the X,Z plane
```

Set the number of frames for the DisplayWindow history buffer. The history buffer is a history of displayed frames.

```
Resource.DisplayWindow(1).numFrames = 20;
```

Set the type of units for the axes of the DisplayWindow. The default is 'wavelengths', but we can select 'mm' for millimeters.

```
Resource.DisplayWindow(1).AxesUnits = 'mm';
```

Set the colormap for the DisplayWindow. The attribute here is a function that returns an array of 256x3 double values, with each row an R,G,B setting.

```
Resource.DisplayWindow(1).Colormap = gray(256);
```

Occasionally, you may see other Resource attributes used in some of the example scripts. These are typically used for special purposes or for debugging.

This completes the definition of the non-event structures of our sequence, that is, those that are not involved directly in specifying acquisition and processing actions. The remaining structures to be defined will be referenced directly or indirectly in our Event sequence by their index numbers. Obviously, one must already have a very good idea of what events will compose the sequence to define the referenced structures, so in some cases, the user may want to rough out a sequence of events before attempting to define the referenced items. For a fairly simple sequence such as the L11_4vFlash, it is fairly easy to keep in one's head what the Event sequence will look like, allowing defining the referenced items up front.

5.6 Define the Transmit Waveform structure, TW.

The TW structure defines the characteristics of the transmit pulse. Each TW structure defines a single transmit waveform, so if we have several different waveforms, we have to create a separate TW structure for each one. However, for many scans, especially 2D echo scans such as defined by the L11_4vFlash script, it is sufficient to define a single waveform that is used for each transmit event.

Set the type of transmit waveform definition to use. A simple transmit waveform that can be generated by the Vantage hardware can defined the type as 'parametric'. Other more complex transmit waveform types are 'envelope' and 'pulse code'. If one is writing

a script for simulation only, additional types named 'function' and 'sampled' are available. (see section 3.2.1)

```
TW(1).type = 'parametric';
```

Set the parameters for a parametric waveform. The values represent: A, the transmit frequency in MHz (which should have a period that can be implemented with an even number of 250MHz clock cycles); B, the duty cycle of a half cycle period, expressed as a fraction of 1.0 (used for transmit apodization); C, the number of half cycle periods in the transmit burst; D, the polarity of the first half cycle. In this instance, we want to set a frequency as close as possible to our transducer center frequency of 6.25 MHz, which happens to have a period of exactly 40 250MHz clock cycles. We set the half cycle on time to some fraction of the half cycle period to better approximate a sine wave, in this case 0.67. For this transmit waveform, we want a short burst of 1 cycle, so we set the number of half cycle periods to 2. Finally, we set the initial polarity of the first half cycle to 1, which is positive. In this case, we want all transmitters to use the same waveform, so we only have to specify a single set of parameters.

```
TW(1).Parameters = [Trans.frequency,0.67,2,1];   % A, B, C, D
```

5.7 Define the Transmit events structure, TX.

The TX structure defines the characteristics of each transmit event used during the scan. This includes the transmit waveform, the focal point (if any), the steering angle, and the apodization function. For the L11_4vFlash scan, we only need a single transmit specification, which we will use to generate a flat transmit wavefront.

```
% Specify TX structure array.
```

Specify the transmit waveform to use for this transmit.

```
TX(1).waveform = 1;            % use 1st TW structure.
```

Set the origin point for the transmit beam. In the case of a flat wavefront transmit, this is not particularly meaningful, and we just set the origin as the center of the aperture. For a focused beam, the origin parameter would be point on the transducer where the beam appears to originate.

```
TX(1).Origin = [0.0,0.0,0.0]; % flash transmit origin at (0,0,0)
```

The next two attributes, 'focus', and 'Steer' are needed if one wants to use the utility function, 'computeTXDelays' for computing the transmit delays.

Specify the transmit focal point. For an unfocused beam, this is set to 0.

```
TX(1).focus = 0;
```

Specify the beam steering to use for this transmit. The steering can be specified with two angles: θ specifies the angle of the beam projection into the x,z plane from the positive z axis (azimuth), and α specifies the angle of the beam with respect to the x,z plane (elevation).

```
TX(1).Steer = [0.0,0.0];       % theta, alpha = 0.
```

Specify the transmit apodization function to use for this transmit. For transmit events that use the Vantage hardware, the TX.Apod values can range from 0 to 1, with 0 turning a transmitter off. The B parameter of TW.Parameters in the TW waveform

structure reference by this TX will be modified by the TX.Apod values (the duty cycle fraction of the half cycle period will be multiplied by the TX.Apod value for each transmitter).

```
TX(1).Apod = ones(1,Trans.numelements);
```

In this example, we have chosen to use the utility function, 'computeTXDelays', to calculate the transmit delay times for each transmitter. These delay times are computed in wavelengths of the Trans.frequency attribute, which translate to times based on the speed of sound defined in the Resource.Parameters.speedOfSound attribute.

```
TX(1).Delay = computeTXDelays(TX(1));
```

5.8 Define the Time Gain Control waveform structure, TGC.

The TGC structure defines the initial time gain control curve to be used with the VDAS hardware. Multiple curves can be defined, but typically one curve is sufficient, which is used by all channels. The curve is defined by eight control points, which represent the position of the TGC slider controls on the GUI panel. The curve always starts at 0, and the values range from 0 to 1023 (max. gain).

```
% Specify TGC Waveform structure.
```

Define the initial value of the control points.

```
TGC.CntrlPts = [0,200,344,452,606,747,870,920];
```

Set the maximum range over which the TGC curve will be scaled. The scaling is needed to adapt the curve to different depth settings.

```
TGC.rangeMax = P.endDepth;
```

Compute the curve using the utility function 'computeTGCWaveform'.

```
TGC.Waveform = computeTGCWaveform(TGC);
```

5.9 Define the receive events structure, Receive.

The Receive structures define the receiver characteristics of each acquisition event. There should be a separate Receive specification for each acquisition event in the sequence that goes to a unique location in the RcvBuffer. In the L11_4vFlash scan, there is a single transmit/receive event per frame. Since there are potentially a large number of Receive structures, it is best to predefine the structures with default attributes, then over-write the default attributes with the desired ones in a loop. The definition of the default set of attributes is as follows:

```
maxAcqLength = ceil(sqrt(P.endDepth^2 + ...
                    ((Trans.numelements-1)*Trans.spacing)^2));
Receive=repmat(struct('Apod', ones(1,Trans.numelements), ...
                      'startDepth', P.startDepth, ...
                      'endDepth', maxAcqLength, ...
                      'TGC', 1, ...
                      'bufnum', 1, ...
                      'framenum', 1, ...
                      'acqNum', 1, ...
                      'sampleMode', 'NS200BW', ...
```

```
'mode', 0, ...
'callMediaFunc', 1), ...
 1,Resource.RcvBuffer(1).numFrames);
```

The Receive.Apod attribute sets the individual receive channel gain. A value of 0 turns a receive channel off, writing all zeros to the channels memory. The value of one is set here for all channels, meaning we want full amplitude on channel outputs. The number of values in the Receive.Apod array is typically equal to the number of elements in the array for non-multiplexed transducers.

The Receive.startDepth attribute determines the range at which A/D sampling begins, and is set to the same wavelength value set by P.startDepth.

The Receive.endDepth attribute determines the range at which A/D sampling ends. For this attribute, we need to set a larger value than found in P.endDepth, since to reconstruct pixels at P.endDepth, we need to acquire RF data for the longer path lengths of elements that are not directly over the reconstruction pixel. One can compute the worst case longest path length for the L11-4v as the square root of the sum of the squares of P.endDepth and the transducer aperture. For a P.endDepth value of 192 wavelengths, and an aperture size of 128 wavelengths, the worst case path length is then approximately 230 wavelengths (maxPathLength).

The Receive.TGC attribute is the number of a Time Gain Control waveform defined in a TGC structure (see the previous object defined). This curve defines how the receiver gain increases with time over the depth of the scan.

The next three attributes, 'bufnum', 'framenum', and 'acqnum' define where the acquisition data for a Receive goes in the host memory RcvBuffer. Recall that the RcvBuffer (in this case, RcvBuffer number one) was defined with multiple frames. Within each frame, each column contains all the acquisition data for a corresponding channel, in separate acqnum segments, which are packed consecutively down the column. Note that the acquisition data are first stored in local memory on the VDAS modules, so these attributes specify the target location for the acquisition data when they are transferred to host memory.

The Receive.sampleMode attribute is set to 'NS200BW, which will translate to a sample rate for the RcvBuffer of four times Trans.frequency (set to the nearest realizable sample rate based on the 250MHz master clock).

The Receive.mode attribute specifies how a receive channel is to store its RF data. For Receive.mode = 0, the RF data replaces data in the local memory. For Receive.mode = 1, the acquired data are added to the data already in local memory. This mode allows accumulating a number of acquisitions in local memory before transferring the accumulated sum to the RcvBuffer (for usage, see coding examples section).

*The Receive.LowPassCoef and Receive.InputFilter attributes are missing in our specification, meaning that we will use the default values that the system provides. These attributes can be examined in the Matlab workspace **after** running our script with VSX. The filter responses for Receive(1) can be plotted using the following Matlab commands:*

```
>> freqz([Receive(1).InputFilter,Receive(1).InputFilter(20:-1:1)])
>> freqz([Receive(1).LowPassCoef,Receive(1).LowPassCoef(11:-1:1)])
```

The last attribute, 'callMediaFunction', is used only in simulate mode. When set to 1 for a Receive structure, the Media.function name provided in the Media structure is called before acquiring simulated data. This allows moving the media points between frames so that the simulation can process moving targets.

The following loop sets the attributes of individual Receive structures:

```
% - Set event specific Receive attributes.
for i = 1:Resource.RcvBuffer(1).numFrames
    Receive(i).framenum = i;
end
```

In this loop, which modifies the receive structure associated with each acquisition frame, the parameter that is being set is:

Receive(i).framenum – The frame number to be used in the RcvBuffer for the acquisition data.

5.10 Define the reconstruction structure, Recon.

The Recon structure defines the details of the image/signal reconstruction process. This is the process that computes the complex signal or magnitude values at the pixel locations defined in the PData.Region structures. The input data for a Recon structure is always RF data in a RcvBuffer, and output data is directed to an InterBuffer for complex signal data and/or an ImageBuffer for magnitude data. There can be multiple parts to a reconstruction; for example, in a synthetic aperture reconstruction that combines the complex signal data from separate acquisitions. The multiple parts of a reconstruction are defined using ReconInfo structures, which will be described below.

```
% Specify Recon structure arrays.
Receive=struct('senscutoff', 0.6, ...
```

The single Recon structure is defined in a structure definition. The first attribute, the 'senscutoff' parameter represents a threshold from 0 to 1.0 that determines when an individual element contributes to a reconstruction. The Trans.ElementSens function is used to determine an element's relative sensitivity to the pixel point being reconstructed – if the sensitivity is above the 'senscutoff' value, the element's signal is included in reconstruction, otherwise not.

Set the PData structure number that this Recon will use. This structure defines the pixel locations that will be reconstructed.

```
                'pdatanum', 1, ...
```

The 'rcvBufFrame' is an optional parameter that can be set to over-ride the RcvBuffer frame number specified in the ReconInfo structures that are associated with this Recon. If the value is a positive integer, it indicates directly the frame number to be used. If set to a -1, as in this case, it indicates that the most recently transferred acquisition frame should be used.

```
                'rcvBufFrame', -1, ...
```

The InterBuffer frame is used for storing IQ data from the reconstruction. In this case, the InterBuffer is actually not required, as we will only have one acquisition per frame, which can be reconstructed directly to intensity in the ImageBuffer. Including an

InterBuffer definition [buffer 1, frame 1] in a 'replaceIntensity' reconstruction will output the IQ data as well as the intensity data, even though this script has no further use of this data.

```
                'IntBufDest', [1,1], ...
```

Set the ImageBuffer frame to use for storing magnitude data from the reconstruction. In this case, we use buffer number one, and set the frame number to -1, which means use the next available frame in the buffer. This allows capturing a history of output frames that can be examined in 'freeze' mode.

```
                'ImgBufDest', [1,-1], ...
```

Define the index numbers of the ReconInfo structures to be used with this Recon. If multiple ReconInfo structures are referenced, they should be defined in a row array or vector. In this case, there is only one RINum to specify.

```
                'RINums', 1);
```

5.11 Define the ReconInfo structures.

The ReconInfo structure(s) define the details of each step in a reconstruction process. Each ReconInfo structure's index is referenced in a Recon structure and we must define a ReconInfo structure for each reference. Multiple Recon structures must not reference the same ReconInfo structures. The ReconInfo structures reference the transmit and receive structures used to acquire the acquisition data, and specify the region of the PData array to reconstruct. They also define a mode of reconstruction, that determines whether to output complex signal data or magnitude data, and whether to replace the data in the output buffer or accumulate with data already in the buffer.

```
% Define a single ReconInfo structure.
ReconInfo = struct('mode', 'replaceIntensity', ...
                   'txnum', 1, ...
                   'rcvnum', 1, ...
                   'regionnum', 1);
```

The ReconInfo.mode attribute defines the output of the reconstruction and how to process it into the output buffer. This example uses the 'replaceIntensity' mode (see section 3.4.1 for a complete definition of the different modes). Mode 'replaceIntensity' outputs reconstructed intensity data, replacing the data in the output ImageBuffer.

The rcvnum attribute is set to 1, but in the Recon structure we have specified rcvBufFrame = -1, meaning that the 1 will get replaced during run time with the Receive index of the most recent frame acquired.

Note the regionnum attribute for this reconstruction is set to 1. We didn't define a PData.Region in the PData structure, but when we run the script, the computeRegions utility function will be called by VSX, which will add a single PData.Region structure which will include the entire PData array.

5.12 Define the Process structure(s).

The Process structures define the type of processing to be performed on the reconstructed pixel data, and set various parameters of that processing. The Process

structure consists of a class name, a method from within the class, and a list of Parameters (attributes). In the L11_4vFlash script, the only processing to be performed is to display the image in the DisplayWindow.

```
% Specify Process structure array.
```

Set the classname attribute to 'Image'. This is the class for methods that operate on image data within an ImageBuffer.

```
Process(1).classname = 'Image';
```

Set the method to 'imageDisplay'. This is the method to use for processing an image in an ImageBuffer and sending the result to a DisplayWindow.

```
Process(1).method = 'imageDisplay';
```

The list of parameters to use for processing. The first is the ImageBuffer to use.

```
Process(1).Parameters = {'imgbufnum',1,...
```

Set the frame number to process. In this case, -1 means process the last reconstructed frame.

```
                        'framenum',-1,... % (-1 => lastFrame)
```

Specify the PData structure index that describes the pixel locations for the frame.

```
                        'pdatanum',1,...  % PData struct to use
```

The 'pgain' value applies a gain factor to the reconstruction output. A value of 1.0 leaves the data unchanged.

```
                        'pgain',1.0,...   % processing gain
```

The 'reject value cuts off low level intensities before mapping intensities to the display. The value is the percentage of the lower quartile of the intensity range to reject.

```
                        'reject',2,...    % processing gain
```

Set the persistence method to 'simple', and the level to 20, which means that the intensity value output consists of 20% of the previous frame's intensity plus 80% of the new intensity value.

```
                        'persistMethod','simple',...
                        'persistLevel',20,...
```

Set the interpolation method. In this case, '4pt' means to use a 4 point bi-linear interpolation method. This is currently the only method supported.

```
                        'interpMethod','4pt',...    % interp. method
```

Set the 'grainRemoval', 'processMethod' and 'averageMethod'. The first two attributes are used to apply 3x3 spatial filters to the image data, while the last attribute is use to average 2 or 3 consecutive frames prior to display. In this case, 'none' means no processing applied.

```
                        'grainRemoval','none',...
                        'processMethod','none',...
                        'averageMethod','none',...
```

Set the 'compressMethod' and 'compressFactor'. The 'compressMethod' options are 'power' or 'log'. The 'power' option raises the intensity data to the power n, where n is a fraction set by the 'compressFactor' value (40 equates to 0.5). Additional compression

or expansion can accomplished by modifying the display windows colormap.

```
                    'compressMethod','power',...
                    'compressFactor',40,...  % X^0.5
```

Set the 'mappingMode' to display the intensity data over the full range of the grey scale/ color map. Color flow images divide the color map into grey and color halves.

```
                    'mappingMode','full',...
```

Setting the 'display' attribute to 1 means that we want to display the image after processing. There are some conditions where we don't want the display to appear until additional processing has been performed.

```
                    'display',1,...   % display image
```

Set the index of the Resource.DisplayWindow that we want the image to appear in.

```
                    'displayWindow',1};
```

5.13 Define the SeqControl and Event structures.

We are now at the point where we are ready to define the individual sequence events for our sequence. Each event can reference a transmit action, a receive action, a reconstruction action, and a processing action. Finally, a SeqControl action can be set that controls factors such as DMA transfers and event flow control. For actions that are not needed, a zero index is set, which indicates no action for that item. In the case of tx and rcv actions, these must both be set or only the tx index set (for a transmit only event). A receive only `Event` can have a tx reference to a `TX` structure with all transmitters disabled (`TX.Apod = zeros(128)`).

It is often easier to define the SeqControl structures at the point where they are needed in the sequence of Events. The exception to this approach would be for SeqControl structures that are used repeatedly, in which case, it is simpler to define them up front. In the SetUpL11_4vFlash script, we define four `SeqControl` structures initially.

```
% Specify known SeqControl structures.
SeqControl(1).command = 'jump'; % jump back to start.
SeqControl(1).argument = 1;
SeqControl(2).command = 'timeToNextAcq';  % time between frames
SeqControl(2).argument = 10000;  % 10 msec for 100fps
SeqControl(3).command = 'returnToMatlab';
```

Define a counter variable, nsc, to keep track of new SeqControl indices.

```
nsc = 4; % nsc is count of SeqControl objects
```

Define a counter variable, n, to keep track of new Event indices. This makes it easy to insert new Events in the sequence if needed.

```
n = 1; % n is count of Events

% Acquire all frames defined in RcvBuffer
```

The acquisition and reconstruction for all the frames in the RcvBuffer are defined in a loop. This makes it easy to change the number of frames acquired in the RF cineloop

by simply changing the Resource.RcvBuffer.numFrames attribute.

```
for i = 1:Resource.RcvBuffer(1).numFrames
    Event(n).info = 'acquisition';
    Event(n).tx = 1;          % use 1st TX structure.
    Event(n).rcv = i;         % use ith Rcv structure.
    Event(n).recon = 0;       % no reconstruction.
    Event(n).process = 0;     % no processing
    Event(n).seqControl = [2,nsc]; % use seqControl nsc defined below
       SeqControl(nsc).command = 'transferToHost';
       nsc = nsc + 1;
    n = n+1;
```

After the acquisition for each frame, set the 'timeToNextAcq' SeqControl to control the frame rate for the VDAS hardware, and request that the acquired data for the frame be transferred to the host memory RcvBuffer. We need a separate 'transferToHost" command for each frame that transfers to a unique location in the RcvBuffer, since these commands will be converted to unique 'DMA' commands.

Specify a reconstruction after each acquired frame. The hardware sequencer ignores these events and continues to acquire new frames while the software is doing reconstruction processing. The software processing ignores tx and rcv actions (unless in simulate mode), and with Recon.rcvBufFrame = -1, will try and process the most recently transferred frame of data. If the RcvDataLoop control is toggled, the reconstruction will process the next frame in the RcvBuffer from the last one processed, and will sequentially process every frame in the buffer.

```
    Event(n).info = 'Reconstruct';
    Event(n).tx = 0;          % no transmit
    Event(n).rcv = 0;         % no rcv
    Event(n).recon = 1;       % reconstruction
    Event(n).process = 1;     % processing
```

Since the RF cineloop might be quite long, resulting in a lot of frames reconstructed before reaching the end of the sequence, we insert a 'returnToMatlab' at every 5th frame reconstructed. This allows for more rapid response to GUI controls, which can only happen while back in the Matlab environment.

```
    if floor(i/5) == i/5    % Exit to Matlab every 5th frame
        Event(n).seqControl = 3; % return to Matlab
    else
        Event(n).seqControl = 0;
    end
    n = n+1;
end
```

The last event is a simply a jump back to the first event.

```
Event(n).info = 'Jump back to first event';
Event(n).tx = 0;          % no TX
Event(n).rcv = 0;         % no Rcv
Event(n).recon = 0;       % no Recon
Event(n).process = 0;
```

```
Event(n).seqControl = 1; % jump command
```

The above Events define an asynchronous sequence, meaning that the hardware sequencer runs independently of the software sequencer without any synchronization. To understand how this works, it is useful to remember that there are two sequencers operating in the system, one implemented in hardware on the VDAS modules, and the other in software in the runAcq.c mex program. These sequencers run independently, and respond only to the event actions that they can implement. The only event actions that the hardware sequencer operates on are `tx, rcv` and some `seqControl` actions. The hardware sequencer ignores `recon` and `process` actions. Alternately, the only event actions that the software sequencer operates on (when running with the Vantage hardware) are `recon, process`, and some `seqControl` actions.

From the hardware sequencer's perspective, our `Event` sequence says "Acquire `Resource.RcvBuffer(1).numFrames` frames with a time of 10msec between frames, transferring the acquisition data to host memory after every frame; then jump back to the start of the sequence and do it again." From the software sequencer's perspective, our Event sequence says "Do a reconstruction, using the frame data that was most recently transferred to the RcvBuffer (as indicated by the `Recon.rcvBufFrame = -1`), then process the reconstructed data and send it to the display. After 5 reconstruct/process frames, return to Matlab." In the case that no action is necessary in the Matlab environment, the Event sequence is re-entered at the Event after the 'returnToMatlab' command, and the software sequencer will do 5 more reconstruction/process actions. After `Resource.RcvBuffer(1).numFrames`/5 reconstruction/process sets, the jump back to the beginning of the sequence will occur, and the processing will repeat.

As long as the hardware frame acquisition is faster than the reconstruction processing time, a new image frame will be displayed after each reconstruction, and the frame rate will appear fairly uniform (the eye will have a hard time discerning time differences between displayed frames if the displayed frame rate is above about 25 fps.) In this example, we set the acquisition frame rate to roughly 100 fps (using the `'timeToNextAcq'` command on the acquisition for the frame). If our reconstruction/processing rate for a single flash image is around 40 fps (this will depend on computer processing power but typically it is much higher, at over 100 fps), we can display a new image every 2.5 acquisition frames. Since we can only process and display complete frames, our displayed frames will step through the RcvBuffer unevenly, possibly in the following fashion: 1, 3, 6, 8, 11, 13, 16, 18, 21, 23, 26 ... This uneven stepping is not discernible to the eye, which sees a uniform frame rate.

In RcvDataLoop mode (entered by pressing the toggle button on the GUI panel), the reconstruction/process action will step through all the acquired frames sequentially, and the acquired data will appear to play back in slow motion. In this mode, we are reconstructing and seeing all 100 fps of acquisition, and the time between frames is uniform. This is the mode in which any motion in the media should be measured, since the time between frames is precisely set by the `'timeToNextAcq'` value.

If one would like a synchronous sequence operation, where the hardware waits for the processing to complete before acquiring additional frames, the above sequence of Events requires only a few minor changes. The added statements are shown in a red color in the revised sequence of Events below (these changes are from the

'SetUpL11_4vFlashSync.m' file).

```
% Specify SeqControl structure arrays.
```

We can eliminate the previous SeqControl(2) 'timeToNextAcq' for frame rate, as the frame rate will now be determined by the processing rate.

```
SeqControl(1).command = 'returnToMatlab';
SeqControl(2).command = 'jump'; % jump back to start.
SeqControl(2).argument = 1;
nsc = 3; % nsc is count of SeqControl objects
```

Create a new variable to hold the index of the last 'transferToHost' SeqControl.

```
lastTTHnsc = 0;

n = 1; % n is count of Events
% Acquire all frames defined in RcvBuffer
for i = 1:Resource.RcvBuffer(1).numFrames
    Event(n).info = 'Acquisition';
    Event(n).tx = 1;         % use 1st TX structure.
    Event(n).rcv = i;        % use 1st Rcv structure.
    Event(n).recon = 0;      % no reconstruction.
    Event(n).process = 0;
```

Replace the following seqControl specification with the one below it in red.

```
    Event(n).seqControl = [1,nsc]; % SeqControl structs
       SeqControl(nsc).command = 'transferToHost';
       nsc = nsc + 1;
```

*Set the SeqControl.condition of the 'transferToHost' command to 'waitForProcessing', and set the SeqControl.argument to point to the previous 'transferToHost' SeqControl. This indicates that we want the processing of the data transferred by the previous 'transferToHost' SeqControl to complete before the **next** 'transferToHost' SeqControl command is executed. The 'timeToNextAcq' command is no longer needed, since the hardware sequencer will be synchronized to the processing rate.*

```
    Event(n).seqControl = nsc;
       SeqControl(nsc).command = 'transferToHost';
       SeqControl(nsc).condition = 'waitForProcessing';
       SeqControl(nsc).argument = lastTTHnsc;
       lastTTHnsc = nsc;
       nsc = nsc + 1;
    n = n+1;
```

The recon event can remain the same, since it automatically will wait for the 'transferToHost' command to complete before processing, and afterwards it will perform a 'markTransferProcessed' to release the hardware sequencer to perform the next 'transferToHost'.

```
    Event(n).info = 'Reconstruct';
    Event(n).tx = 0;         % no transmit
    Event(n).rcv = 0;        % no rcv
    Event(n).recon = 1;      % reconstruction
    Event(n).process = 1;    % processing
    n = n+1;
```

```
end
```

Fix the argument in the first synchronizing SeqControl structure to point to last transferToHost SeqControl. The first time the sequence executes, it will not wait for processing of this last transfer, since no data have been transferred yet.

```
SeqControl(3).argument = lastTTHnsc;

Event(n).info = 'Jump back to first event';
Event(n).tx = 0;         % no TX
Event(n).rcv = 0;        % no Rcv
Event(n).recon = 0;      % no Recon
Event(n).process = 0;
Event(n).seqControl = 2; % jump command
```

With these changes, the hardware acquisition frame rate will match the rate of reconstruction/processing. We can leave the Recon.rcvBufFrame set to -1 (process the most recent frame transferred) since with acquisition paced by processing, the most recent frame transferred will always be the one just acquired.

One final note: If our asynchronous L11_4vFlash sequence is executed in simulate mode (by setting Resource.Parameters.simulateMode = 1), the simulation of acquisition from our set of Media points will be initiated, and every acquisition frame will be processed. In effect, our sequence will turn into a synchronous sequence, since software is now processing all Event actions in series.

This completes the description of the parameters of the L11_4vFlash sequence. The remainder of the script defines optional GUI controls to be added to the GUI panel by VSX during load time. The creation of these controls is covered in section 4.1.

6. Coding Techniques

6.1 Averaging RF Data

Acquisition data can be accumulated into local memory on the Vantage modules prior to being transferred to the host Receive buffer. This can be useful for extracting small signals buried in noise, or for summing pulse inversion acquisitions for harmonic imaging. When accumulate mode (Receive.mode = 1) is used, the local memory on the VDAS modules is used to accumulate acquisition data, before any transfers occur to the RcvBuffer in host memory. Note that the local memory on the Vantage modules is only 16 bits wide, so with a large number of accumulations, there is a potential for saturation. If one is averaging very small signals buried in noise, this is typically not an issue, as the accumulation of the noise only grows as the square root of the number of acquisitions. Overflow detection is implemented in the accumulation adder, so accumulations that exceed the range for 16 bit signed integers will be clipped at the maximum positive or negative values.

For averaging two acquisitions (as used for pulse inversion), simply perform two acquisitions, the first using Receive.mode = 0, and the second using Receive.mode = 1, using the same Receive bufnum, framenum and acqnum specifications. When defining the Receive structures for the frame, define a mode 0 Receive as usual with an acqNum that increases by 1 from the previous mode 0 (for multiple acquisitions per frame). Then define the mode 1 Receive immediately following the mode 0, with matching bufnum, framenum, and acqNum attributes. The mode 0 and mode 1 Receives can also be defined as two separate groups, if more convenient - define all the mode 0 Receives for the frame, then define all the mode 1 Receives with identical bufnum, framenum, and acqNum attributes.

For averaging more than two acquisitions, it is useful to set up a looping sequence where the number of acquisitions averaged can be set with a loop counter. The Matlab code below is an example of a partial sequence of Events that uses several flow control SeqControl commands to accomplish this. The first set of Receives is defined exactly the same as a single acquisition using `Receive.mode = 0`, so these Receives will have ascending acqNum values, starting from 1. VSX will assign `Receive.VDASblockNum` values to these Receives as it usually does, starting from the first memory location in local memory. The next set of Receives, which can be defined immediately following the first set, will be identical to the first set, but with Receive.mode = 1. The bufnums, framenums, and acqNums for this second set should be the same as for the first set.

You can now set up your Events to accumulate as many acquisitions as desired, by using the two sets of Receives. The Event list will have a loop to do the accumulates, the number of which can be defined in a variable, 'numAccum'. If numAccum = 1, no accumulation is done, and the first set of acquisitions is transferred to the host computer.

```
numAccum = 11;

% Specify SeqControl Structures
%  - Set loop count.
SeqControl(1).command = 'loopCnt';
SeqControl(1).argument = numAccum-1;
```

```
%  - Jump to test loop count.
SeqControl(2).command = 'jump';
SeqControl(2).argument = [];  %
%  - Jump back to start of accumulate.
SeqControl(3).command = 'loopTst';
SeqControl(3).argument = [];
%  - Transfer data frame to host.
SeqControl(4).command = 'transferToHost';


% Specify Event Structures
% Acquire the first set of acquisitions for the frame
for j = 1:numAcqs
    Event(n).tx = j;            % use appropriate TX structure.
    Event(n).rcv = j;           % use 1st set of Receives
    Event(n).recon = 0;         % no reconstruction
    Event(n).process = 0;       % 0=no processing
    n = n+1;
end

Event(n).info = 'Set loop count for number of accumulates.';
Event(n).tx = 0;
Event(n).rcv = 0;
Event(n).recon = 0;
Event(n).process = 0;
Event(n).seqControl = 1;  % command = 'loopCnt', argument = numAccum-1
n = n+1;

Event(n).info = 'Jump to loop count test.';
Event(n).tx = 0;
Event(n).rcv = 0;
Event(n).recon = 0;
Event(n).process = 0;
Event(n).seqControl = 2; % 'jump-to-test' SeqControl
n = n+1;

SeqControl(3).argument = n; % Sets the jump event no. for start of accums.

for j = 1:numAcqs
    Event(n).tx = j;            % use appropriate TX structure.
    Event(n).rcv = j+numAcqs;   % use 2nd set of Receive
    Event(n).recon = 0;         % no reconstruction
    Event(n).process = 0;       % 0=no processing
    n = n+1;
end

SeqControl(2).argument = n; % Set jmp event for the 'jmp-to-test' SeqControl.

Event(n).info = 'Test loop count - if nz, jmp back to start of accumulates.';
Event(n).tx = 0;
Event(n).rcv = 0;
Event(n).recon = 0;
Event(n).process = 0;
Event(n).seqControl = 3;
n = n+1;

% - Transfer data to host
```

```
Event(n).info = 'Transfer data to host.';
Event(n).tx = 0;
Event(n).rcv = 0;
Event(n).recon = 0;
Event(n).process = 0;
Event(n).seqControl = 4;
n = n+1;
```

6.2 Conditional Sequence Event Coding

There are often times when one wishes to execute an event or series of events conditionally, depending on perhaps a GUI button press or other action. There are several ways of coding conditional events, and one should choose the method that best suits the application. Due to the asynchronous execution of the hardware and software sequencers, it is possible to get unexpected results if an inappropriate method is chosen. The methods for conditional execution of events are as follows:

a. **Change the Resource.Parameters.startEvent** - This is the preferred method for conditionally executing Events that can be placed in short sequences near the top of the Event list. Refer to the outline of events illustrated below.

One-shot Conditional Execution:

```
Event(1)
  ---
  ---
Event(i-1).seqControl -> 'jump' to Event j;
Event(i)
  ---
  ---
Event(j)
  ---
  ---
Event(k).seqControl -> 'jump' to Event j
```

The above sequence of events conditionally executes the events from Event(1) to Event(i-2) or from Event(i) to Event(j), depending on whether the startEvent is set to 1 or i. After executing the conditional Events one time, the execution will then loop on the Event sequence from Event(j) to Event(k). The conditional events will be executed whenever the Resource.Parameter.startEvent is set during run time, or when the sequence is frozen and un-frozen.

Either-Or Conditional Execution:

```
Event(1)
  ---
  ---
Event(i-2).seqControl -> 'call' events starting at Event k;
Event(i-1).seqControl -> 'jump' to event Event 1;
Event(i)
  ---
  ---
Event(j-1).seqControl -> 'call' events starting at Event k;
Event(j).seqControl -> 'jump' to event Event i;
Event(k)
  ---
  ---
Event(m).seqControl -> 'rtn'
```

The sequence above can be set to execute starting at Event 1 or Event i, by simply setting the Resource.Parameters.startEvent to 1 or i. For example, when set to start

at 1, the events from 1 to i-3 are executed, then a call is made to a subroutine of events starting at k. After the subroutine returns, execution jumps back to the first event and the sequence repeats. The subroutine contains the events that are to be executed in both cases, while the events from 1 to i-3 are only executed when the startEvent is set to 1 and the events from i to j are only executed when the startEvent is set to i.

An application where this method of coding might be used is in a script that changes the characteristics of a transmit beam for a sequence of acquisitions. The acquisitions for beam 1 can be coded after the first event, and the acquisitions for beam 2 can be coded after Event i. The reconstruction and processing events are the same for both cases and are coded in the subroutine starting at Event k. This allows precomputing of the transmit beam characteristics for both beams in the setup script and switching quickly between them.

The startEvent method of executing conditional Events is generally preferred over other methods, since it synchronizes the hardware and software sequencers whenever the startEvent changes. The change can only take place after a 'returnToMatlab' point in the sequence, so the time at which the change takes place with respect to processing can be controlled. At the 'returnToMatlab' point, any pending GUI actions are executed, and the 'set' of Resource.Parameters.startEvent causes the hardware and software sequencers to both start at the startEvent when the sequence resumes.

b. **Use the LoopCnt/LoopTst Commands to Implement a 'Switch' structure** - A typical switch structure in Matlab or C allows executing different code segments according to the value of a control parameter. A similar structure can be created within a sequence of events, provided there is only one use of the loopCnt/loopTst commands in the sequence. This approach also requires using the startEvent to specify different values of the loopCnt. The following sequence outline implements a 3 case switch.

```
Event(1).seqControl -> 'loopCnt' set to 0
Event(2).seqControl -> 'jump' to Event 6
Event(3).seqControl -> 'loopCnt' set to 1
Event(4).seqControl -> 'jump' to Event 6
Event(5).seqControl -> 'loopCnt' set to 2
Event(6)
  ---
  ---
  ---
  (common events here)
  ---
  ---
  ---
```

Case Events

```
Event(i).seqControl -> 'loopTst' dec if nz & jump to Event(j)
  ---
  (case 0)
  ---
Event(j-2).seqControl -> 'loopCnt' (re)set to 0
Event(j-1).seqControl -> 'jump' to Event n;
Event(j).seqControl -> 'loopTst' dec if nz & jump to Event(k)
  ---
  (case 1)
  ---
Event(k-2).seqControl -> 'loopCnt' (re)set to 1
Event(k-1).seqControl -> 'jump' to Event n;
Event(k)
  ---
  (case 2)
  ---
Event(n-1).seqControl -> 'loopCnt' (re)set to 2

                                        Event(n)

  ---
  (common events here)
  ---
Event(m).seqControl -> 'jump' to Event 6 to repeat.
```

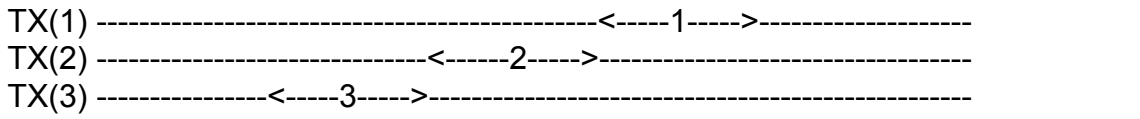
The initial value of the startEvent (1, 3 or 5) determines which case will execute, and that case will repeat with each pass through the sequence loop. During run time, the user can change the startEvent with a GUI control to select a different loopCnt, and that will cause a different case to execute. The hardware and software sequencer will be synchronized with each change of the startEvent value.

c. **Use the 'cBranch' (conditional branch) Command** - While this command may seem to be the most obvious way to conditionally execute events, it is difficult to use correctly. The problem is the asynchronous execution of the hardware and software sequencers. When the branch flag is set using a function call, the branch will be taken by the hardware and software sequencers ***when they next reach the event containing the 'cBranch' Sequence Control command***. And therein lies the problem, since the hardware and software sequencers may be at separate events when the branch flag is set, with perhaps the hardware sequencer ahead of the cBranch event and the software sequencer before it. This can lead to unexpected behavior that is difficult to analyze. The best use of the 'cBranch' command is to switch between two independent sequence loops that have no synchronous SeqControl commands. This allows either the hardware sequencer or the software sequencer to branch first without hanging up the sequence by either the hardware sequencer waiting for processing or the software sequencer waiting for DMA completion.

6.3 Combining RF Data for Multi-focal Zone Transmit/Receive Acquisitions

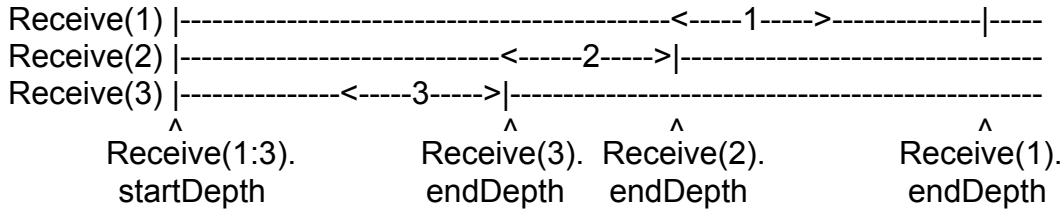
A frequently used method of acquisition for line mode scans involves transmitting along each line direction with different transmit focal zones and combining the RF data to achieve a type of 'dynamic' focus on transmit. This improves lateral resolution over a larger depth of field, at the expense of longer line acquisition times (which may not present a problem for short depths or non-moving targets). This method of acquisition can be implemented in the Verasonics system without impacting DMA transfer times or processing times by using the following programming steps:

1) The multiple transmit focal zones are defined with different TX structures. For example, with 3 transmit zones, one might define the focal points, as follows:



Depth is increasing from left to right in the above drawing and the depth of field for each focal zone is shown be the left and right arrows. The object is to combine the receive acquisitions for the 3 different focal zones so that the effective depth of field can be extended.

2) Define the Receive startDepth and endDepth attributes of the structures that correspond to the TX structures, starting with the deepest depth first. All Receives should start acquiring at the same startDepth, and use an endDepth that extends just to the end of the corresponding TX focal zone.



The first Receive's endDepth is set to acquire to the depth of the scanning region. The other Receives are set to acquire just to the end of there corresponding focal zone.

3) Define the `Receive.acqNum` values so that the acquisition for Receive(2) overwrites that of Receive(1) and Receive(3) overwrites that of Receive(2).

```
Receive(1).acqNum = 1;
Receive(2).acqNum = 1;
Receive(3).acqNum = 1;

Receive(4).acqNum = 2;  % Receive for deepest zone of next ray.
```

Note: the acqNum value for Receive(4) of the next ray line is set to 2, and must be one greater than the maximum acqNum defined previously to allocate new storage for Receive(4), rather than overwriting a previous Receive.

4) Define the Event structures to acquire the RF data for the deepest zone first, followed by the shallower zones next.

```
Event(n).info = 'Acquire deepest zone first.';
Event(n).tx = 1;
```

```
Event(n).rcv = 1;
Event(n).recon = 0;
Event(n).process = 0;
Event(n).seqControl = 1; % Use an appropriate 'timeToNextAcq' here
n = n+1;

Event(n).info = 'Acquire zone 2';
Event(n).tx = 2;
Event(n).rcv = 2;
Event(n).recon = 0;
Event(n).process = 0;
Event(n).seqControl = 1;
n = n+1;

Event(n).info = 'Acquire zone 3';
Event(n).tx = 3;
Event(n).rcv = 3;
Event(n).recon = 0;
Event(n).process = 0;
Event(n).seqControl = 1;
n = n+1;
```

For a 128T/64R VDAS system, one might want to use synthetic aperture acquisitions, requiring a slight modification to our sequence programming. In this case, we will have two Receives for each TX focal zone, and we would define the Receives as follows:

```
Receive(1).acqNum = 1;  % Receive for 1st half of aperture for deepest zone, 1.
Receive(2).acqNum = 2;  % Receive for 2nd half of aperture for deepest zone, 1.
Receive(3).acqNum = 1;  % Receive for 1st half of aperture for zone 2.
Receive(4).acqNum = 2;  % Receive for 2nd half of aperture for zone 2.
Receive(5).acqNum = 1;  % Receive for 1st half of aperture for zone 3.
Receive(6).acqNum = 2;  % Receive for 2nd half of aperture for zone 3.

Receive(7).acqNum = 3;  % First Receive for next ray line.
    ------
```

The Event structures would then be defined as follows:

```
Event(n).info = 'Acquire 1st half of aperture for deepest zone, 1.';
Event(n).tx = 1;
Event(n).rcv = 1;
Event(n).recon = 0;
Event(n).process = 0;
Event(n).seqControl = 1; % Use an appropriate 'timeToNextAcq' here
n = n+1;

Event(n).info = 'Acquire 2nd half of aperture for zone 1';
Event(n).tx = 1;
Event(n).rcv = 2;
Event(n).recon = 0;
Event(n).process = 0;
Event(n).seqControl = 1;
n = n+1;

Event(n).info = 'Acquire 1st half of aperture for zone 2.';
Event(n).tx = 2;
```

```
Event(n).rcv = 3;
Event(n).recon = 0;
Event(n).process = 0;
Event(n).seqControl = 1;
n = n+1;

Event(n).info = 'Acquire 2nd half of aperture for zone 2';
Event(n).tx = 2;
Event(n).rcv = 4;
Event(n).recon = 0;
Event(n).process = 0;
Event(n).seqControl = 1; % Use an appropriate 'timeToNextAcq' here
n = n+1;

Event(n).info = 'Acquire 1st half of aperture for zone 3.';
Event(n).tx = 3;
Event(n).rcv = 5;
Event(n).recon = 0;
Event(n).process = 0;
Event(n).seqControl = 1;
n = n+1;

Event(n).info = 'Acquire 2nd half of aperture for zone 3';
Event(n).tx = 3;
Event(n).rcv = 6;
Event(n).recon = 0;
Event(n).process = 0;
Event(n).seqControl = 1;
n = n+1;
```

With the above programming, the resulting line of RF data in local memory will consist of the multiple parts from the 3 acquisitions. The transfer of the combined data to the host will then occur at the end of the frame and will consist of the same amount of data as if a single acquisition per line was performed (of the deepest zone). When an image reconstruction is performed on this combined data, using the deepest zone 1 for the `ReconInfo.rcvnum` value, the seams between the multiple acquisition segments will typically not show, as the reconstruction combines RF data taken from each zone at the zone boundaries, thus smoothing the transition. Since the TX.Origin is the same for each of the multiple focal zones on the line, any of the TX structures for the line can be used for `ReconInfo.txnum`.

Examples of a two transmit focal zone scan with the L7-4 transducer are included in the ExampleScripts directory for the current software release.

7. Using the Vantage System High Frequency Configuration

This section provides an overview of programming techniques, system features and constraints for transmit and receive operation at frequencies above 15 MHz. Other than the details noted below, developing a script for a high frequency probe is identical to the 'standard' frequency system configuration as explained above.

7.1 High Frequency Transmit

For the high frequency configuration, a different transformer is used with much broader bandwidth to support high frequency operation. While the functionality for defining a high frequency transmit waveform using the TW structure is identical to the standard system, the flexibility in defining arbitrary waveforms at higher frequencies is reduced. This is because there are two system transmit constraints that must be applied: minimum pulse duration, and the transformer saturation flux limit.

7.1.1 Minimum Pulse Duration

The High Frequency transmitter cannot produce consistent output levels less than three system clock periods, or 12 nsec (3/250MHz). Pulses of 1 or 2 clock periods duration are not harmful to the system but will produce unpredictable results that vary widely from channel to channel. Therefore the highest frequency periodic transmit waveform that can be produced is a 41.67 MHz square wave with 1.0 relative pulse width. Note also that as a result of this constraint, equalization pulses for the TW 'parametric' waveform type should not be used at some higher frequencies, since the period of equalization pulse is only half that of the transmit waveform. The table below summarizes transmit frequencies and relative pulse widths that can be produced above 15 MHz using the 'parametric' waveform definition:

Frequency MHz	Relative Pulse Width	Equalization Pulses
41.67	1.0	disabled
31.25	0.75 or 1.0	disabled
25.00	0.6, 0.8, or 1.0	disabled
20.83	0.5, 0.67, or 0.83	disabled
	1.0	could be enabled
17.86	0.42, 0.57, or 0.71	disabled
	0.86, or 1.0	could be enabled
15.63	0.37, 0.50, or 0.63	disabled
	0.75, 0.88, or 1.0	could be enabled

Table 7.1.1 Supported Transmit Frequencies and Pulse Width above 15 MHz

As an example, the following three lines of code define a 25 MHz two-cycle transmit waveform with relative pulse width of 0.8:

```
TW.type = 'parametric';
TW.Parameters = [25.0, 0.8, 4, 1];
TW.equalize = 0; % disable equalization pulses
```

The third Parameter's value of 4 represents the burst duration in half-cycles; note that when equalization pulses are disabled this must be an even integer to produce a waveform with no DC content.

7.1.2 Transformer Saturation Flux Limit

The transmit transformer flux limit restricts the maximum transmit voltage that can be used at low transmit frequencies to prevent saturation of the transformer core. This limit is 6.25 Volt-usec (4.2 Volt-usec on systems built before approximately September, 2015) for the High Frequency Configuration, much more restrictive than the 25 Volt-usec allowed with the standard-frequency transmitter. If a script defines a transmit waveform that will exceed this limit at the `Trans.maxHighVoltage` setting (see 2.1), the system will exit with an error condition stating the maximum voltage allowed to stay within the flux limit for that waveform. For example, a 2.5 MHz square wave has a half-cycle duration of 0.2 usec. Thus for a relative pulse width of 1.0, the maximum allowed transmit voltage for the High Frequency Configuration would be 21 Volts since:

```
21 Volts * 0.2 usec = 4.2 Volt-usec
```

For the same waveform using equalization pulses, the maximum allowed voltage would increase to 42 Volts. If in addition to the equalization pulses the relative pulse width was reduced to 0.7, the maximum allowed voltage would be 60 Volts. With equalization pulses enabled and 0.7 relative pulse width, any transmit voltage up to the system maximum of 96 Volts can be used for transmit frequencies of 4 MHz or higher.

7.2 High frequency Receive Acquisition

Three techniques are available for high frequency receive acquisitions, depending on the center frequency and bandwidth of probe.

7.2.1 4 samplesPerWave

The High Frequency Configuration can still use the same receive data acquisition scheme that is typically used on the standard frequency system, of RF data sampling rate set to 4x`Trans.frequency`. Since the highest A/D sample rate supported by the HW system is 62.5 MHz, this results in a 15.625 MHz maximum center frequency for the receive data, with the Nyquist bandwidth limit spanning from DC to 31.25 MHz. A conservative rule-of-thumb is that the receive digital signal path and Recon processing can provide very good performance over a range of +/- 60% from the acquisition center frequency, or roughly 6 to 25 MHz for the 62.5 MHz A/D rate. For any probe and application where the desired bandwidth falls within that range, one can use this acquisition scheme and achieve very good performance by specifying the bandpass filter coefficients in `Receive.InputFilter` to match the desired center frequency and bandwidth. Note that if you do not specify `Receive.InputFilter` the system will assign a default wide-bandwidth (Approx. 100%) filter centered at `Trans.frequency`, which will be decidedly sub-optimal if the actual center frequency is well above 15 MHz.

The example script "SetUpL22_14vFlashAngles" provided with the system is an

example of this operating state. In this script, `Trans.frequency` is set to 15.625 MHz so the receive A/D sample rate is 62.5 MHz. The transmit waveform is at a nominal frequency of 17.86 MHz, and the `Receive.InputFilter` coefficients have been chosen to provide a bandwidth of 12 to 24 MHz, centered on 18 MHz. Refer to the comments in the script itself for more explanatory notes.

A script using this approach for the receive acquisition will work equally well on both the High Frequency Configuration and standard frequency system configurations. Note, however, that the transmit output amplitude and channel-to-channel uniformity will be significantly degraded on the standard frequency system at transmit frequencies above 15 MHz, as compared to the High Frequency Configuration.

Another important aspect of the High Frequency Configuration that must be considered for any of the possible receive acquisition schemes: The network that provides a programmable input impedance for the receive preamp on standard frequency systems has been modified in the High Frequency system so it provides a high-pass filtering function, rolling off at frequencies below 20 MHz. When using a probe on the High Frequency system whose bandwidth extends below 20 MHz (such as the L22-14v example cited above) this high-pass response must be disabled. This is achieved by programming the input impedance to its 'high impedance' state (see line 50 in the example script, and 3.3.3.1 for more information).

7.2.2 Interleaved sampling with 4 samplesPerWave

7.2.2.1 Overview

"Interleaved Sampling" is an acquisition scheme to sample above the 25 MHz upper bandwidth limit described above, for the typical acquisition with 4 samplesPerWave. The basic idea of interleaved sampling is to double the effective sample rate of the A/D converter by combining the samples from two successive transmit-receive acquisitions, with the sampling points in the second acquisition shifted by half of the A/D sampling period from those in the first. For example, two acquisitions at the 62.5 MHz sample rate can be combined to produce an effective sample rate of 125 MHz. All that is required is to interleave the samples from the two acquisitions, and shift the timing of the sample points by 8 nsec from one acquisition to the other. If the data being acquired is from a round trip transmit-receive event implemented on the Vantage system, the 8 nsec relative shift can easily be implemented by just adding 8 nsec of additional transmit delay to the transmit timing of one of the events relative to the other. Since transmit delay on the Vantage system is provided by a counter running at the 250 MHz system clock rate, the 8 nsec transmit delay offset is achieved by just adding two counts to the transmit delay values for the second event.

The only significant tradeoff for the interleaved sampling concept is the requirement to combine data from two successive acquisitions. This reduces the maximum achievable acquisition PRF by a factor of 2, and it also requires any motion of the tissue being imaged (relative to the transducer) to be no more than a small fraction of a wavelength, over the time interval required to complete the two acquisitions.

7.2.2.2 Implementation of Interleaved Sampling

The steps listed below provide an overview on how to implement the interleaved sampling mode, 'NS200BWI', with each step followed by some discussion and guidelines. Refer to the example script "SetUpL22_14vFlashAnglesInterleave.m" for an example of the full implementation of all of these steps. This script uses 2X interleave with an A/D rate of 41.7 MHz for an effective RF data acquisition sample rate of 83.3 MHz, allowing normal "4 samples per wave" reconstruction processing at a center frequency of 20.83 MHz. Read through the comments included in this example script, and review each of the implementation steps listed below.

1. Determine the closest supported center frequency to your value of `Trans.frequency` from the table below. In your Receive structures, set `Receive.sampleMode` to '`NS200BWI`' and `Receive.demodFrequency` to the center frequency in the table. The target interleave sample rate will 4 times the center frequency, and the A/D sample rate will be half the interleave rate.

CenterFreq.	4x SampleRate	VDASADClk	VDASADRate	VDASFilter
17.85	71.429	35.714	7	1
20.8333 MHz	83.3333MHz	41.6667MHz	6	1
25.0	100.0	50.0	5	1
31.25	125.0	62.5	4	1

2. Set the transmit delay offset between interleaved acquisitions. For the L22 example of 2X interleave from an A/D rate of 41.7 MHz (250 MHz / 6), half of a sample period is 3 cycles of the 250 MHz system clock, or 12 nsec, and thus this offset should be added to the transmit delay for all channels for the first acquisition. Adding the offset to the first transmit puts the A/D samples for this transmit ahead of the samples for the transmit with no delay. Since the transmit delay is specified in wavelengths of `Trans.frequency`, we need to convert the time period of 12 nsec to wavelengths of `Trans.frequency`. Since one wavelength represents 1/Trans.frequency of time, the fraction of a wavelength delay we wish to add is given by 12 nsec/(1/`Trans.frequency`). Since 12 nsec is actually 0.5 times (1/41.6667 MHz) we can compute the delay from the formula:

   delay(wavelengths) = `0.5` * (`Trans.frequency`/`41.6667`MHz)

   Note: For some 4x SampleRates (71.429 and 100 MHz), it is not possible to precisely program a 1/2 period A/D clock delay for TX.Delay, based on the 250MHz clock. Reconstruction accuracy may be slightly affected in these cases.

3. Set up the interleaved acquisition event sequence. For each interleaved acquisition, two separate transmit-receive events are requested in the sequence. The Receive structures for the two events must be identical except for incrementing the acqnum value, and the Transmit structures for the two must also be identical except for the `TX.Delay` offset added to the first of them, from step 2.

4. Set the digital RF data filters in the acquisition HW system: Note that since these filters are operating separately on RF data from each acquisition for 2X interleave, they cannot be used to implement an arbitrarily defined FIR filter for the interleaved RF sample. With respect to the interleaved output data, these filters represent an FIR filter structure with every other tap coefficient set to zero. This

means any filtering they provide will be mirrored around ¼ of the interleaved sample rate, producing a symmetric frequency response centered at Fs/4. As an example, one could define what appeared to be a high-pass filter in the `Receive.InputFilter`, and the resulting filter response for the interleaved output RF data would be a symmetric bandpass filter centered at Fs/4. Note that current Vantage SW releases do not provide a default setting of `Receive.InputFilter` for the interleaving operation state. Therefore the user should always manually specify the desired `Receive.InputFilter` coefficients in user setup script, even if for a flat 'pass through' filter. In future releases we may add additional logic in the system SW, to provide appropriately chosen default filter coefficients for interleaved sampling.

5. For reconstruction, the `Recon` and `ReconInfo` structures are defined as if the RF channel data were acquired at the higher interleave sampling rate. The `ReconInfo` should reference only the first Receive of the interleave pair of `Receives`, the one with the extra delay for transmit. The reconstruction software, when processing a `Receive` that has the '`NS200BWI`' sampleMode, will automatically interleave the two acquisitions in the `RcvBuffer`, so that the data in the `RcvBuffer` will appear as if it was acquired at the higher sample rate.

7.2.3 4/3 samplesPerWave

Another method Vantage system provides to support receive data acquisition and processing for frequencies above 25 MHz is "4/3 sampling", where the sample rate Fs is set to 4/3 x `Trans.frequency`. Therefore the transducer bandwidth is centered between Fs/2 and Fs and it will be aliased by the sampling process into the frequency range from 0 to Fs/2. This concept works quite well, but only within the confines of two fairly severe constraints: First, the usable imaging fractional bandwidth is restricted to less than 67%, since the Nyquist bandwidth of 0.5 Fs is centered at 0.75 Fs. Second, the system combined with the transducer must impose a very effective high-pass filter to reject the unwanted spectrum from DC to Fs/2. Any noise and detected signal in this spectrum will be mixed into the desired signals from Fs/2 to Fs, degrading the sensitivity, resolution and beamforming performance.

For an example of the 4/3 sampling method, one can imagine a probe with approximately 50% fractional bandwidth centered at 37.5 MHz (bandwidth extends from 28 to 47 MHz). The 'BS67BW' `sampleMode` would set the A/D sample rate to 50 MHz (37.5 times 4/3), yielding a Nyquist bandwidth of 25 MHz. The probe center frequency of 37.5 MHz will alias to 12.5 MHz in the RF data output samples. Note also that the receive data spectrum will be 'folded' by the aliasing: the upper band edge of 47 MHz will appear at 3 MHz in the output data, and the 28 MHz lower band edge will appear at 22 MHz. If the Recon processing function recognizes a `Receive.sampleMode` of 'BS67BW', the system automatically takes the aliasing and folding of the receive data into account, so the detected output data will be the same as what would have been achieved if sampling were performed at an effective 150 MHz sampling rate.

The biggest challenge with getting good performance using the 'BS67BW' `sampleMode` is to provide adequate high-pass filtering to reject signals and noise at frequencies below Fs/2 to avoid contaminating the aliased copy of the desired signals from Fs/2 to

Fs. Note that this filtering must be done in the analog signal path prior to the A/D since the sampling at the A/D is the point where the aliasing occurs. In the Vantage system we have provided two mechanisms for implementing the high-pass filter:

- Preamp input high-pass filter: In Vantage HW systems with the High Frequency Configuration, the feedback network at the input of the receive preamp has been modified to introduce a first order high-pass filter (for the standard configuration this network is used to allow programmable control of the receive input impedance with a flat frequency response). The cutoff for this filter is set to 20 MHz when the input impedance is set to its lowest level, as controlled by `RcvProfile.LnaZinSel` (see 3.3.3.1). The cutoff will move to progressively lower frequencies as the programmed input impedance is increased, and at the maximum "high Z" setting (`RcvProfile.LnaZinSel` = 31) the feedback network is disabled so the highpass filter is no longer present.
- "8/3" input sampling: If the processing center frequency set by `Trans.frequency` allows it, the system will automatically run the A/D converter at twice the desired output sample rate. In the digital filtering provided by the HW system the `Receive.LowPassCoef` attribute will be programmed to provide a high-pass function with a cutoff of ¼ of the A/D sample rate (2/3 Fc), followed by subsampling down to the output 4/3 sample rate. This 22nd order filter will provide much better rejection of the unwanted spectrum prior to subsampling than can be achieved with the analog preamp input high-pass filter, but the constrain is that it can only be used when it is possible to run the A/D at the 8/3 Fc sample rate.

Note that with either of these techniques, the bandpass filtering provided by `Receive.InputFilter` will typically have to be adjusted. Since it is operating on the aliased copy of the receive data spectrum, the apparent fractional bandwidth will be tripled (for example, if the probe has 50% fractional bandwidth centered on Fc, the aliased copy of that spectrum centered on 1/3 Fc will have an apparent fractional bandwidth of 150% relative to the aliased center frequency). The existing Vantage SW does not automatically take this into account when the 'BS67BW' `sampleMode` is being used, so the default `Receive.InputFilter` coefficients provided by the system may have a bandwidth that is too restrictive.

The achievable processing center frequencies and A/D sample rates for 'BS67BW' `sampleMode` probe center frequencies above 15 MHz on the Vantage system are listed in the table below:

Center Frequency MHz [Approx. usable bandwidth, MHz]	Output sample rate MHz	A/D sample rate MHz	Acquisition scheme
46.87 [35 – 58]	62.50	62.50	'BS67BW'
37.50 [28 – 47]	50.00	50.00	'BS67BW'
31.25 [23 – 39]	41.67	41.67	'BS67BW'
26.79 [20 – 33]	35.71	35.71	'BS67BW'
23.44 [18 – 29]	31.25	62.50	'BS67BW'
20.83 [16 – 26]	27.78	27.78	'BS67BW'
18.75 [14 – 23]	25.00	50.00	'BS67BW'

17.05	[13 – 21]	22.73	22.73	'BS67BW'
15.63	[12 – 19]	20.83	41.67	'BS67BW'

The example script "SetUpL22_14vFlashAngles67BW" provides an example of the "8/3 input sampling" approach. In this script, the center frequency is set at 18.75 MHz, so the desired 4/3 RF data sample rate is 25 MHz. The A/D runs at 50 MHz with the highpass filtering and subsampling by 2 provided in the digital signal path after the A/D. Refer to that example script and the explanatory comments within it for a detailed example of the concepts presented here.

8. History

2020-09-12 Added text to describe DisplayWindow.Type attribute in section 2.5.4. Added new ReconInfo modes in section 3.4.2. Completely rewrote section 3.6.1 to describe methods of acquisition and processing.

2019-11-30 Updates to section 2.2, Transducer Object; removed ScanFormat section.

2019-08-23 Added description of new reconstruction modes in section 3.4.1.

2019-06-21 Added description of dynamic MUX programming method to section 2.2.2.

2019-06-05 Documented use of -1 for destination frame of Image and Inter buffers in Recon and External processing functions.

2018-11-21 Corrected text in section 3.6 - software now allows more than three SeqControl commands in an Event.

2018-02-22 Added TX.FocalPt to TX structure with explanation. Added section 3.2.4.4 on paralleling transmit channels.

2018-01-30 Updates to Doppler processing method.

2017-03-16 Minor edits to clarify new attributes only available in software releases 3.2 and higher.

2017-01-31 Edits to sections 2.1, 2.2, 2.6, 3.3 and 3.6.

2017-01-18 Added drawings for PData with rectangular and polar coordinates in section 2.4.

2016-12-02 Document completely revised for software releases 3.0 and above.