User Guide to Changes in 4.0

Ken L. Verasonics Inc. Nov. 24, 2018

This document provides an overview of the changes in the Vantage 4.0.0 software release, in terms of how they affect a user's SetUp script and how to use the new features of the system in a script.

Table of Contents

1. New UTA Modules and "Vantage 32 LE" Product

As part of the 4.0.0 software release, the family of UTA modules available for use with the system nearly doubles in size with the addition of six new modules. A new "Vantage 32 LE" product configuration is also being added.

The Vantage product line available with the 4.0.0 release supports five system product configurations:

- **Vantage 32 LE** with 64 transmit, 32 receive channels
- **Vantage 64** with 64 transmit, 64 receive channels
- **Vantage 64 LE** with 128 transmit, 64 receive channels
- **Vantage 128** with 128 transmit, 128 receive channels
- **Vantage 256** with 256 transmit, 256 receive channels

The table on the following page lists the product configurations, software licensed options, and UTA Adapter modules that are available for each Vantage configuration. The color-coding of each cell indicates the availability status:

- **Green Highlight** represents a configuration that is listed for sale by Verasonics, with functionality that is fully supported, and verification tested.
- Yellow Highlight represents a configuration that is not listed for sale or actively promoted by Verasonics, but is fully functional.
- **Red Highlight** represents a configuration that is not currently listed for sale by Verasonics and cannot be used. If a system is mistakenly built in one of these configurations, it will report an error or fault condition and not allow ordinary user scripts to run. System diagnostic test utilities will continue to be functional to the extent possible, to allow testing of the system (For example, the illegal reported configuration may be an artifact of some other fault in the system hardware or software, or an error in the generation of a customer's SW license file. In situations such as these, access to the diagnostic utilities may be needed to diagnose and correct the problem.).

Notes in and following the table explain specific configurations that have limited functionality or are dependent on the state of other configuration options.

Vantage Configuration:	V- 32LE	V- 64	V- 64LE	V- 128	V- 256
Acquisition Module Configurations					
SF (Standard Frequency)					
HF (High Frequency)					
LF (Low Frequency)					
HIFU (Standard Frequency only)					
External Clock (Standard Frequency only)					
Software Licensed Options					
Extended Transmit **(Note 1)**					
Arbitrary Waveform					
Trigger in / out				(included)	(included)
Reconstruction Processing			(included)	(included)	(included)
HIFUPlex					
Impedance Measurement					
UTA Adapter Modules					
UTA 260-S	64 ch only	64 ch only			
UTA 260-D			64 ch only each Conn	64 ch only each Conn	
UTA 260-MUX					
UTA 360					
UTA 408				128 ch only	all 256 ch
UTA 408-GE				128 ch only	all 256 ch
UTA 160-DH/32 LEMO	32 ch only each Conn	32 ch only each Conn	64 ch only each Conn	64 ch only each Conn	full 128 ch each Conn
UTA 160-SH/8 LEMO	64 ch only	64 ch only			
UTA 128 LEMO	64 ch only	64 ch only			
UTA 64 LEMO					
UTA 160-SI/8 LEMO	64 ch only	64 ch only			
UTA 156-U					
UTA 256 Direct	64 ch only	64 ch only	128 ch only	128 ch only	
UTA 1024-MUX					

Note 1: Extended Transmit is not available as a stand-alone optional feature on Vantage 32 LE or Vantage 64. It will be enabled, however, if the system is configured with the Low Frequency option or if the customer has purchased the arbitrary waveform package.

Pre-UTA System Configurations: A pre-UTA Vantage 128 or Vantage 64 LE system configured with a single connector SHI will have exactly the same functionality and optional feature support as the corresponding UTA system in the table when configured with a UTA 260-S adapter module. Similarly, a pre-UTA Vantage 256 system configured with a dual connector SHI will have exactly the same functionality and optional feature support as the V-256 UTA system in the table when configured with a UTA 260-D adapter module

2. Simulation Operating States

In all earlier Vantage software releases there have been several different ways in which transmit-receive simulate operation could be invoked and used, with some differences in the functionality that was provided depending on how you did it. For 4.0.0 and future releases, two different simulation operating states have been defined, to make it easier to specify which one is desired and to clarify the level of functionality that will be provided. The two simulation states have been named "SimulateOnly" and "SoftwareOnly", as defined below:

- **"SimulateOnly"** identifies a user SetUp script that is intended **only** for simulation use and does not provide all of the parameters needed to run on the hardware system. In the 4.0.0 system software and within the Resource.System structure, the SimulateOnly name will be used to identify this operating state. The SimulateOnly operating state allows you to write a script using configurations that are not supported by the Vantage hardware system- for example, you can do full transmit-receive simulation with any arbitrary number of transducer elements in the range from 1 to 1024. A SimulateOnly script cannot be run on a hardware system, and if a hardware system is present it will be ignored.
- **"SoftwareOnly"** identifies simulation operation with a script that has been designed to run on one of the Vantage hardware system configurations, and thus provides all of the parameters needed to program the hardware as well as conforming to the hardware system constraints for that configuration. SoftwareOnly operation can be used to run the script when no hardware system is present, or when a hardware system is present that does not match the configuration needed by the script. "SoftwareOnly" also identifies an installation of the Vantage software on a computer that is not intended for use with a hardware system (this requires a software license keyed to the ID if that particular computer; see the Software Licensing section of the Vantage User Manual for more information). When a SetUp script is run in the SoftwareOnly operating state, the Vantage software will check and enforce all hardware system operating constraints and requirements for the hardware configuration identified by the script, but the script will only run in simulate mode. This allows you to run and debug a script when no hardware system is available, or when the connected hardware system does not match the configuration required by the script (in the SoftwareOnly operating state, the Vantage software will ignore the presence of the hardware system and will not attempt to communicate with it at all).

• Behavior of 'Resource.Parameters.simulateMode'

The meaning of this field is unchanged from earlier software releases: it can be used to select one of three operating modes defined as follows:

- simulateMode = 0 enables live RF data acquisition from the hardware system
- simulateMode = 1 disables data acquisition from the hardware system and enables the transmit-receive software simulation function, to generate synthesized receive data based on the array of scatterers defined by the user script in the Media structure.

- simulateMode = 2 processes previously acquired (or simulated) RF data from the Receive data buffer in the Matlab workspace.

Two toggle buttons in the GUI control window generated by vsx_gui can be used to switch between RF data loop playback or live acquisition, and to switch live data acquisition between software simulation and the hardware system. The value of the Resource.Parameters.simulateMode field is updated by these two toggle buttons, so it always matches the current operating mode of the system. The value of this field specified by the user's SetUp script determines the initial operating mode of the system to be used when it starts running.

If the system is running in any of the Software-Only or Simulate-Only operating states, the value of simulateMode will be restricted to 1 or 2 since live acquisition from a hardware system has been disabled. The simulate button in vsx_gui will be blanked since the system cannot switch out of simulation operation. If the user script had specified simulateMode = 0, VSX will automatically change it to 1 to set the initial operating mode when the script starts running.

For Vantage software releases 3.4 and earlier the behavior of simulateMode while a script was running was the same as described above, but during user script initialization it served a very different purpose- to select the operating state of the system:

- If simulateMode was set to zero, the system would go into the Live Hardware Acquisition operating state, with the initial mode set to live acquisition. The user could then toggle in and out of simulate mode as desired, but they had no control over the initial mode when the script starts up.
- If simulateMode was set to one, the system would go into the equivalent of what we are now calling the "SimulateOnly" operating state.

In summary, for the 4.0.0 and future Vantage software releases, Resource.Parameters.simulateMode will specify the initial operating mode of the script when it starts running, but no longer plays any role in defining the operating state of the system. The Resource.System.Product field must be used to specify the desired operating state. If the Product field is not specified and a hardware system is present, live acquisition from the hardware system will be the operating state selected by default. If a hardware system is not available, the system will default to the Software-Only operating state.

3. Resource.System Structure

Resource.System is a new structure that can be included in a user's SetUp script, to provide a very simple, high-level way of identifying what system configuration the script is intended for. This serves the dual benefits of making it easier for the system software to determine exactly what system configuration the script requires, and also to communicate that intent to other people who may be using the script or wanting to modify it. In many cases use of the Resource.System structure is optional, but there are situations where it is required (One such example is the UTA 160-SH and UTA 160-DH adapter modules. The same probe could be used with either of these modules, but the behavior of a script for that probe will be very different depending on which UTA module is being used. There is no way the system can unambiguously determine from the other settings in the script which UTA module is intended, so in this case Resource.System.UTA is required to explicitly identify the UTA module. If this field is not defined in a script using the Hypertac connector, VSX will exit with an error indicating that the Resource.System.UTA field must be specified.)

The Resource.System structure also makes it very easy to identify other aspects of the system configuration that is intended for a user script. For example, if you intend the script to be used with a High Frequency system you can simply set the field Resource.System.Frequency to 'High Frequency'. In earlier Vantage software releases there was no clearly defined mechanism for specifying the intended frequency range.

• Field Definitions in 'Resource.System' structure

The following fields are recognized by the system in the 4.0.0 software release. All of these fields are optional, except in those cases where one is required to fully specify the system configuration required by the script.

- **Resource.System.Product** A string variable identifying the product configuration of the Vantage hardware system, such as 'Vantage 256' or 'Vantage 64 LE'. When the script is intended to be used without a hardware system, this is indicated by setting the Product field to 'SimulateOnly' or 'SoftwareOnly'. VSX will search the string for the shortest possible segments needed to clearly identify the system, to make that identification less dependent on exactly what spelling is used by the author. For example, to identify a Vantage 64 LE system VSX will search separately for the following two string segments: '64' and 'LE'. This ensures that all of the following plausible names for this configuration will be recognized successfully: "Vantage 64 LE", "V-64-LE", "vantage 64LE", or even just "64LE". The recognized Resource.System.Product values are:
    - SimulateOnly
    - SoftwareOnly
    - Vantage 32 LE

  - Vantage 64
  - Vantage 64 LE
  - Vantage 128
  - Vantage 256
- **Resource.System.Frequency** A string variable to identify the system's acquisition module configuration. The following acquisition module types are supported by the 4.0.0 release:
  - Low Frequency
  - Standard Frequency
  - High Frequency
  - HIFU
  - External Clock

  Alternate spellings such as 'Low' or 'LF' are also accepted. If this field is not specified, Standard Frequency will be assumed by default. Note also that 'HIFU' and 'External Clock' are special versions of the Standard Frequency configuration and are not available with Low Frequency or High Frequency.
- **Resource.System.UTA** A string variable identifying the marketing name of the UTA module to be used, such as "160-DH/32 LEMO". This is another case where VSX searches for the shortest possible string segment needed to uniquely identify the module- in this case just '160-DH'. The following UTA modules are supported by the 4.0.0 release (to avoid ambiguity, the spelling listed here should be used in all documentation, comments in source code, variable names, etc.):
  - UTA 260-S
  - UTA 260-D
  - UTA 260-MUX
  - UTA 360
  - UTA 408
  - UTA 408-GE
  - UTA 160-DH/32 LEMO
  - UTA 160-SH/8 LEMO
  - UTA 128 LEMO
  - UTA 64 LEMO
  - UTA 160-SI/8 LEMO
  - UTA 156-U
  - UTA 256-Direct
  - UTA 1024-MUX
- **Resource.System.SoftwareVersion** The minimum vantage software version required for a script to run, expressed as a 1 X 3 array of Matlab doubles set to the integer values of the Vantage software release level. For example, Vantage-3.4.3-1807021300 would be identified as

  ```
  Resource.System.SoftwareVersion = [ 3 4 3 ];
  ```

  If this optional field is present, VSX initialization will compare it to the actual Vantage software version being used and exit with an error if the system software version is older than the level specified by Resource.System.SoftwareVersion.

4. Dynamic Mux Programming for HV-Mux Probes

This section provides a high-level overview of the new "Dynamic Mux Programming" feature as implemented in 4.0.0, and how it differs from the original "Active Aperture" Mux programming scheme used in all previous releases. The 4.0.0 software provides full support for both the Dynamic Mux and Active Aperture programming models, so a user can freely choose to use either approach, based on what is most effective for the specific application and probe geometry being used in a script. The Active Aperture programming model is fully backward compatible to existing SetUp scripts from earlier releases using that approach.

Active Aperture programming allows access only to a predefined set of contiguous apertures selected through the HVMux switches, with TX and Receive Apod arrays mapped only to the elements within the selected aperture. Dynamic Mux programming allows access to any arbitrary Aperture that is physically possible to be selected by the HVMux switches, through Apod arrays that include all elements in the probe (the HVMux aperture selection is derived from the non-zero entries in the Apod array).

An HVMux script (either for an HVMux Probe, or for a UTA module that uses HVMux element switching) identifies itself as using the Dynamic Mux programming feature by defining all TX and Receive Apod arrays with length equal to Trans.numelements. The Active Aperture programming model is selected by defining all Apod arrays with length equal to the number of system channels being used by the probe. In either case, all Apod arrays used within a SetUp script must be the same length, and that length must match one of these two values. Any Apod array definition that violates these constraints will result in an error condition.

Using the Dynamic Mux Programming Model: When using Dynamic Mux programming, the user can define an Apod array for each individual TX and Receive structure to enable any arbitrary combination of elements that can be accessed by the HVMux switches, since the Apod array has access to all elements in the probe. After defining the Apod array the 'computeMuxAperture' function is called:

```
aperture = computeMuxAperture(Apod, Trans);
```

The input argument 'Apod' is the Apod array that has just been defined, and 'Trans' is the Trans structure being used in the script. 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. The user should assign the returned aperture value to the associated TX.aperture or Receive.aperture field in their 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 addition to using the "computeMuxAperture" function to create the Aperture array and associated aperture index value, the user has two alternative ways to set the required aperture value:

- If the user knows the Apod array they have just defined is supported by one of the precomputed Apertures provided by computeTrans, they can manually set the aperture field to that known Trans.HVMux.Aperture index value.
- If the user knows the Apod array they have just defined is supported by one of the Apertures that has previously been created by "computeMuxAperture" for this script, they can re-use the aperture index associated with that Aperture.

When either of these alternatives is used, there is no need to call the computeMuxAperture function.

An additional option available to the user through their SetUp script is to delete the predefined set of apertures provided by the computeTrans function if they will not be used in the script and all apertures will instead be created through calls to "computeMuxAperture". In this case just insert the following lines in the SetUp script after the Trans structure has been fully defined but prior to the first call to computeMuxAperture:

```
Trans.HVMux.ApertureES = [];
Trans.HVMux.VDASAperture = [];
```

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 the user 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.

Existing "Active Aperture" Mux functionality: The 4.0.0 software is fully backward compatible to this programming scheme, as used in all previous Vantage software releases. The Active

Aperture programming scheme was 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. The HVMux switching is arranged so any contiguous group of elements could be selected across the full width of the transducer array, in “tractor tread” fashion. For example, the L12-3v is a 192 element HVMux probe, to allow use with a 128-channel system. The Mux switches provide 65 different apertures, where aperture 1 selects elements 1:128; aperture 2 selects elements 2:129; …; and aperture 65 selects elements 65:192. 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.
- 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 within that Aperture.
- Apod arrays used in the script 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.

5. "States" Array Transmit Waveform Definition

In Vantage 4.0.0 we are replacing the "PulseCode" representation of all transmit waveforms with "States", a simpler and more flexible mechanism for defining a waveform with more degrees of freedom than were possible with the PulseCode format. The primary goals of the new States array format are to simplify transmit waveform definition and analysis and to allow more flexibility in creating arbitrary waveforms by allowing loops over any desired waveform segment and the ability to nest loops with independent start and end points (the earlier PulseCode format could only apply a repeating loop to a single row of the array and did not allow nested loops). From the perspective of the overall system software functionality in processing transmit waveforms through the TW and TX structures, the States array is simply a drop-in replacement for PulseCode.

To support the new "States" representation, the 4.0.0 software includes a redesigned utility function for converting the States array that defines a waveform into the waveform descriptor format used by the hardware system. This new utility will encode waveforms more efficiently by detecting and exploiting any repeating patterns that may exist in the waveform, and will also complete the waveform encoding process much faster than in earlier releases.

All of these changes are fully backward compatible to existing user scripts and utility functions that were based on the "PulseCode" waveform definition. Any user script that defines a waveform in the PulseCode format will function properly with 4.0.0; the 4.0.0 software will automatically translate the PulseCode array into the States format for use by the rest of the system.

• States Array Definition

The ***States*** array as used in the Matlab workspace is an N X 2 array of Matlab doubles. The "***States***" format is motivated by the recognition that the simplest way to characterize the three-level output of the Vantage transmitter is as a sequence of states, where each state has only two attributes:

- The output *level*, either +HV, ground, or –HV (signified in the ***States*** array by the values +1, 0, or -1 respectively).
- The output *duration*, a positive integer value representing the number of system clock cycles over which this state persists.

The two entries in each row of a ***States*** array are these two values: *level* in column 1, restricted to the values -1, 0, or 1 and *duration* in column 2, any positive integer value. Note that a particular hardware system configuration may impose constraints on the maximum duration of the individual states, but these limits will be enforced by the system software- for the ***States*** array itself there is no fixed upper or lower limit on the duration of an individual state.

A waveform state row with a *duration* value that is zero or negative will generate an error condition, preventing the creation of a transmit waveform. If a waveform definition algorithm has generated zero-duration "placeholders" as part of the waveform synthesis process, any zero-duration states that remain when the process is complete must be removed. This requirement allows the logic interpreting the States array to be simpler and will make execution faster. It will also maximize the chances that a waveform definition typo or programming error on the user's part will be detected, rather than allowing the system to function with a waveform that is not what the user intended.

Commands

Individual rows of the ***States*** array as defined above specify individual states of the waveform output; any number of rows can be used to define a waveform segment. If a waveform includes a segment that is to be repeated multiple times, command rows are inserted in the ***States*** array just prior to and immediately after the waveform segment to be repeated, defining the start and end of the "loop" (similar to a "for … end" loop in Matlab code). This approach supports the two objectives that a loop can be applied to a waveform segment of any desired duration, and that nested loops are supported as well.

A "command row" in the ***States*** array is identified by the first entry in the row. If this entry is any value other than -1, 0, or 1 the row will be interpreted as a command and not as a waveform state. For each command, the second entry in the command row is used as an argument for that command. At present only three commands have been defined: "***LoopStart***", "***LoopEnd***", and "***WaveformEnd***". (Future iterations of the Vantage system may add additional commands to support new waveform generation features). Each command is encoded as an integer value to be used as the first entry in a ***States*** command row. If the first entry in a row of the ***States*** array is not a recognized command value and also is not -1, 0, or 1 then an "Unrecognized Command" error will be generated, preventing the system from generating an output waveform.

- ***LoopStart*** command**:** An absolute value of 10 (i.e. the actual value can be either 10 or minus 10) in the first entry in a ***States*** row identifies the start of a repeating waveform segment loop. (See the 'waveform inversion' section below for the reason to use absolute value.) The second entry in the ***LoopStart*** command row is the loop count for the associated repeating segment loop, which must be a positive integer greater than 1, specifying the number of times the following segment is to be repeated. The end of the repeating loop segment is marked by a ***LoopEnd*** command of the same loop level that was created by the ***LoopStart*** command. After the specified number of loop iterations have been completed, execution moves on to the next row in the ***States*** array following the ***LoopEnd*** command, and the loop level will be decremented by 1. A ***LoopStart*** command argument value of 1, 0, any negative value, or any non-integer value will trigger an "Unrecognized Loop Count" error condition, preventing the system from generating an output waveform

- ***LoopEnd*** command**:** An absolute value of 20 in the first entry in a ***States*** row identifies the end of the waveform segment associated with a previous ***LoopStart*** command. The repeating waveform segment is identified as all waveform states (and possibly nested loops) within the ***States*** array that come after the most recent ***LoopStart*** command whose resulting loop level matches the specified loop level in the ***LoopEnd*** command. Nested loops are supported by multiple ***LoopEnd*** commands, just as would be the case in Matlab code with nested "for" loops each with its own "end" statement. The argument for the ***LoopEnd*** command must be an integer value in the range from one to four, identifying the level of nested loop that this particular ***LoopEnd*** command is ending. A loop level of zero means no ***LoopStart*** commands are currently active at this point in the waveform; when a ***LoopStart*** command is encountered the loop level will increment by 1, and must match the argument value for the associated ***LoopEnd*** command. If another ***LoopStart*** command is encountered before a ***LoopEnd*** command we have a nested loop and the loop level will be incremented again, and this value must be reflected in the argument of the associated ***LoopEnd*** command. The maximum loop level is four because the existing CGD FPGA waveform generator provides support for up to four levels of nested loops. A ***States*** array that attempts to define more than four levels of nested loops, or assigns a Loop command argument value that does not match the "incrementing loop level" described above will result in an error condition.

- ***WaveformEnd*** command**:** An absolute value of 30 in the first entry in a ***States*** row identifies the end of the entire waveform. The argument value for the ***WaveformEnd*** command must be set to zero; any other value will result in an error condition. If the loop level generated by the waveform compiler while parsing the ***States*** array is not at zero when a ***WaveformEnd*** command is encountered an error condition will result, preventing generation of the waveform. Any entries in the States array following the ***WaveformEnd*** command will be ignored, regardless of their values. If the end of the waveform is at the end of the States array the ***WaveformEnd*** command is optional; reaching the end of the States array will be interpreted as an implicit "***WaveformEnd***".

In a legitimate States array using ***LoopStart*** and ***LoopEnd*** commands as defined above, there must be precisely one ***LoopEnd*** command for every ***LoopStart*** command within the overall ***States*** array. The ***LoopEnd*** command argument identifying the loop level is intended to help identify user errors when composing a waveform, and thus minimize the chance of actually transmitting an incorrect waveform. If the waveform compiler encounters a ***LoopEnd*** command whose loop level argument value does not match the expected value while parsing the ***States*** array, or if the end of the ***States*** array is reached with a loop level greater than zero, the software will exit with an error condition and no waveform will be generated.

Waveform Inversion

With the States array defined as stated in the above paragraphs, it now is trivial to invert a previously defined waveform: just multiply the first column of the entire States array by

negative 1, since this will invert the level of all active pulses but have no effect on the zero-level states. This does impose an additional requirement on identifying and interpreting command values, however- since the entire first column will have been inverted, a first entry value of either 10 or -10 must be interpreted as a valid ***LoopStart*** command, and similarly a value of either 20 or -20 is a valid ***LoopEnd*** command, and either 30 or -30 is a valid ***WaveformEnd*** command. Any function that is intended to interpret the ***States*** array must produce one of the following thee responses based on the value it finds in the first entry of each row:

- If the value is -1, 0, or 1 this is a waveform state, with a duration set by the second entry in that row (or an error condition if the second entry is not a positive integer)
- If the value is one of the six values currently recognized as command codes (+/-30, +/-20, +/-10) this is a command row (or an error condition depending on the argument value and the context in which the command occurs).
- Any value other than the nine listed above must be interpreted as an error condition, preventing any output waveform from being generated.

• Transmit Waveform Definition Changes in 4.0.0

This section provides an overview of the Vantage software functionality and data flow for transmit waveforms, from definition in a user's SetUp script through to execution on either the hardware system or in simulation. The overall behavior is identical to earlier software releases, preserving backwards compatibility for existing user scripts. The 'Parametric' and 'Envelope' waveform types will be converted directly into the new States waveform format, and a PulseCode array will not be generated. But if the user script has supplied a waveform in the PulseCode format, it will automatically be converted into an equivalent States waveform array for use by the rest of the system software.

In a user script, transmit waveforms for use with the system are defined in the TW structure through any one of following formats as identified by the TW.type field: 'Parametric', 'Envelope', 'PulseCode', or 'States'. The Parametric and Envelope types provide a simple, high-level waveform definition that is converted into a States array by the system software through the "computeTWWaveform" function (invoked by VSX through a call to VsUpdate('TW')). The 'PulseCode' TW.type allows a user to define the waveform directly, or through their own waveform generation algorithm (PulseCode is also used as the output format from our Arbwave toolbox). Similarly, the new 'States' TW.type will allow a user (or user-supplied algorithm) to define a waveform directly in the States format. For all TW types, the user can either define a single waveform that will be applied to all active channels or a per-channel array with a unique

waveform definition for each active channel (i.e. there must be a one-to-one relationship between the array of defined waveforms and all of the entries in the TX.Apod array).

Regardless of how the transmit waveform was originally defined, it is always converted into the universal States array format for use by the system. After completing this conversion, the computeTWWaveform function will then use it to synthesize the transmit waveform to be used in simulation, and also to derive several waveform characteristics that are used by TXEventCheck and other system utilities: estimatedAvgFreq, burst duration ('Bdur'), pulse count (Numpulses) and gate driver cycle rate, peak transmit current ('chIpk1V'), transformer fluxHVlimit, etc.

Next, the VsUpdate('TX') function creates the actual per-channel transmit waveform definitions that will be sent to the hardware system for use in Events using that TX, based on one of three approaches depending on the TW structure that is indexed by TX, and the TX.Apod values that the user script has provided:

1. If the TW structure has provided a per-channel array of waveforms, the TX.Apod array will be interpreted as a logical (any non-zero entry enables that channel, and a zero entry disables the associated channel). The per-channel waveforms provided by the TW structure are applied directly to the associated channels (using the same mapping to channels that applies to TX.Apod and TX.Delay) with the only exception being that any disabled channels per TX.Apod will have a "null waveform" (no transmit output) replacing the TW waveform table for that channel.
2. If the TW structure has provided a single transmit waveform for use by all active channels, and the TX.Apod array only has the values one and zero to enable/ disable individual transmit channels, then the TW waveform will be applied directly to all channels that are enabled by TX.Apod and a null waveform to all channels that are disabled.
3. If the TW structure has provided a single waveform and the TX.Apod array contains values other than 0 and 1 in the range from -1 to 1, then the system will use the TX.Apod values to apply pulse-width modulation to the TW waveform definition array as well as polarity inversion if TX.Apod is negative. The net result is that a unique transmit waveform for each active channel will be created from the 'prototype' waveform provided by TW, based on the value of TX.Apod for that channel. A null waveform will be sent to the hardware for any channels disabled by TX.Apod (i.e. TX.Apod values of zero).

- **A Simple Example, PulseCode VS States**

This example illustrates how a typical TW parametric waveform definition would be represented in both the PulseCode and States formats:

```
TW.type = 'parametric';
TW.Parameters = [5,  0.8,  200,  1];
```

The above two lines define a 100 cycle 5 MHz waveform with relative pulse width of 0.8, plus equalization pulses. The 3.4 software will translate this into the following PulseCode array:

```
TW.PulseCode =  [  1    -10    10    20     1 ; …
                   5    -20     5    20   99; …
                   5    -20    10    10     1 ];
```

The 4.0.0 software will translate this same waveform into the following States array:

```
TW.States = [   0       1; …
               -1      10; …
                0       5; …
               10     100; …     % loop start command with repeat count of 100
                0       5; …
                1      20; …
                0       5; …
               -1      20; …
               20       1; …     % loop end command for loop level 1
                0      10; …
                1      10; …
               30       0 ];     % waveform end command at loop level 0
```

Even though the mechanism for implementing the repeat loop is quite different, a comparison confirms the PulseCode and States arrays do indeed define the exact same tri-level waveform.

The 4.0.0 software includes a utility "PulseCode2States" that can be used to convert any valid PulseCode array into an equivalent States array.

6. Changes to Trans Structure Use and Functionality

Numerous changes have been made to the Trans structure and how it is used in the Vantage 4.0.0 software release, to support the new features of 4.0.0 including:

- Dynamic HVMux programming, allowing much greater flexibility for the user in defining Apertures for use with HVMux probes and the HVMux UTA modules, and to simplify the effort needed to integrate a customer-defined HVMux probe.
- New UTA modules which led to the need for a more flexible yet consistent mechanism for managing connector selection and element-to-channel mapping.
- Better support for Vantage 32 LE and Vantage 64 probes and scripts using the UTA 260-Mux adapter module.
- Use of the UTA 1024-MUX adapter module with 2D Matrix Array probes.
- More flexible support for “SimulateOnly” and “SoftwareOnly” operating states.

• Element to Channel Mapping Through Trans and UTA

This section defines the variable names and mapping arrays used to specify the relationship between elements within the transducer, element signals at the connector, and channels within the system. With the growing number of UTA modules, including several cases where multiple modules support the same connector type and probe family but with different features and different mapping of element signals to channels, we have recognized the need to more clearly and consistently define the two layers of element signal mapping that occur in the system: First the mapping within a probe from transducer elements to signals at the probe connector, and second the mapping from element signal pins at the probe connector to system channels through the UTA modules.

Definitions

- **Elements (“EL”)**: Sequential numbering of elements within an array transducer, and/or for multiple individual transducers connected to the system. In the Matlab workspace, element indices start at 1 but in C-language code such as runAcq they usually start at zero. Note also that while most probe manufacturers number their elements from 1 there are several that start at zero. For the Vantage system software, our convention is to add one to the element numbers for the ‘zero-based’ manufacturers so all Vantage documentation and connector pin assignment listings will start at 1. The order of element numbering is arbitrary, but is usually arranged in a way that makes sense for the physical design of the probe. The 4.0.0 Vantage software can support element counts up to 1024. Regardless of how the element numbering sequence has been assigned, the Trans.elementPos array must correctly identify the relative position and orientation of each individual element within a transducer assembly.
- **Element Signals**, (“**ES**”): Individual element signals as identified at the probe connector interface. Usually there is a one-to-one mapping from elements to element signals, but

there are many exceptions such as probes whose element count is less than the number of element signals at the connector, or HVMux probes where more than one element is mapped through the HVMux switches to the same element signal. For probe connector interfaces that are intended to support a commercially available family of probes, Verasonics will use the element signal assignments at the connector as defined by the manufacturer (or the manufacturer's element numbering plus one if they start at zero).

- **System Channels**, ("**CH**"): The physical channels of the Vantage hardware system, numbered sequentially starting at one. The channel number also identifies the column number of the Receive Data buffer that contains the RF data samples from that channel. For current 4.0.0 Vantage software releases system channels must be enabled in contiguous groups of 32 representing all channels of a Channel Group in the hardware system, even if not all of those channels are actually being used by the probe. As a result, the number of active system channels may be greater than the number that are actually connected to probe elements. Unused channels will be automatically disabled by the system on both transmit and receive, but they will still be represented in the receive data buffer since there is always a one-to-one mapping between active system channels and columns of the receive data buffer (for a disabled channel, the data in the associated receive buffer column will be set to all zeros).

Mapping from Elements to Element Signals to System Channels

- **Trans.ConnectorES** defines the mapping within a probe from transducer elements to the element signals as defined at the system connector interface. In the Matlab workspace Trans.ConnectorES is a column vector of Matlab doubles, of length equal to the number of elements in the transducer. The index to an entry in Trans.ConnectorES is the Element number, and the integer value at that index is the Element Signal number to which that Element is connected. If the transducer definition includes elements that are not connected to the system, the Trans.ConnectorES value for those elements will be set to zero. As has been the case in prior releases, Trans.ConnectorES is a required field but if it is not specified by the user VSX will create it with a default one-to-one mapping (but only if the number of Elements is equal to the number of Element Signals at the connector; otherwise the absence of Trans.ConnectorES will trigger an error condition). For an HVMux probe more than one element will be mapped to the same element signal, representing the possible connections through the HVMux switch array. For the special case of a probe that connects several system channels together in parallel to drive the same element (such as may be done for a HIFU therapy transducer), Trans.ConnectorES will be a two-dimensional array where the column index is the Element number and the entries in each row identify all the Element Signals at the connector that are wired in parallel to that element.
- **UTA.TransConnector** defines the mapping through the UTA module from Element Signals at the probe connector to the system Channels (and to columns of the Receive data buffer in memory on the host computer). This mapping is always provided by the

computeUTA function; note that the actual mapping can change depending on the system configuration, element count, and which connectors are being used.

- **Trans.Connector** defines the composite mapping through Trans.ConnectorES and UTA.TransConnector, from transducer Elements to system Channels. This is a "dependent" or "read-only" field that is created by the system software during script initialization and added to the Trans structure for use by the rest of the system (and also to make it easy for the user to see what the composite mapping actually is, after they have run the script). If this field is defined by the user's script, it will simply be ignored and overwritten with the values created by the system.

For HVMux probes, the same two levels of mapping will be applied in the same way:

- **Trans.HVMux.ApertureES** is the mapping defined by the HVMux probe, from individual Elements to Element Signals at the connector for each aperture selection that has been defined for the HV Mux switches. Each column of Trans.HVMux.ApertureES is a direct copy of Trans.ConnectorES, but with the Element entries that are not included in the associated Aperture selection set to zero.
- **Trans.HVMux.Aperture** is the composite mapping created during system initialization, representing the overall mapping (through the probe and the UTA module) from Elements to system Channels for each aperture selection in the same fashion as Trans.Connector for non-Mux probes.

Backward Compatibility Issues:

In software releases prior to 4.0.0, the composite mapping from transducer elements through the UTA module to system channels was usually defined by the Trans.Connector field (or by Trans.HVMux.Aperture for HV Mux probes). As a result, the content of Trans.Connector in earlier releases was equivalent to Trans.Connector as now defined for 4.0.0 (however, there are a few situations where this is not the case, such as for scripts using the Hypertac connector on the UTA 160-DH/32 LEMO). If you have a script written for an earlier release that uses the computeTrans function to create the Trans structure, then it will be fully compatible with 4.0.0 since the new computeTrans in 4.0.0 will create Trans.ConnectorES and then VSX initialization will use that to create Trans.Connector as described above. But for backward compatibility to scripts that defined the Trans structure explicitly and did not use computeTrans, the script will have created Trans.Connector and not Trans.ConnectorES. VSX will recognize this situation and assume the Trans.Connector field actually represents the full composite mapping from elements to system channels and use it as-is, preserving backward compatibility. Exceptions such as the Hypertac connector mentioned above will not match this assumption; in those cases you will have to modify the script to correctly define Trans.ConnectorES to get it to run properly with 4.0.0.

- ## Trans Structure Field Definitions

The following changes and additions have been made to the Trans structure, to support the dynamic Mux programming features:

- **Trans.Connector (and Trans.ConnectorES, see description above):** In earlier software releases, the Connector field was not used in scripts using HVMux switching; the Trans.HVMux.Aperture field took its place. The presence of Trans.Connector in an HVMux script would trigger an error condition. For dynamic mux programming in 4.0.0 the Trans.ConnectorES field is now required, to specify the mapping from every element through the HVMux switches to Element Signals at the connector. This field is used by computeMuxAperture to create the columns in Trans.HVMux.Aperture that define a particular aperture and HVMux switch setting.
- **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 will 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.type:** This field contains a string variable identifying the type of HVMux switching that is in use, which in turn allows the system to decide which features can be used. It currently supports three values:
    - **'preset'** Identifies an HVMux design that does not allow use of the dynamic mux programming feature. One example of this type is the GE L3-12D probe; the available apertures for this probe are defined through an FPGA in the probe and thus we cannot define an aperture it does not already support. When the system finds the 'preset' type it will disable the dynamic mux programming features and require that the script include predefined aperture selections in all TX and Receive structures. An error will be reported if the script attempts to use dynamic programming. If Trans.HVMux.type has not been specified, the 'preset' type will be set by default.
    - '**perEL**' Identifies an HVMux design that provides a separate bit for programming each element's switch in the HVMux chips, thereby allowing the selection of more than one element in parallel connected to the associated system channel. When this type is present, the system will accept and use Apod arrays that connect elements in parallel.
    - **'perCH'** Identifies an HVMux design that cannot select elements in parallel, since both states of each bit in the mux programming table are used to select a different element. The "L12-5_50mm" probe is an example of this constraint. Dynamic mux programming can be used with the 'perCH' type, but any Apod

array that selects elements in parallel will be detected and will trigger an error condition.

- **Trans.id:** As was the case in previous software releases, Trans.id is a Matlab double set to the 24-bit unsigned integer value representing the ID code of the associated probe and thus valid Trans.id values will be in the range from zero to (2^24 – 1). For the 4.0.0 and future software releases, Trans.id can also be set to the value -1 to indicate that there is no id for this probe (or that the id should be ignored if the probe provides an id value). If the Trans.id value provided by the user script is minus one, VSX will ignore the ID value read from the probe and proceed to run the script without any ID check at all. This duplicates the VSX behavior in previous releases if Trans.name had been set to the special value 'custom'. By using the minus one Trans.id value instead of 'custom' for the name, you can get the same ignore-the-ID behavior for your script while still providing a meaningful probe name in Trans.name. This new feature will allow you to support multiple probes that do not have unique ID's through computeTrans and still be able to differentiate them through the Trans.name field. The 3.5-4.0 software will still recognize the Trans.name of 'custom' as another way to indicate the probe ID read through the connector should be ignored.